Object-Oriented Programming (OOP) with Java

By:
Richard Baldwin

Object-Oriented Programming (OOP) with Java

By:
Richard Baldwin

Online:
< http://cnx.org/content/col11441/1.121/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: May 7, 2013
PDF generated: May 28, 2013
For copyright and attribution information for the modules contained in this collection, see p. 1742.

Table of Contents

Chapter 1

Preface

1.1 Jy0010: Preface to OOP with Java[1]

1.1.1 Table of Contents

1.1.2 Welcome

Welcome to my collection titled *Object-Oriented Programming (OOP) with Java* .

During the past eighteen years, I have published hundreds of Java and OOP programming tutorials on a variety of different topics and websites. I have also developed the teaching materials for several different college-level programming courses in Java/OOP.

A work in progress

This is a work in progress. I am currently combining selected content from those earlier endeavors with new material that I am developing to create a freely downloadable E-book that covers Java/OOP programming from programming fundamentals to very advanced OOP concepts.

Among other things, the collection contains the material that I use to teach the following three courses at Austin Community College in Austin Texas:

- ITSE 2321 - Object-Oriented Programming (Java) [2]
- ITSE2317 - Java Programming (Intermediate) [3]
- GAME 2302 Mathematical Applications for Game Development [4]

The collection also includes:

- An OOP self-assessment test [5]

[1] This content is available online at <http://cnx.org/content/m45136/1.9/>.
[2] http://cnx.org/content/m45222
[3] http://cnx.org/content/m45258
[4] http://cnx.org/content/m45315
[5] http://cnx.org/content/m45252

- Course materials for a complete course in Programming Fundamentals [6]

Because it is a work in progress, the collection is growing on a daily basis. If you don't find what you need today, come back and take another look in a week or two and you may find what you need then.

Download options

I encourage you to take advantage of all of the download options *(most of which are free)* that cnx.org [7] has to offer. You can also customize this material for use in your organized courses or for personal self study.

Feedback is appreciated

And if you find the material useful, I would like to hear more about how you are using it.

1.1.3 Getting started with Java programming

As is the case with most worthwhile endeavors, Java programming requires that you have some tools to begin. Fortunately, all of the tools that you need to get started programming in Java are available for free downloading.

In addition to a computer with web access, you will need:

- The Java Development Kit *(JDK)* and Java Runtime Engine *(JRE)*
- The Java API documentation
- A suitable text editor

1.1.3.1 The JDK and the JRE

The JDK, the JRE, and the API documentation are all freely available from Oracle. As of December 2012, you will find links to that material on the web page titled *Java Platform Standard Edition 7 Documentation* at http://docs.oracle.com/javase/7/docs/ [8] *(The links given in this module may change as new versions of Java are released, but newer versions shouldn't be too difficult to locate with a web search.)*

Download

The JDK and the included JRE can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/inde [9] That page provides several download options. Beginners should download the ***Java Platform (JDK)*** for the latest released version. *((The JRE is included in the JDK package, so you don't need to download both.)*

Install

You will also probably need to follow the installation instructions for your computer that are available at http://docs.oracle.com/javase/7/docs/webnotes/install/index.html [10] Pay particular attention to the instructions for setting the *path* and *classpath* environment variables. This is where many students stumble. Another useful document on the path and classpath is available at http://docs.oracle.com/javase/tutorial/essential/environment/paths.html [11]

1.1.3.2 The Java API documentation

The Java Platform, Standard Edition 7 API Specification is available at http://docs.oracle.com/javase/7/docs/api/index.html [12]

Also see my Java OOP documentation [13] module for instructions on how to use the documentation.

[6] http://cnx.org/content/m45179
[7] http://cnx.org/
[8] http://docs.oracle.com/javase/7/docs/
[9] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[10] http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
[11] http://docs.oracle.com/javase/tutorial/essential/environment/paths.html
[12] http://docs.oracle.com/javase/7/docs/api/index.html
[13] http://cnx.org/content/m45117

1.1.3.3 A suitable text editor

The module titled Jb0110: Java OOP: Programming Fundamentals, Getting Started [14] explains how to use a text editor to create Java program code. Just about any text editor will do as long as you can ensure that the file name extension is .java. Something as simple as *Windows Notepad* or *Windows WordPad* would probably be best for your first few simple programs.

Soon, however, you will probably want to upgrade to an editor that uses different colors to identify the different parts of your program. My favorite color-coded editor is the free version of JCreator [15] . *(The free version seems to have disappeared from their web page so you may have trouble finding it.)*

Another free editor is DrJava [16] . An advantage of this editor is that it can be run from a USB drive with no installation required. Another possibility, although I have never had occasion to use it, is jGRASP [17] . Numerous other Java color-coded editors, including BlueJ [18] are available for free downloading on the web.

1.1.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0010: Preface to Object-Oriented Programming (OOP) with Java
- File: Jy0010.htm
- Published: 11/16/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[14] http://cnx.org/content/m45137
[15] http://www.jcreator.com/
[16] http://drjava.sourceforge.net/
[17] http://www.jgrasp.org/
[18] http://www.bluej.org/

Chapter 2

Objects First

2.1 Gf0100: Objects First with Greenfoot[1]

2.1.1 Table of Contents

2.1.2 Preface

If you are new to programming and want to learn how to program, or if you have programming experience but are new to *object-oriented programming (OOP)* and want to learn OOP, I can't think of a better way to take that first step than through the use of Greenfoot [2] .

Greenfoot provides a set of interactive visual tools coupled with the Java programming language that makes learning to program both interesting and productive.

(Note that descriptions of the Greenfoot website contained herein are current as of 02/19/13.)

[1] This content is available online at <http://cnx.org/content/m45790/1.1/>.

[2] http://www.greenfoot.org/door

2.1.3 Overview

2.1.3.1 A visual interactive programming environment

The greenfoot overview [3] page describes Greenfoot as an Interactive Visual World that *"...teaches object orientation with Java."*

When programming with Greenfoot, you *"Create 'actors' which live in 'worlds' to build games, simulations, and other graphical programs."*

Greenfoot is both instructive and enjoyable because it is visual and interactive. Visualization and interaction tools are built into the programming environment.

Unlike Scratch [4] and Alice [5] *(two programming environments for beginners that use a drag-and-drop approach to programming)* , Greenfoot actors *"are programmed in standard textual Java code, providing a combination of programming experience in a traditional text-based language with visual execution."*

According to Wikipedia [6] ,

"Greenfoot is an interactive Java development environment designed primarily for educational purposes at the high school and undergraduate level. It allows easy development of two-dimensional graphical applications, such as simulations and interactive games.

Greenfoot is being developed and maintained at the University of Kent [7] and La Trobe University [8] , with support from Oracle [9] . It is free software, released under the GPL license. Greenfoot is available for Microsoft Windows [10] , Mac OS X [11] , Linux [12] , Sun Solaris [13] , and any recent JVM [14] ."

Continuing with Wikipedia [15] ,

"Greenfoot aims to motivate learners quickly by providing easy access to animated graphics, sound and interaction. The environment is highly interactive and encourages exploration and experimentation. Pedagogically, the design is based on constructivist and apprenticeship approaches.

Secondly, the environment is designed to illustrate and emphasize important abstractions and concepts of object-oriented programming. Concepts such as the class/object relationship, methods, parameters, and object interaction are conveyed through visualizations and guided interactions. The goal is to build and support a mental model that correctly represents modern object-oriented programming systems."

2.1.3.2 Greenfoot is not a toy

Your first impression when you enter the Greenfoot home page [16] may be that Greenfoot is a toy, but that definitely is not the case.

While it is true that the first impulse of many beginning programmers is to create computer games *(as evidenced on the home page [17] and the Greenfoot Gallery [18])*, Greenfoot can also be used to create serious interactive simulations. As an example, I will point you to the following simulations:

- Ants [19] by mik

[3] http://www.greenfoot.org/overview
[4] http://scratch.mit.edu/
[5] http://www.alice.org/
[6] http://en.wikipedia.org/wiki/Greenfoot
[7] http://en.wikipedia.org/wiki/University_of_Kent
[8] http://en.wikipedia.org/wiki/La_Trobe_University
[9] http://en.wikipedia.org/wiki/Oracle
[10] http://en.wikipedia.org/wiki/Microsoft_Windows
[11] http://en.wikipedia.org/wiki/Mac_OS_X
[12] http://en.wikipedia.org/wiki/Linux
[13] http://en.wikipedia.org/wiki/Sun_Solaris
[14] http://en.wikipedia.org/wiki/Java_Virtual_Machine
[15] http://en.wikipedia.org/wiki/Greenfoot
[16] http://www.greenfoot.org/home
[17] http://www.greenfoot.org/home
[18] http://www.greenfoot.org/scenarios
[19] http://www.greenfoot.org/scenarios/1016

- Wave-Lab [20] by delmar
- Birds and Trees [21] by polle

(Click a link given above to download and run a simulation. When the simulation window appears, click the Run button on the simulation window to start the simulation running.)

You can view more than 7500 scenarios *(Greenfoot programs are commonly referred to as scenarios)* by clicking the **Scenarios** link at the top of the home page [22] . Some of the scenarios are impressive. Some are not so impressive. Many, possibly most, were written by beginning programmers.

(Note that you must have Java applets enabled on your computer to run these simulations. Alternatively, if you have Greenfoot installed on your computer, you can click the "Open in Greenfoot" button to cause the program to be downloaded for compilation and execution locally on your computer.)

2.1.3.3 The Greenfoot Gallery

What you see when you click the **Scenarios** link at the top of the home page [23] has been called the *Greenfoot Gallery* [24] in times past. This is a place where the authors of Greenfoot scenarios can publish their scenarios if they so choose.

This is a social network or virtual programming community where Greenfoot programmers gather to encourage one another and to critique the work being done by themselves and others. Note however that active participation in the gallery is completely voluntary.

2.1.4 Software

The Greenfoot software and the Java Development Kit (JDK), both of which run on Windows, Mac, and Linux, are available for free download [25] .

2.1.4.1 Software to download and install

When you visit the download [26] page, you will find download links for Windows, Mac OS X, and Ubuntu. Click one of those links to download and install the appropriate version of Greenfoot on your computer.

You will also need to download and install the Java Development Kit (JDK). There is a download button labeled **Download JDK** for that purpose.

2.1.4.2 Stand alone software

As an alternative to the version that can be installed on your computer, there is a self-contained **Stand Alone** version that you can download, unzip, copy to, and run either from a disk folder or from a USB memory stick with no other installation required. This version contains both the Greenfoot software and the Java JDK.

Once you extract the software from the zip file, you will need about 200 Mbytes to store it in a disk folder. Depending on the formatting, you may need as much as 275 Mbytes of available space to copy the software to a USB memory stick.

The stand alone version is particularly useful in several situations including the following:

- You want to run Greenfoot on a computer for which you don't have installation privileges, such as in a public library or a college computer lab.

[20] http://www.greenfoot.org/scenarios/597
[21] http://www.greenfoot.org/scenarios/267
[22] http://www.greenfoot.org/home
[23] http://www.greenfoot.org/home
[24] http://www.greenfoot.org/scenarios
[25] http://www.greenfoot.org/download
[26] http://www.greenfoot.org/download

- You don't already have the Java JDK installed on your computer and would prefer not to go through the effort to download and install it.
- You want to run Greenfoot, but you don't want to install anything on your computer.

2.1.5 Running Greenfoot

2.1.5.1 The installed version

If you download and install Greenfoot and the Java JDK on your computer, you will have an opportunity to place a Greenfoot icon on your desktop. Just double-click the icon to start the Greenfoot program running. *(Although not necessary, it is probably best to install the JDK before installing Greenfoot.)*

2.1.5.2 The stand alone version

If you elect to use the stand alone version, you will extract the following folders from the zip file:

- BlueJ [27] *(a more advanced Java programming environment)*
- Greenfoot *(the programming environment of interest in this module)*
- jdk *(the Java Development Kit)*
- userhome *(not sure the purpose of this folder)*

You will find a file named **Greenfoot.exe** inside the **Greenfoot** folder. It will have an icon of a small green footprint. Double click this file to start the Greenfoot program running.

Once the program is running, you can select the *Greenfoot Tutorial* item on the **Help** menu to open a webpage containing a variety of educational tutorials. *(I will have more to say about this later.)* The **Help** menu also provides access to several other useful items, such as documentation for both Greenfoot and the Java JDK.

2.1.6 Textbook and tutorials

There is a textbook titled Introduction to Programming with Greenfoot - Object-Oriented Programming in Java with Games and Simulations [28] that you can purchase on Amazon for about $75.00. *(As usual, you can rent it for less or purchase a used copy for less.)*

There are also many free online tutorials [29] available to help you get started programming with Java and Greenfoot.

Whether you need the textbook, or whether the tutorials will suffice will depend on your background. My advice is to begin with the tutorials alone, and then purchase the textbook if needed.

2.1.6.1 The textbook

That having been said, I will comment on the textbook. If I had the luxury of designing a computer programming curriculum from scratch, I would choose a model very similar to the Greenfoot textbook.

In particular, unlike the programming curricula in many colleges and universities, I would begin with the big picture *(classes, interfaces, objects, methods, variables, etc.)* and work my way down to the more detailed aspects of programming *(selection, loops, operators, types, etc.)* .

(The programming curricula at many colleges and universities begin at the bottom and work their way up instead of beginning at the top and working their way down. As a result, many students become bogged down in details and either quit or fail and never get a chance to see the big picture of object-oriented programming.)

[27] http://www.bluej.org/
[28] http://www.greenfoot.org/book
[29] http://www.greenfoot.org/doc

Because I believe in the top down model on which the textbook is based, I can recommend this textbook as a very good way to get started learning computer programming in general and object-oriented programming in particular. *(I am not affiliated with the author or the publisher and receive no compensation from the sale of this textbook.)*

2.1.6.2 The tutorials

2.1.6.2.1 Written tutorials

As of 02/19/13, the Greenfoot website provides the following written tutorials to help you learn how to use Greenfoot and begin programming:

- Interacting with Greenfoot [30]
- Movement and Key Control [31]
- Detecting and Removing Actors, and Making Methods [32]
- Saving the World, Making and Playing Sound [33]
- Adding a Randomly Moving Enemy [34]
- How to Access One Object From Another [35]

2.1.6.2.2 Video tutorials

If you prefer video tutorials, the Joy Of Code [36] is a thorough introduction to Greenfoot broken down into a large number of videos. A wide range of other short videos [37] are also available.

2.1.6.2.3 Other possibilities

Once you become an accomplished Greenfoot programmer, you might be interested in some of the following possibilities, which generally require other resources or more advanced knowledge:

- Kinect with Greenfoot [38] .
- PicoBoard with Greenfoot [39] .
- Sense Board with Greenfoot [40] .
- Finch with Greenfoot [41] .
- Loading native libraries with Greenfoot [42] .
- Gamepads with Greenfoot [43] .
- CS Unplugged with Greenfoot [44] .
- AP Computer Science with Greenfoot [45] .
- Learn Maths with Greenfoot [46]

[30] http://www.greenfoot.org/doc/tut-1
[31] http://www.greenfoot.org/doc/tut-2
[32] http://www.greenfoot.org/doc/tut-3
[33] http://www.greenfoot.org/doc/tut-4
[34] http://www.greenfoot.org/doc/tut-5
[35] http://www.greenfoot.org/doc/howto-1
[36] http://www.joyofcode.org/
[37] http://www.youtube.com/user/18km?ob=5#g/u
[38] http://www.greenfoot.org/doc/kinect
[39] http://www.greenfoot.org/doc/pico
[40] http://www.greenfoot.org/doc/sense
[41] http://www.greenfoot.org/doc/finch
[42] http://www.greenfoot.org/doc/native_loader
[43] http://www.greenfoot.org/doc/gamepad
[44] http://www.greenfoot.org/doc/csunplugged
[45] http://www.greenfoot.org/doc/ap
[46] http://sinepost.wordpress.com/

2.1.7 What comes next after Greenfoot

While this may sound like an exaggeration, programming with Greenfoot is sort of like riding a bicycle with training wheels. After awhile, you no longer need the training wheels and you are ready to move on to true two-wheeler.

While is it possible to push Greenfoot into some pretty complex and sophisticated scenarios, there comes a time when you need to take the training wheels off and ride that two-wheeler.

Numerous possibilities are possible in this regard. One possibility is BlueJ - The interactive Java environment [47] from the same folks who brought you Greenfoot. Like Greenfoot, the BlueJ software is available for free downloading. Also like Greenfoot, a textbook can be purchased and some free tutorials are available.

Another possibility is to continue working through the modules in this collection.

A very good possibility is to do both.

2.1.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Gf0100: Objects First with Greenfoot
- File: Gf0100.htm
- Published: 02/19/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[47] http://www.bluej.org/

Chapter 3

OOP Self-Assessment

3.1 Ap0005: Preface to OOP Self-Assessment[1]

3.1.1 Welcome

Welcome to my group of modules titled *OOP Self-Assessment* .

This is a self-assessment test designed to help you determine how much you know about object-oriented programming *(OOP)* using Java.

In addition to being a self-assessment test, it is also a major learning tool. Each module consists of about ten to twenty questions with answers and explanations on two or three specific topics. In many cases, the explanations are extensive. You may find those explanations to be very educational in your journey towards understanding OOP using Java.

To give you some idea of the scope of this self-assessment test, when you can successfully answer most of the questions in modules Ap0010 [2] through Ap0140 [3] , your level of knowledge will be roughly equivalent to that of a student who has successfully completed an AP Computer Science course in a U.S. high school up to but not including data structures. This is based on my interpretation of the College Board's Computer Science A Course Description [4] .

3.1.2 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0005: Preface to OOP Self-Assessment
- File: Ap0005.htm
- Published: 11/28/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

[1] This content is available online at <http://cnx.org/content/m45252/1.6/>.
[2] http://cnx.org/content/m45284
[3] http://cnx.org/content/m45302
[4] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

3.2 Ap0010: Self-assessment, Primitive Types[5]

3.2.1 Table of Contents

3.2.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

Questions and answers

The test consists of a series of questions (p. 13) with answers (p. 30) and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

Programming challenge questions

The module also contains a section titled Programming challenge questions (p. 20) . This section provides specifications for one or more programs that you should be able to write once you understand the answers to all of the questions. *(Note that it is not always possible to confine the programming knowledge requirement to this and earlier modules. Therefore, you may occasionally need to refer ahead to future modules in order to write the programs.)*

Unlike the other questions, solutions are not provided for the *Programming challenge questions* . However, in most cases, the specifications will describe the output that your program should produce.

Listings

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 29) to easily find and view the listings while you are reading about them.

[5]This content is available online at <http://cnx.org/content/m45284/1.4/>.

3.2.3 Questions

3.2.3.1 Question 1 .

What output is produced by the program in Listing 1 (p. 13) ?

- A. Compiler error
- B. Runtime error
- C. Hello World
- D. Goodbye Cruel World

Listing 1: Listing for Question 1.

```
    public class Ap001{
  public static void main(
                        String args[]){
    new Worker().hello();
  }//end main()
}//end class definition

class Worker{
  public void hello(){
    System.out.println("Hello World");
  }//end hello()
}//end class definition
```

3.1

Answer and Explanation (p. 35)

3.2.3.2 Question 2 .

What is the largest (algebraic) value of type int?

- A. 32767
- B. 2147483647
- C. -2147483647
- D. -32768

Answer and Explanation (p. 34)

3.2.3.3 Question 3 .

What is the smallest (algebraic) value of type int?

- A. -2147483648
- B. -2147483647
- C. 32767
- D. -32768

Answer and Explanation (p. 34)

3.2.3.4 Question 4 .

What two values are displayed by the program in Listing 2 (p. 14) ?

- A. -2147483648
- B. 1.7976931348623157E308
- C. -2147483647
- D. 4.9E-324

Listing 2: Listing for Question 4.

```
    public class Ap003{
  public static void main(
                        String args[]){
    new Worker().printDouble();
  }//end main()
}//end class definition

class Worker{
  public void printDouble(){
    System.out.println(
                      Double.MAX_VALUE);
    System.out.println(
                      Double.MIN_VALUE);
  }//end printDouble()
}//end class definition
```

3.2

Answer and Explanation (p. 33)

3.2.3.5 Question 5 .

What output is produced by the program in Listing 3 (p. 15) ?

- A. true
- B. false
- C. 1
- D. 0

Listing 3: Listing for Question 5.

```
    public class Ap004{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  private boolean myVar;
  public void printBoolean(){
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

3.3

Answer and Explanation (p. 32)

3.2.3.6 Question 6 .

What output is produced by the program shown in Listing 4 (p. 16) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 4: Listing for Question 6.

```
    public class Ap005{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar;
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

3.4

Answer and Explanation (p. 32)

3.2.3.7 Question 7 .

What output is produced by the program shown in Listing 5 (p. 17) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 5: Listing for Question 7.

```
    public class Ap006{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar = true;
    myVar = false;
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

3.5

Answer and Explanation (p. 32)

3.2.3.8 Question 8 .

The plus (+) character can be used to perform numeric addition in Java. What output is produced by the program shown in Listing 6 (p. 18) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. 2
- E. 1

Listing 6: Listing for Question 8.

```
    public class Ap007{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar = true;
    System.out.println(1 + myVar);
  }//end printBoolean()
}//end class definition
```

3.6

Answer and Explanation (p. 31)

3.2.3.9 Question 9 .

The plus (+) character can be used to perform numeric addition in Java. What output is produced by the program shown in Listing 7 (p. 19) ?

- A. Compiler Error
- B. Runtime Error
- C. 6
- D. 6.0

Listing 7: Listing for Question 9.

```
    public class Ap008{
  public static void main(
                        String args[]){
    new Worker().printMixed();
  }//end main()
}//end class definition

class Worker{
  public void printMixed(){
    double x = 3;
    int y = 3;
    System.out.println(x+y);
  }//end printMixed()
}//end class definition
```

3.7

Answer and Explanation (p. 31)

3.2.3.10 Question 10 .

The slash (/) character can be used to perform numeric division in Java. What output is produced by the program shown in Listing 8 (p. 20) ?

- A. Compiler Error
- B. Runtime Error
- C. 0.33333334
- D. 0.3333333333333333

Listing 8: Listing for Question 10.

```
    public class Ap009{
  public static void main(
                        String args[]){
    new Worker().printMixed();
  }//end main()
}//end class definition

class Worker{
  public void printMixed(){
    System.out.println(1.0/3);
  }//end printMixed()
}//end class definition
```

3.8

Answer and Explanation (p. 30)

3.2.4 Programming challenge questions

3.2.4.1 Question 11

Write the program described in Listing 9 (p. 21) .

Listing 9: Listing for Question 11.

```
/*File Ap0010a1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce the output shown
below:

ITSE
2321
**********************************************************/
public class Ap0010a1{
  public static void main(String args[]){
    System.out.println("ITSE");
    new Worker().doIt();
  }//end main()
}//end class definition
//=========================================================//

Class Worker{
  public void doIt(){
    System.out.println("2321");
  }//end doIt()
}//end class definition
//=========================================================//
```


3.2.4.2 Question 12

Write the program described in Listing 10 (p. 22) .

Listing 10: Listing for Question 12.

```
/*File Ap0010b1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Receives and displays an incoming parameter of type int.

The result should be similar to the following but the
values should be different each time the program is
run.

484495695
484495695
**************************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010b1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    int intVar = random.nextInt();

    //Student should understand the following
    int var = intVar;
    System.out.println(var);
    new Worker().doIt(var);
  }//end main()
}//end class definition
//============================================================//

class Worker{
  //----------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //----------------------------------------------------------//
}//end class definition
//============================================================//
```

3.10

3.2.4.3 Question 13

Write the program described in Listing 11 (p. 23) .

Listing 11: Listing for Question 13.

```
/*File Ap0010c1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that returns the largest value of type
int as type float.

The result should be 2.14748365E9
**********************************************************/
public class Ap0010c1{
  public static void main(String args[]){
    float val = new Worker().doIt();
    System.out.println(val);
  }//end main()
}//end class definition
//=========================================================//

class Worker{
  //-------------------------------------------------------//
  //Insert the method named doIt between these lines.
  //-------------------------------------------------------//
}//end class definition
//=========================================================//
```

3.11

3.2.4.4 Question 14

Write the program described in Listing 12 (p. 24) .

Listing 12: Listing for Question 14.

```
/*File Ap0010d1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Receives an incoming parameter of type double.
2. Converts that value to type int.
3. Returns the int

The result should be similar to the following but the
values should be different each time the program is
run.

6.672032181818181E8
667203218
**********************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010d1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    int intVar = random.nextInt();

    //Student should understand the following
    double var = intVar/1.1;
    System.out.println(var);
    System.out.println(new Worker().doIt(var));
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

3.2.4.5 Question 15

Write the program described in Listing 13. (p. 26)

Listing 13: Listing for Question 15.

```
/*File Ap0010e1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce an output similar
to that shown below. Note that the values should be
different each time the program is
run.

-1.30240579E8
-1.30240579E8
**************************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010e1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    double doubleVar = random.nextInt()/1.0;

    //Student should understand the following
    double var = doubleVar;
    System.out.println(doubleVar);
    new Worker().doIt(doubleVar);
  }//end main()
}//end class definition
//============================================================//

class Worker{
  public void doIt(double val){
    int var = val;
    System.out.println(var);
  }//end doIt()
}//end class definition
//============================================================//
```

3.13

3.2.4.6 Question 16

Write the program described in Listing 14 (p. 27) .

Listing 14: Listing for Question 16.

```
/*File Ap0010f1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code shown below, modify the
code in the method named doIt so that the program
displays

3.3333333333333335 instead of 3

Then modify the method again so that the program displays

3.3333333 instead of 3
**********************************************************/
public class Ap0010f1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=========================================================//

class Worker{
  public void doIt(){
    System.out.println(10/3);
  }//end doIt()
}//end class definition
//=========================================================//
```

3.14

3.2.4.7 Question 17

Write the program described in Listing 15 (p. 28) .

Listing 15: Listing for Question 17.

```
/*File Ap0010g1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code shown below, modify the
code in the method named doIt so that the program
displays

2048 instead of 2730

Did you notice anything particularly interesting about the
values involved?
**************************************************************/
public class Ap0010g1{
  public static void main(String args[]){
    new Worker().doIt(16384);
  }//end main()
}//end class definition
//============================================================//

class Worker{
  public void doIt(int val){
    System.out.println(val/6);
  }//end doIt()
}//end class definition
//============================================================//
```

3.15

3.2.4.8 Question 18

Write the program described in Listing 16 (p. 29) .

Listing 16: Listing for Question 18.

```
/*File Ap0010h1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce the output shown
below:

false
**********************************************************/

public class Ap0010h1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=========================================================//

class Worker{
  public void doIt(){
    boolean var;
    System.out.println(var);
  }//end doIt()
}//end class definition
//=========================================================//
```

3.16

3.2.5 Listings

- Listing 14 (p. 27) . Listing for Question 16.
- Listing 15 (p. 28) . Listing for Question 17.
- Listing 16 (p. 29) . Listing for Question 18.
- Listing 17 (p. 36) . Listing for Answer 1.

3.2.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0010: Self-assessment, Primitive types
- File: Ap0010.htm
- Originally published: December 17, 2001
- Published at cnx.org: 12/01/12
- Revised: December 10, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.2.7 Answers

3.2.7.1 Answer 10 .

D. 0.3333333333333333

3.2.7.1.1 Explanation 10

Divide floating type by integer type

This program divides the literal floating value of 1.0 by the literal integer value of 3 *(no decimal point is specified in the integer literal value)* .

Automatic conversion from narrow to wider type

To begin with, whenever division is performed between a floating type and an integer type, the integer type is automatically converted *(sometimes called promoted)* to a floating type and floating arithmetic is performed.

What is the actual floating type, float or double?

The real question here is, what is the type of the literal shown by 1.0 *(with a decimal point separating the 1 and the 0)* . Is it a **double** or a **float** ?

Type double is the default

By default, a literal floating value is treated as a **double** .

The result is type double

Consequently, this program divides a **double** type by an integer type, producing a result of type **double** . This is somewhat evident in the output, which shows about 17 digits plus a decimal point in the result. *(Recall that the maximum value for a float shown earlier had only about eight digits plus the decimal point and the exponent.)*

How can you create literals of type float?

What if you don't want your literal floating value to be treated as a **double** , but would prefer that it be treated as a **float** instead.

You can usually force this to be the case by adding a suffix of either F or f to the end of the literal value *(as in 1.0F)* . If you were to modify this program to cause it to divide 1.0F by 3, the output would be 0.33333334 with only nine digits in the result.

Back to Question 10 (p. 19)

3.2.7.2 Answer 9 .

D. 6.0

3.2.7.2.1 Explanation 9

Declare and initialize two local variables

This program declares and initializes two local variables, one of type **double** and the other of type **int** . Each variable is initialized with the integer value 3.

Automatic conversion to floating type double

However, before the value of 3 is stored in the **double** variable, it is automatically converted to type **double** .

Automatic conversion in mixed-type arithmetic

Numeric addition is performed on the two variables. Whenever addition is performed between a floating type and an integer type, the integer type is automatically converted to a floating type and floating arithmetic is performed.

A floating result

This produces a floating result. When this floating result is passed to the **println** method for display, a decimal point and a zero are displayed to indicate a floating type, even though in this case, the fractional part of the result is zero.

Back to Question 9 (p. 18)

3.2.7.3 Answer 8 .

A. Compiler Error

3.2.7.3.1 Explanation 8

Initialize boolean variable to true

This program declares and initializes a **boolean** variable with the value *true* . Then it attempts to add the literal value 1 to the value stored in the **boolean** variable named **myVar** .

Arithmetic with boolean values is not allowed

As mentioned earlier, unlike C++, **boolean** types in Java cannot participate in arithmetic expressions.

Therefore, this program will not compile. The compiler error produced by this program under JDK 1.3 reads partially as follows:

NOTE:

```
Ap007.java:13: operator + cannot be applied to int,boolean
System.out.println(1 + myVar);
```

Back to Question 8 (p. 17)

3.2.7.4 Answer 7 .

D. false

3.2.7.4.1 Explanation 7

Format for variable initialization

This program declares a local **boolean** variable and initializes it to the value *true* . All variables, local or otherwise, can be initialized in this manner provided that the expression on the right of the equal sign evaluates to a value that is assignment compatible with the type of the variable. *(I will have more to say about assignment compatibility in a future module)* .

Value is changed before display

However, before calling the **println** method to display the initial value of the variable, the program uses the assignment operator (=) to assign the value *false* to the variable. Thus, when it is displayed, the value is *false* .

Back to Question 7 (p. 16)

3.2.7.5 Answer 6 .

A. Compiler Error

3.2.7.5.1 Explanation 6

A local boolean variable

In this program, the primitive variable named **myVar** is a local variable belonging to the method named **printBoolean** .

Local variables are not automatically initialized

Unlike instance variables, if you fail to initialize a local variable, the variable is not automatically initialized.

Cannot access value from uninitialized local variable

If you attempt to access and use the value from an uninitialized local variable before you assign a value to it, you will get a compiler error. The compiler error produced by this program under JDK 1.3 reads partially as follows:

NOTE:

```
Ap005.java:13: variable myVar might not have been initialized
System.out.println(myVar);
```

Must initialize or assign value to all local variables

Thus, the programmer is responsible for either initializing all local variables, or assigning a value to them before attempting to access their value with code later in the program. The good news is that the system won't allow you to compute with garbage left over in memory occupied by variables, either local variables or member variables.

Back to Question 6 (p. 15)

3.2.7.6 Answer 5 .

B. false

3.2.7.6.1 Explanation 5

The boolean type

In this program, the primitive variable named **myVar** is an instance variable of the type **boolean** .

What is an instance variable?

An instance variable is a variable that is declared inside a class, outside of all methods and constructors of the class, and is not declared static. Every object instantiated from the class has one. That is why it is called an instance variable.

Cannot use uninitialized variables in Java

One of the great things about Java is that it is not possible to make the mistake of using variables that have not been initialized.

Can initialize when declared

All Java variables can be initialized when they are declared.

Member variables are automatically initialized

If the programmer doesn't initialize the variables declared inside the class but outside of a method *(often referred to as member variables as opposed to local variables)* , they are automatically initialized to a default value. The default value for a **boolean** variable is false.

Did you know the boolean default value?

I wouldn't be overly concerned if you had selected the answer A. true, because I wouldn't necessarily expect you to memorize the default initialization value.

Great cause for concern

However, I would be very concerned if you selected either C. 1 or D. 0.

Java has a true boolean type

Unlike C++, Java does not represent true and false by the numeric values of 1 and 0. *(At least the numeric values that represent true and false are not readily accessible by the programmer.)*

Thus, you cannot include boolean types in arithmetic expressions, as is the case in C++.

Back to Question 5 (p. 14)

3.2.7.7 Answer 4 .

- B. 1.7976931348623157E308
- D. 4.9E-324

3.2.7.7.1 Explanation 4

Floating type versus integer type

If you missed this one, shame on you!

I didn't expect you to memorize the maximum and minimum values represented by the floating type double, but I did expect you to be able to distinguish between the display of a floating value and the display of an integer value.

Both values are positive

Note that both of the values given above are positive values.

Unlike the integer types discussed earlier, the constants named **MAX_VALUE** and **MIN_VALUE** don't represent the ends of a signed number range for type **double** . Rather, they represent the largest and smallest *(non-zero)* values that can be expressed by the type.

An indication of granularity

MIN_VALUE is an indication of the degree of granularity of values expressed as type **double** . Any **double** value can be treated as either positive or negative.

Two floating types are available

Java provides two floating types: **float** and **double** . The **double** type provides the greater range, or to use another popular terminology, it is the *wider* of the two.

What is the value range for a float?

In case you are interested, using the same syntax as above, the value range for type float is from 1.4E-45 to 3.4028235E38

Double is often the default type

There is another thing that is significant about type **double** . In many cases where a value is automatically converted to a floating type, it is converted to type double rather than to type float. This will come up in future modules.

Back to Question 4 (p. 14)

3.2.7.8 Answer 3 .

A. -2147483648

3.2.7.8.1 Explanation 3

Could easily have guessed

As a practical matter, you had one chance in two of guessing the correct answer to this question, already having been given the value of the largest algebraic value for type int.

And the winner is ...

Did you answer B. -2147483647? – WRONG

If so, you may be wondering why the most negative value isn't equal to the negative version of the most positive value?

A twos-complement characteristic

Without going into the details of why, it is a well-known characteristic of binary twos-complement notation that the value range extends one unit further in the negative direction than in the positive direction.

What about the other two values?

Do the values of -32768 and 32767 in the set of multiple-choice answers to this question represent anything in particular?

Yes, they represent the extreme ends of the value range for a 16-bit binary number in twos-complement notation.

Does Java have a 16-bit integer type?

Just in case you are interested, the **short** type in Java is represented in 16-bit binary twos-complement signed notation, so this is the value range for type short.

What about type byte?

Similarly, a value of type **byte** is represented in 8-bit binary twos-complement signed notation, with a value range extending from -128 to 127.

Back to Question 3 (p. 13)

3.2.7.9 Answer 2 .

B. 2147483647

3.2.7.9.1 Explanation 2

First question on types

This is the first question on Java types in this group of self-assessment modules.

32-bit signed twos-complement integers

In Java, values of type **int** are stored as 32-bit signed integers in twos-complement notation.

Can you calculate the values?

There are no unsigned integer types in Java, as there are in C++. If you are handy with binary notation, you could calculate the largest positive value that can be stored in 32 bits in twos-complement notation.

See documentation for the Integer class

Otherwise, you can visit the documentation [6] for the **Integer** class, which provides a symbolic constant *(public static final variable)* named **MAX_VALUE** . The description of MAX_VALUE reads as follows:

"The largest value of type int. The constant value of this field is 2147483647."

Back to Question 2 (p. 13)

3.2.7.10 Answer 1 .

C. Hello World

3.2.7.10.1 Explanation 1

The answer to this first question is intended to be easy. The purpose of the first question is to introduce you to the syntax that will frequently be used for program code in this group of self-assessment modules.

The controlling class and the main method

In this example, the class named **Ap001** is the *controlling class* . It contains a method named **main** , with a signature that matches the required signature for the **main** method. When the user executes this program, the Java virtual machine automatically calls the method named **main** in the controlling class.

Create an instance of Worker

The **main** method uses the **new** operator along with the default constructor for the class named **Worker** to create a new instance of the class named **Worker** *(an object of the Worker class)* . This is often referred to as instantiating an object.

A reference to an anonymous object

The combination of the **new** operator and the default constructor for the **Worker** class returns a reference to the new object. In this case, the object is instantiated as an *anonymous object* , meaning that the object's reference is not saved in a named reference variable. *(Instantiation of a non-anonymous object will be illustrated later.)*

Call hello method on Worker object

The main method contains a single executable statement.

As soon as the reference to the new object is returned, the single statement in the **main** method calls the **hello** method on that reference.

Output to standard output device

This causes the **hello** method belonging to the new object *(of the class named **Worker**)* to execute. The code in the **hello** method calls the **println** method on the **static** variable of the **System** class named **out** .

Lots of OOP embodied in the hello method

I often tell my students that I can tell a lot about whether a student really understands object-oriented programming in Java by asking them to explain everything that they know about the following statement:

System.out.println("Hello World");

I would expect the answer to consume about ten to fifteen minutes if the student really understands Java OOP.

The one-minute version

When the virtual machine starts a Java application running, it automatically instantiates an I/O stream object linked to the standard output device *(normally the screen)* and stores a reference to that object in the **static** variable named **out** belonging to the class named **System** .

Call the println instance method on out

Calling the **println** method on that reference, and passing a literal string *("Hello World")* to that method causes the contents of the literal **String** object to be displayed on the standard output device.

Display Hello World on the screen

In this case, this causes the words *Hello World* to be displayed on the standard output device. This is the answer to the original question.

Time for main method to terminate

[6] http://cnx.org/content/m45117

When the **hello** method returns, the **main** method has nothing further to do, so it terminates. When the **main** method terminates in a Java application, the application terminates and returns control to the operating system. This causes the system prompt to reappear.

A less-cryptic form

A less cryptic form of this program is shown in Listing 17 (p. 36) .

Listing 17: Listing for Answer 1.

```
    public class Ap002{
  public static void main(
                       String args[]){
    Worker refVar = new Worker();
    refVar.hello();
  }//end main()
}//end class definition
class Worker{

  public void hello(){
    System.out.println("Hello World");
  }//end hello()
}//end class definition
```

3.17

Decompose single statement into two statements

In this version, the single statement in the earlier version of the **main** method is replaced by two statements.

A non-anonymous object

In the class named **Ap002** shown in Listing 2 (p. 14) , the object of the class named **Worker** is not instantiated anonymously. Rather, a new object of the **Worker** class is instantiated and the object's reference is stored in *(assigned to)* the named reference variable named **refVar** .

Call hello method on named reference

Then the **hello** method is called on that reference in a separate statement.

Produces the same result as before

The final result is exactly the same as before. The only difference is that a little more typing is required to create the source code for the second version.

Will often use anonymous objects

In order to minimize the amount of typing required, I will probably use the anonymous form of instantiation whenever appropriate in these modules.

Now that you understand the framework ...

Now that you understand the framework for the program code, I can present more specific questions. Also, the explanations will usually be shorter.

Back to Question 1 (p. 13)

-end-

3.3 Ap0020: Self-assessment, Assignment and Arithmetic Operators[7]

3.3.1 Table of Contents

3.3.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

Questions and answers

The test consists of a series of questions (p. 37) with answers (p. 60) and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

Programming challenge questions

The module also contains a section titled Programming challenge questions (p. 52) . This section provides specifications for one or more programs that you should be able to write once you understand the answers to all of the questions. *(Note that it is not always possible to confine the programming knowledge requirement to this and earlier modules. Therefore, you may occasionally need to refer ahead to future modules in order to write the programs.)*

Unlike the other questions, solutions are not provided for the *Programming challenge questions* . However, in most cases, the specifications will describe the output that your program should produce.

Listings

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 59) to easily find and view the listings while you are reading about them.

3.3.3 Questions

3.3.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 38) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.0
- D. 4.0
- E. 7.0

[7] This content is available online at <http://cnx.org/content/m45286/1.3/>.

Listing 1: Listing for Question 1.

```
    public class Ap010{
  public static void main(
                        String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    double myVar;
    myVar = 3.0;
    myVar += 4.0;
    System.out.println(myVar);
  }//end doAsg()
}//end class definition
```

3.18

Answer and Explanation (p. 68)

3.3.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 39) ?

- A. Compiler Error
- B. Runtime Error
- C. 2.147483647E9
- D. 2.14748365E9

Listing 2: Listing for Question 2.

```
    public class Ap011{
  public static void main(
                       String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    double myDoubleVar;
    //Integer.MAX_VALUE = 2147483647
    int myIntVar = Integer.MAX_VALUE;
    myDoubleVar = myIntVar;
    System.out.println(myDoubleVar);
  }//end doAsg()
}//end class definition
```

3.19

Answer and Explanation (p. 67)

3.3.3.3 Question 3

What output is produced by the following program?

- A. Compiler Error
- B. Runtime Error
- C. 2147483647
- D. 2.147483647E9

Listing 3: Listing for Question 3.

```
    public class Ap012{
  public static void main(
                       String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    //Integer.MAX_VALUE = 2147483647
    double myDoubleVar =
                    Integer.MAX_VALUE;
    int myIntVar;
    myIntVar = myDoubleVar;
    System.out.println(myIntVar);
  }//end doAsg()
}//end class definition
```

3.20

Answer and Explanation (p. 66)

3.3.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 41) ?

- A. Compiler Error
- B. Runtime Error
- C. 2147483647
- D. 2.147483647E9

Listing 4: Listing for Question 4.

```
    public class Ap013{
  public static void main(
                       String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    //Integer.MAX_VALUE = 2147483647
    double myDoubleVar =
                    Integer.MAX_VALUE;
    int myIntVar;
    myIntVar = (int)myDoubleVar;
    System.out.println(myIntVar);
  }//end doAsg()
}//end class definition
```

3.21

Answer and Explanation (p. 66)

3.3.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 42) ?

- A. Compiler Error
- B. Runtime Error
- C. 4.294967294E9
- D. 4294967294

Listing 5: Listing for Question 5.

```
    public class Ap014{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    //Integer.MAX_VALUE = 2147483647
    int myIntVar = Integer.MAX_VALUE;
    System.out.println(2.0 * myIntVar);
  }//end doMixed()
}//end class definition
```

3.22

Answer and Explanation (p. 65)

3.3.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 43) ?

- A. Compiler Error
- B. Runtime Error
- C. 2147483649
- D. -2147483647

Listing 6: Listing for Question 6.

```
    public class Ap015{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    //Integer.MAX_VALUE = 2147483647
    int myVar01 = Integer.MAX_VALUE;
    int myVar02 = 2;
    System.out.println(
                     myVar01 + myVar02);
  }//end doMixed()
}//end class definition
```

3.23

Answer and Explanation (p. 64)

3.3.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 44) ?

- A. Compiler Error
- B. Runtime Error
- C. 33.666666
- D. 34
- E. 33

Listing 7: Listing for Question 7.

```
    public class Ap016{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    int myVar01 = 101;
    int myVar02 = 3;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

3.24

Answer and Explanation (p. 64)

3.3.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 45) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. 11

Listing 8: Listing for Question 8.

```
    public class Ap017{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    int myVar01 = 11;
    int myVar02 = 0;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

3.25

Answer and Explanation (p. 63)

3.3.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 46) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. 11

Listing 9: Listing for Question 9.

```
    public class Ap018{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    double myVar01 = 11;
    double myVar02 = 0;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

3.26

Answer and Explanation (p. 63)

3.3.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 47) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. -2

Listing 10: Listing for Question 10.

```
    public class Ap019{
  public static void main(
                       String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int myVar01 = -11;
    int myVar02 = 3;
    System.out.println(
                     myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

3.27

Answer and Explanation (p. 62)

3.3.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 48) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 11

Listing 11: Listing for Question 11.

```
    public class Ap020{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int myVar01 = -11;
    int myVar02 = 0;
    System.out.println(
                      myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

3.28

Answer and Explanation (p. 62)

3.3.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 49) ?

- A. Compiler Error
- B. Runtime Error
- C. -0.010999999999999996
- D. 0.010999999999999996

Listing 12: Listing for Question 12.

```
    public class Ap021{
  public static void main(
                       String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = -0.11;
    double myVar02 = 0.033;
    System.out.println(
                     myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

3.29

Answer and Explanation (p. 61)

3.3.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 50) ?

- A. Compiler Error
- B. Runtime Error
- C. 0.0
- D. 1.5499999999999996

Listing 13: Listing for Question 13.

```
    public class Ap022{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = 15.5;
    double myVar02 = 1.55;
    System.out.println(
                      myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

3.30

Answer and Explanation (p. 61)

3.3.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 51) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. NaN

Listing 14: Listing for Question 14.

```
    public class Ap023{
  public static void main(
                       String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = 15.5;
    double myVar02 = 0.0;
    System.out.println(
                     myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

3.31

Answer and Explanation (p. 61)

3.3.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 52) ?

- A. Compiler Error
- B. Runtime Error
- C. -3 2
- D. -3 -2

Listing 15: Listing for Question 15.

```
    public class Ap024{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int x = 11;
    int y = -3;
    System.out.println(
                      x/y + " " + x % y);
  }//end doMod()
}//end class definition
```

3.32

Answer and Explanation (p. 60)

3.3.4 Programming challenge questions

3.3.4.1 Question 16

Write the program described in Listing 16 (p. 53) .

Listing 16: Listing for Question 16.

```
/*File Ap0020a1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the proper use of the combined
arithmetic/assignment operators such as the following
operators:

+=
*=

**********************************************************/
public class Ap0020a1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

3.33

3.3.4.2 Question 17

Write the program described in Listing 17 (p. 54) .

Listing 17: Listing for Question 17.

```
/*File Ap0020b1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the detrimental impact of integer arithmetic
overflow.

**********************************************************/
public class Ap0020b1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

3.34

3.3.4.3 Question 18

Write the program described in Listing 18 (p. 55) .

Listing 18: Listing for Question 18.

```
/*File Ap0020c1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of integer truncation that
occurs with integer division.

**********************************************************/
public class Ap0020c1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

3.35

3.3.4.4 Question 19

Write the program described in Listing 19 (p. 56) .

Listing 19: Listing for Question 19.

```
/*File Ap0020d1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of double divide by zero.
2. Illustrates the effect of integer divide by zero.

**********************************************************/
public class Ap0020d1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

3.36

3.3.4.5 Question 20

Write the program described in Listing 20 (p. 57) .

Listing 20: Listing for Question 20.

```
/*File Ap0020e1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of the modulus operation with
integers.

**********************************************************/
public class Ap0020e1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------//
}//end class definition
//========================================================//
```

3.37

3.3.4.6 Question 21

Write the program described in Listing 21 (p. 58) .

Listing 21: Listing for Question 21.

```
/*File Ap0020f1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of the modulus operation with
doubles.

**********************************************************/
public class Ap0020f1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------//
}//end class definition
//========================================================//
```

3.38

3.3.4.7 Question 22

Write the program described in Listing 22 (p. 59) .

Listing 22: Listing for Question 22.

```
/*File Ap0020g1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the concatenation of the following strings
separated by space characters.

"This"
"is"
"fun"

Cause your program to produce the following output:
This
is
fun
This is fun
**********************************************************/
public class Ap0020g1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //------------------------------------------------------//
}//end class definition
//========================================================//
```

3.39

3.3.5 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.3.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0020: Self-assessment, Assignment and Arithmetic Operators
- File: Ap0020.htm
- Originally published: January 7, 2002
- Published at cnx.org: 12/01/12
- Revised: December 11, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.3.7 Answers

3.3.7.1 Answer 15

C. -3 2

3.3.7.1.1 Explanation 15

String concatenation

This program uses **String** concatenation, which has not been previously discussed in this group of self-assessment modules.

In this case, the program executes both an integer divide operation and an integer modulus operation, using **String** concatenation to display both results on a single line of output.

Quotient = -3 with a remainder of 2

Thus, the displayed result is the integer quotient followed by the remainder.

What is String concatenation?

If either operand of the plus (+) operator is of type **String** , no attempt is made to perform arithmetic addition. Rather, the other operand is converted to a **String** , and the two strings are concatenated.

A space character, " "

The string containing a space character (" ") in this expression appears as the right operand of one plus operator and as the left operand of the other plus operator.

If you already knew about **String** concatenation, you should have been able to figure out the correct answer to the question on the basis of the answers to earlier questions in this module.

Back to Question 15 (p. 51)

3.3.7.2 Answer 14

D. NaN

3.3.7.2.1 Explanation 14

Floating modulus operation involves floating divide

The modulus operation with floating operands and 0.0 as the right operand produces **NaN** , which stands for *Not a Number* .

What is the actual value of Not a Number?

A symbolic constant that is accessible as **Double.NaN** specifies the value that is returned in this case.

Be careful what you try to do with it. It has some peculiar behavior of its own.

Back to Question 14 (p. 50)

3.3.7.3 Answer 13

D. 1.5499999999999996

3.3.7.3.1 Explanation 13

A totally incorrect result

Unfortunately, due to floating arithmetic inaccuracy, the modulus operation in this program produces an entirely incorrect result.

The result should be 0.0, and that is the result produced by my hand calculator.

Terminates one step too early

However, this program terminates the repetitive subtraction process one step too early and produces an incorrect remainder.

Be careful

This program is included here to emphasize the need to be very careful how you interpret the result of performing modulus operations on floating operands.

Back to Question 13 (p. 49)

3.3.7.4 Answer 12

C. -0.010999999999999996

3.3.7.4.1 Explanation 12

Modulus operator can be used with floating types

In this case, the program returns the remainder that would be produced by dividing a double value of -0.11 by a double value of 0.033 and terminating the divide operation at the beginning of the fractional part of the quotient.

Say that again

Stated differently, the result of the modulus operation is the remainder that results after

- subtracting the right operand from the left operand an integral number of times, and
- terminating the repetitive subtraction process when the result of the subtraction is less than the right operand

Modulus result is not exact

According to my hand calculator, taking into account the fact that the left operand is negative, this operation should produce a modulus result of -0.011. As you can see, the result produced by the application of the modulus operation to floating types is not exact.

Back to Question 12 (p. 48)

3.3.7.5 Answer 11

B. Runtime Error

3.3.7.5.1 Explanation 11

Integer modulus involves integer divide

The modulus operation with integer operands involves an integer divide.

Therefore, it is subject to the same kind of problem as an ordinary integer divide when the right operand has a value of zero.

Program produces a runtime error

In this case, the program produced a runtime error that terminated the program. The error produced by JDK 1.3 is as follows:

NOTE:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Worker.doMod(Ap020.java:14)
    at Ap020.main(Ap020.java:6)
```

Dealing with the problem

As with integer divide, you can either test the right operand for a zero value before performing the modulus operation, or you can deal with the problem after the fact using try-catch.

Back to Question 11 (p. 47)

3.3.7.6 Answer 10

D. -2

3.3.7.6.1 Explanation 10

What is a modulus operation?

In elementary terms, we like to say that the modulus operation returns the remainder that results from a divide operation.

In general terms, that is true.

Some interesting behavior

However, the modulus operation has some interesting behaviors that are illustrated in this and the next several questions.

This program returns the modulus of -11 and 3, with -11 being the left operand.

What is the algebraic sign of the result?

Here is a rule:

The result of the modulus operation takes the sign of the left operand, regardless of the sign of the quotient and regardless of the sign of the right operand. In this program, that produced a result of -2.

Changing the sign of the right operand would ***not*** have changed the sign of the result.

Exercise care with sign of modulus result

Thus, you may need to exercise care as to how you interpret the result when you perform a modulus operation having a negative left operand.

Back to Question 10 (p. 46)

3.3.7.7 Answer 9

C. Infinity

3.3.7.7.1 Explanation 9

Floating divide by zero

This program attempts to divide the **double** value of 11 by the **double** value of zero.

No runtime error with floating divide by zero

In the case of floating types, an attempt to divide by zero does not produce a runtime error. Rather, it returns a value that the **println** method interprets and displays as Infinity.

What is the actual value?

The actual value returned by this program is provided by a **static final** variable in the **Double** class named **POSITIVE_INFINITY** .

(There is also a value for NEGATIVE_INFINITY, which is the value that would be returned if one of the operands were a negative value.)

Is this a better approach?

Is this a better approach than throwing an exception as is the case for integer divide by zero?

I will let you be the judge of that.

In either case, you can test the right operand before the divide to assure that it isn't equal to zero.

Cannot use exception handling in this case

For floating divide by zero, you cannot handle the problem by using try-catch.

However, you can test the result following the divide to see if it is equal to either of the infinity values mentioned above.

Back to Question 9 (p. 45)

3.3.7.8 Answer 8

B. Runtime Error

3.3.7.8.1 Explanation 8

Dividing by zero

This program attempts to divide the **int** value of 11 by the **int** value of zero.

Integer divide by zero is not allowed

This produces a runtime error and terminates the program.

The runtime error is as follows under JDK 1.3:

NOTE:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at Worker.doMixed(Ap017.java:14)
     at Ap017.main(Ap017.java:6)
```

Two ways to deal with this sort of problem

One way is to test the right operand before each divide operation to assure that it isn't equal to zero, and to take appropriate action if it is.

A second *(possibly preferred)* way is to use exception handling and surround the divide operation with a *try* block, followed by a *catch* block for the type

NOTE:

```
java.lang.ArithmeticException.
```

The code in the catch block can be designed to deal with the problem if it occurs. *(Exception handling will be discussed in a future module.)*

Back to Question 8 (p. 44)

3.3.7.9 Answer 7

E. 33

3.3.7.9.1 Explanation 7

Integer truncation

This program illustrates the integer truncation that results when the division operator is applied to operands of the integer types.

The result of simple long division

We all know that when we divide 101 by 3, the result is 33.666666 with the sixes extending out to the limit of our arithmetic accuracy.

The result of rounding

If we round the result to the next closest integer, the result is 34.

Integer division does not round

However, when division is performed using operands of integer types in Java, the fractional part is simply discarded *(not rounded)* .

The result is the whole number result without regard for the fractional part or the remainder.

Thus, with integer division, 101/3 produces the integer value 33.

If either operand is a floating type ...

If either operand is one of the floating types,

- the integer operand will be converted to the floating type,
- the result will be of the floating type, and
- the fractional part of the result will be preserved to some degree of accuracy

Back to Question 7 (p. 43)

3.3.7.10 Answer 6

D. -2147483647

3.3.7.10.1 Explanation 6

Danger, integer overflow ahead!

This program illustrates a very dangerous situation involving arithmetic using operands of integer types. This situation involves a condition commonly known as *integer overflow* .

The good news

The good news about doing arithmetic using operands of integer types is that as long as the result is within the allowable value range for the wider of the integer types, the results are exact *(floating arithmetic often produces results that are not exact)* .

The bad news

The bad news about doing arithmetic using operands of integer types is that when the result is not within the allowable value range for the wider of the integer types, the results are garbage, having no usable relationship to the correct result *(floating arithmetic has a high probability of producing approximately correct results, even though the results may not be exact).*

For this specific case ...

As you can see by the answer to this question, when a value of 2 was added to the largest positive value that can be stored in type **int** , the incorrect result was a very large negative value.

The result is simply incorrect. *(If you know how to do binary arithmetic, you can figure out how this happens.)*

No safety net in this case – just garbage

Furthermore, there was no compiler error and no runtime error. The program simply produced an incorrect result with no warning.

You need to be especially careful when writing programs that perform arithmetic using operands of integer types. Otherwise, your programs may produce incorrect results.

Back to Question 6 (p. 42)

3.3.7.11 Answer 5

C. 4.294967294E9

3.3.7.11.1 Explanation 5

Mixed-type arithmetic

This program illustrates the use of arithmetic operators with operands of different types.

Declare and initialize an int

The method named **doMixed** declares a local variable of type int named **myIntVar** and initializes it with the largest positive value that can be stored in type **int** .

Evaluate an arithmetic expression

An arithmetic expression involving **myIntVar** is evaluated and the result is passed as a parameter to the **println** method where it is displayed on the computer screen.

Multiply by a literal double value

The arithmetic expression uses the multiplication operator (*) to multiply the value stored in **myIntVar** by 2.0 *(this literal operand is type **double** by default)* .

Automatic conversion to wider type

When arithmetic is performed using operands of different types, the type of the operand of the narrower type is automatically converted to the type of the operand of the wider type, and the arithmetic is performed on the basis of the wider type.

Result is of the wider type

The type of the result is the same as the wider type.

In this case ...

Because the left operand is type **double** , the **int** value is converted to type **double** and the arithmetic is performed as type **double** .

This produces a result of type **double** , causing the floating value 4.294967294E9 to be displayed on the computer screen.

Back to Question 5 (p. 41)

3.3.7.12 Answer 4

C. 2147483647

3.3.7.12.1 Explanation 4

Uses a cast operator

This program, named **Ap013.java** , differs from the earlier program named **Ap012.java** in one important respect.

This program uses a *cast operator* to force the compiler to allow a narrowing conversion in order to assign a **double** value to an **int** variable.

The cast operator

The statement containing the cast operator is shown below for convenient viewing.

NOTE:

```
myIntVar = (int)myDoubleVar;
```

Syntax of a cast operator

The cast operator consists of the name of a type contained within a pair of matching parentheses.

A unary operator

The cast operator always appears to the left of an expression whose type is being converted to the type specified by the cast operator.

Assuming responsibility for potential problems

When dealing with primitive types, the cast operator is used to notify the compiler that the programmer is willing to assume the risk of a possible loss of precision in a narrowing conversion.

No loss of precision here

In this case, there was no loss in precision, but that was only because the value stored in the **double** variable was within the allowable value range for an **int** .

In fact, it was the largest positive value that can be stored in the type **int** . Had it been any larger, a loss of precision would have occurred.

More on this later ...

I will have quite a bit more to say about the cast operator in future modules. I will also have more to say about the use of the assignment operator in conjunction with the non-primitive types.

Back to Question 4 (p. 40)

3.3.7.13 Answer 3

A. Compiler Error

3.3.7.13.1 Explanation 3

Conversion from double to int is not automatic

This program attempts to assign a value of type **double** to a variable of type **int** .

Even though we know that the specific double value involved would fit in the **int** variable with no loss of precision, the conversion from **double** to **int** is not a *widening* conversion.

This is a narrowing conversion

In fact, it is a *narrowing* conversion because the allowable value range for an **int** is less than the allowable value range for a **double** .

The conversion is not allowed by the compiler. The following compiler error occurs under JDK 1.3:

NOTE:

```
    Ap012.java:16: possible loss of precision
found   : double
required: int
    myIntVar = myDoubleVar    myIntVar = myDoubleVar;
```

Back to Question 3 (p. 39)

3.3.7.14 Answer 2

C. 2.147483647E9

3.3.7.14.1 Explanation 2

Declare a double

The method named **doAsg** first declares a local variable of type **double** named **myDoubleVar** without providing an initial value.

Declare and initialize an int

Then it declares an **int** variable named **myIntVar** and initializes its value to the integer value 2147483647 *(you learned about Integer.MAX_ VALUE in an earlier module)* .

Assign the int to the double

Following this, the method assigns contents of the **int** variable to the **double** variable.

An assignment compatible conversion

This is an *assignment compatible* conversion. In particular, the integer value of 2147483647 is automatically converted to a **double** value and stored in the **double** variable.

The **double** representation of that value is what appears on the screen later when the value of **myDoubleVar** is displayed.

What is an assignment compatible conversion?

An assignment compatible conversion for the primitive types occurs when the required conversion is a *widening* conversion.

What is a widening conversion?

A widening conversion occurs when the allowable value range of the type of the left operand of the assignment operator is greater than the allowable value range of the right operand of the assignment operator.

A double is wider than an int

Since the allowable value range of type **double** is greater than the allowable value range of type **int** , assignment of an **int** value to a **double** variable is allowed, with conversion from **int** to **double** occurring automatically.

A safe conversion

It is also significant to note that there is no loss in precision when converting from an **int** to a **double** .

An unsafe but allowable conversion

However, a loss of precision may occur when an **int** is assigned to a **float** , or when a **long** is assigned to a **double** .

What would a float produce ?

The value of 2.14748365E9 shown for selection D is what you would see for this program if you were to change the **double** variable to a **float** variable. *(Contrast this with 2147483647 to see the loss of precision.)*

Widening is no guarantee that precision will be preserved

The fact that a type conversion is a widening conversion does not guarantee that there will be no loss of precision in the conversion. It simply guarantees that the conversion will be allowed by the compiler. In

some cases, such as that shown above (p. 67) , an assignment compatible conversion can result in a loss of precision, so you always need to be aware of what you are doing.

Back to Question 2 (p. 38)

3.3.7.15 Answer 1

E. 7.0

3.3.7.15.1 Explanation 1

Declare but don't initialize a double variable

The method named **doAsg** begins by declaring a **double** variable named **myVar** without initializing it.

Use the simple assignment operator

The simple assignment operator (=) is then used to assign the **double** value 3.0 to the variable. Following the execution of that statement, the variable contains the value 3.0.

Use the arithmetic/assignment operator

The next statement uses the combined arithmetic/assignment operator (+=) to add the value 4.0 to the value of 3.0 previously assigned to the variable. The following two statements are functionally equivalent:

NOTE:

```
myVar += 4.0;
```

```
myVar = myVar + 4.0;
```

Two statements are equivalent

This program uses the first statement listed above. If you were to replace the first statement with the second statement, the result would be the same.

In this case, either statement would add the value 4.0 to the value of 3.0 that was previously assigned to the variable named **myVar** , producing the sum of 7.0. Then it would assign the sum of 7.0 back to the variable. When the contents of the variable are then displayed, the result is that 7.0 appears on the computer screen.

No particular benefit

To the knowledge of this author, there is no particular benefit to using the combined arithmetic/assignment notation other than to reduce the amount of typing required to produce the source code. However, if you ever plan to interview for a job as a Java programmer, you need to know how to use the combined version.

Four other similar operators

Java support several combined operators. Some involve arithmetic and some involve other operations such as bit shifting. Five of the combined operators are shown below. These five all involve arithmetic.

- +=
- -=
- *=
- /=
- %=

In all five cases, you can construct a functionally equivalent arithmetic and assignment statement in the same way that I constructed the functionally equivalent statement for += above.

Back to Question 1 (p. 37)

-end-

3.4 Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures[8]

3.4.1 Table of Contents

3.4.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 84) to easily find and view the listings while you are reading about them.

3.4.3 Questions

3.4.3.1 Question 1 .

Given: The use of **String** concatenation in the argument list of the call to the **println** method in the program shown in Listing 1 (p. 70) will cause seven items to be displayed on the screen, separated by spaces.

True or False? The program produces the output shown below:

NOTE:

```
false true false false true true false
```

[8] This content is available online at <http://cnx.org/content/m45287/1.2/>.

Listing 1: Listing for Question 1.

```
    public class Ap025{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    int a = 1, b = 2, c = 3, d = 2;

    System.out.println(
                      (a == b) + " " +
                      (b == d) + " " +
                      (b != d) + " " +
                      (c < a) + " " +
                      (b <= d) + " " +
                      (c > d) + " " +
                      (a >= c));
  }//end doRelat()
}//end class definition
```

3.40

Answer and Explanation (p. 93)

3.4.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 71) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 2: Listing for Question 2.

```
    public class Ap026{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = new Dummy();
    System.out.println(x == y);
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

3.41

Answer and Explanation (p. 92)

3.4.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 72) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 3: Listing for Question 3.

```
    public class Ap027{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = new Dummy();
    System.out.println(x.equals(y));
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

3.42

Answer and Explanation (p. 91)

3.4.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 73) ?

- A. Compiler Error
- B. Runtime Error
- C. true false
- D. false true
- E. true true

Listing 4: Listing for Question 4.

```
    public class Ap028{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = x;
    System.out.println(
         (x == y) + " " + x.equals(y));
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

3.43

Answer and Explanation (p. 91)

3.4.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 74) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 5: Listing for Question 5.

```
    public class Ap029{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = new Dummy();
    System.out.println(x > y);
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

3.44

Answer and Explanation (p. 91)

3.4.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 75) ?

- A. Compiler Error
- B. Runtime Error
- C. 5 5 8.3 8.3
- D. 6 4 9.3 7.300000000000001

Listing 6: Listing for Question 6.

```
    public class Ap030{
  public static void main(
                       String args[]){
    new Worker().doIncr();
  }//end main()
}//end class definition

class Worker{
  public void doIncr(){
    int w = 5, x = 5;
    double y = 8.3, z = 8.3;

    w++;
    x--;
    y++;
    z--;

    System.out.println(w + "  " +
                       x + "  " +
                       y + "  " +
                       z);
  }//end doIncr()
}//end class definition
```

3.45

Answer and Explanation (p. 90)

3.4.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 76) ?

- A. Compiler Error
- B. Runtime Error
- C. Hello
- D. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap031{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x - y){
      System.out.println("Hello");
    }//end if
  }//end doIf()
}//end class definition
```

3.46

Answer and Explanation (p. 90)

3.4.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 77) ?

- A. Compiler Error
- B. Runtime Error
- C. World
- D. Hello World
- E. None of the above

Listing 8: Listing for Question 8.

```
    public class Ap032{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x < y){
      System.out.print("Hello ");
    }//end if
    System.out.println("World");
  }//end doIf()
}//end class definition
```

3.47

Answer and Explanation (p. 89)

3.4.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 78) ?

- A. Compiler Error
- B. Runtime Error
- C. Hello World
- D. Goodbye World
- E. None of the above

Listing 9: Listing for Question 9.

```
    public class Ap033{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x == y){
      System.out.println(
                         "Hello World");
    }else{
      System.out.println(
                       "Goodbye World");
    }//end else
  }//end doIf()
}//end class definition
```

3.48

Answer and Explanation (p. 88)

3.4.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 79) ?

- A. Compiler Error
- B. Runtime Error
- C. x = 4
- D. x = 5
- E. x = 6
- F. x != 4,5,6
- G. None of the above

Listing 10: Listing for Question 10.

```
    public class Ap034{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 2;
    if(x == 4){
      System.out.println("x = 4");
    }else if (x == 5){
      System.out.println("x = 5");
    }else if (x == 6){
      System.out.println("x = 6");
    }else{
      System.out.println("x != 4,5,6");
    }//end else
  }//end doIf()
}//end class definition
```

3.49

Answer and Explanation (p. 88)

3.4.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 80) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4
- D. 1 2 3 4 5
- E. None of the above

Listing 11: Listing for Question 11.

```
    public class Ap035{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    int cnt = 0;
    while(cnt<5){
      cnt++;
      System.out.print(cnt + " ");
      cnt++;
    }//end while loop
    System.out.println("");
  }//end doLoop()
}//end class definition
```

3.50

Answer and Explanation (p. 88)

3.4.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 81) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4 5
- D. 1 2 3 4 5 5
- E. None of the above

Listing 12: Listing for Question 12.

```
    public class Ap036{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    int cnt;
    for(cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
    }//end for loop
    System.out.println(cnt + " ");
  }//end doLoop()
}//end class definition
```

3.51

Answer and Explanation (p. 87)

3.4.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 82) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4 5
- D. 1 2 3 4 5 5
- E. None of the above

Listing 13: Listing for Question 13.

```
    public class Ap037{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
    }//end for loop
    System.out.println(cnt + " ");
  }//end doLoop()
}//end class definition
```

3.52

Answer and Explanation (p. 86)

3.4.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 83) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 3
- D. 0 1 2 3 4
- E. None of the above

Listing 14: Listing for Question 14.

```
    public class Ap037{
  public static void main(
                       String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public double doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
      if(cnt == 3){
        System.out.println(cnt);
        return cnt;
      }//end if
    }//end for loop
    //return 3.5;
  }//end doLoop()
}//end class definition
```

3.53

Answer and Explanation (p. 86)

3.4.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 84) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 3
- D. 0 1 2 3 4
- E. None of the above

Listing 15: Listing for Question 15.

```
    public class Ap038{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
      if(cnt == 3){
        System.out.println(cnt);
        return;
      }//end if
    }//end for loop
  }//end doLoop()
}//end class definition
```

3.54

Answer and Explanation (p. 85)

3.4.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.4.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures
- File: Ap0030.htm
- Originally published: January, 2002
- Published at cnx.org: 12/02/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.4.6 Answers

3.4.6.1 Answer 15

C. 0 1 2 3 3

3.4.6.1.1 Explanation 15

No return statement is required

A method with a signature that specifies a **void** return type does not require a **return** statement.

However, such a method may contain a **return** statement, provided that it is terminated immediately with a semicolon *(no expression between the word return and the semicolon)* .

(Every method whose return type is not void must contain at least one return statement.)

Multiple return statements are allowed

Any method may contain any number of **return** statements provided that they make sense from a syntax viewpoint, and provided the expression *(or lack thereof)* between the word **return** and the semicolon evaluates to the type specified in the method signature *(or a type that will be automatically converted to the type specified in the method signature)* .

A return statement terminates a method immediately

Whenever the execution stream encounters any **return** statement, the method is terminated immediately, and control is returned to the method that called that method.

Back to Question 15 (p. 83)

3.4.6.2 Answer 14

A. Compiler Error

3.4.6.2.1 Explanation 14

Missing return statement

This program produces the following compiler error under JDK 1.3:

NOTE:

```
    Ap037.java:11: missing return statement
  public int doLoop(){
```

Even though this program contains a **return** statement inside the **for** loop, it is still necessary to place a **return** statement at the end of the method to satisfy the compiler. *(The one shown in the code is a comment.)*

The method named **doLoop** must return a value of type **double** . Apparently the compiler assumes that the **return** statement inside the **for** loop may never be executed *(although that isn't true in this case)* .

Both of the **return** statements must return a value that satisfies the **double** type requirement given in the method signature.

Returning a value of type **int** in the **for** loop will satisfy the type requirement because type **int** will be automatically converted to type **double** as it is returned. *(Conversion from type **int** to type **double** is a widening conversion.)*

Back to Question 14 (p. 82)

3.4.6.3 Answer 13

A. Compiler Error

3.4.6.3.1 Explanation 13

The scope of a local variable

In general, the scope of a local variable extends from the point at which it is declared to the curly brace that signals the end of the block in which it is declared.

This applies to for loop in an interesting way

While it is allowable to declare a variable within the first clause of a **for** loop, the scope of that variable is limited to the block of code contained in the loop structure.

The variable cannot be accessed outside the loop.

Attempts to access variable out of scope

This program attempts to access the value of the variable named **cnt** after the loop terminates.

The program displays the following compiler error under JDK 1.3. This error results from the attempt to display the value of the counter after the loop terminates.

NOTE:

```
    Ap037.java:15: cannot resolve symbol
symbol  : variable cnt
location: class Worker
    System.out.println(cnt + " ");
```

Back to Question 13 (p. 81)

3.4.6.4 Answer 12

C. 0 1 2 3 4 5

3.4.6.4.1 Explanation 12

A simple for loop structure

This program illustrates a simple **for** loop that displays the value of its counter using a call to the **print** method inside the loop.

After the loop terminates, the program displays the value of the counter one last time using a call to **println** .

Three clauses separated by semicolons

The first line of a **for** loop structure always contains three clauses separated by semicolons.

The first and third clauses may be empty, but the semicolons are required in any case.

The first clause ...

The first clause is executed once and only once at the beginning of the loop.

It can contain just about any valid Java expression.

It can even contain more than one expression with the individual expression separated by commas.

When the first clause contains more than one expression separated by commas, the expressions are evaluated in left-to-right order.

The second clause

The second clause is a conditional clause. It must contain an expression that returns a **boolean** value.

(Actually, this clause can also be empty, in which case it is apparently assumed to be true. This leads to an infinite loop unless there is some code inside the loop to terminate it, perhaps by executing a return or a break statement.)

An entry-condition loop

The **for** loop is an entry condition loop, meaning that the conditional expression is evaluated once immediately after the first clause is executed, and once per iteration thereafter.

Behavior of the for loop

If the conditional expression returns true, the block of code following the closing parenthesis is executed.

If it returns false, the block of code is skipped, and control passes to the first executable statement following the block of code.

(For the case where the block contains only one statement, the matching curly brackets can be omitted.)

The third clause

The third clause can contain none, one, or more valid expressions separated by commas.

If there are more than one, they are evaluated in left-to-right order.

When they are evaluated

The expressions in the third clause are evaluated once during each iteration.

However, it is very important to remember that despite the physical placement of the clause in the first line, the expressions in the third clause are not evaluated until after the code in the block has been evaluated.

Typically an update clause

The third clause is typically used to update a counter, but this is not a technical requirement.

This clause can be used for just about any purpose.

However, the counter must be updated somewhere within the block of code or the loop will never terminate.

(Stated differently, something must occur within the block of code to eventually cause the conditional expression to evaluate to false. Otherwise, the loop will never terminate on its own. However, it is possible to execute a ***return*** *or* ***break*** *within the block to terminate the loop.)*

Note the first output value for this program

Because the update in the third clause is not executed until after the code in the block has been executed, the first value displayed (p. 87) by this program is the value zero.

Back to Question 12 (p. 80)

3.4.6.5 Answer 11

E. None of the above

3.4.6.5.1 Explanation 11

And the answer is ...

The output produced by this program is:

1 3 5

A simple while loop

This program uses a simple **while** loop to display the value of a counter, once during each iteration.

Behavior of a while loop

As long as the relational expression in the conditional clause returns true, the block of code immediately following the conditional clause is executed.

When the relational expression returns false, the block of code following the conditional clause is skipped and control passes to the next executable statement following that block of code.

An entry-condition loop

The **while** loop is an entry condition loop, meaning that the test is performed once during each iteration before the block of code is executed.

If the first test returns false, the block of code is skipped entirely.

An exit-condition loop

There is another loop, known as a **do-while** loop, that performs the test after the block of code has been executed once. This guarantees that the block of code will always be executed at least once.

Just to make things interesting ...

Two statements using the increment operator were placed inside the loop in this program.

Therefore, insofar as the conditional test is concerned, the counter is being incremented by twos. This causes the output to display the sequence 1 3 5.

Nested while loops

The **while** loop control structure can contain loops nested inside of loops, which leads to some interesting behavior.

Back to Question 11 (p. 79)

3.4.6.6 Answer 10

F. x != 4,5,6

3.4.6.6.1 Explanation 10

A multiple-choice structure

This is a form of control structure that is often used to make logical decisions in a *multiple-choice* sense.

This is a completely general control structure. It can be used with just about any type of data.

A switch structure

There is a somewhat more specialized, control structure named **switch** that can also be used to make decisions in a multiple choice sense under certain fairly restrictive conditions.

However, the structure shown in this program can always be used to replace a switch. Therefore, I find that I rarely use the **switch** structure, opting instead for the more general form of multiple-choice structure.

Back to Question 10 (p. 78)

3.4.6.7 Answer 9

D. Goodbye World

3.4.6.7.1 Explanation 9

An if-else control structure

This program contains a simple **if-else** control structure.

Behavior of if-else structure

If the expression in the conditional clause returns true, the block of code following the conditional clause is executed, and the block of code following the word **else** is skipped.

If the expression in the conditional clause returns false, the block of code following the conditional clause is skipped, and the block of code following the word **else** is executed.

This program executes the else block

In this program, the expression in the conditional clause returns false.

Therefore, the block of code following the word **else** is executed, producing the words *Goodbye World* on the computer screen.

Can result in very complex structures

While the structure used in this program is relatively simple, it is possible to create very complex control structures by nesting additional **if-else** structures inside the blocks of code.

Back to Question 9 (p. 77)

3.4.6.8 Answer 8

D. Hello World

3.4.6.8.1 Explanation 8

A simple if statement

This program contains a simple **if** statement that

- uses a relational expression
- to return a value of type **boolean** inside its conditional clause

Tests for x less than y

The relational expression tests to determine if the value of the variable named **x** is less than the value of the variable named **y** .

Since the value of **x** is 5 and the value of **y** is 6, this relational expression returns true.

Behavior of an if statement

If the expression in the conditional clause returns true, the block of code following the conditional clause is executed

What is a block of code?

A block of code is one or more statements surrounded by matching curly brackets.

For cases like this one where the block includes only one statement, the curly brackets can be omitted. However, I prefer to put them there anyway. They don't cause any harm and help me avoid programming errors if I come back later and add more statements to the body of the **if** statement.

Display the word Hello

In this program, execution of the code in the block causes the **print** method to be called and the word *Hello* to be displayed followed by a space, but without a newline following the space.

What if the conditional clause returns false?

If the expression in the conditional clause returns false, the block of code following the conditional clause is bypassed.

(That is not the case in this program.)

After the if statement ...

After the **if** statement is executed in this program, the **println** method is called to cause the word *World* to be displayed on the same line as the word *Hello* .

Back to Question 8 (p. 76)

3.4.6.9 Answer 7

A. Compiler Error

3.4.6.9.1 Explanation 7

Not the same as C and C++

Unlike C and C++, which can use an integer numeric expression in the conditional clause of an **if** statement, Java requires the conditional clause of an **if** statement to contain an expression that will return a **boolean** result.

Bad conditional expression

That is not the case in this program, and the following compiler error occurs under JDK 1.3:

NOTE:

```
    Ap031.java:13: incompatible types
found   : int
required: boolean
    if(x - y){
```

Back to Question 7 (p. 75)

3.4.6.10 Answer 6

D. 6 4 9.3 7.300000000000001

3.4.6.10.1 Explanation 6

Postfix increment and decrement operators

This program illustrates the use of the increment (++) and decrement (–) operators in their postfix form.

Behavior of increment operator

Given a variable **x** , the following two statements are equivalent:

NOTE:

```
    x++;
x = x + 1;
```

Behavior of decrement operator

Also, the following two statements are equivalent:

NOTE:

```
    x--;
x = x - 1;
```

Prefix and postfix forms available

These operators have both a prefix form and a postfix form.

Can be fairly complex

It is possible to construct some fairly complex scenarios when using these operators and combining them into expressions.

In these modules ...

In this group of self-assessment modules, the increment and decrement operators will primarily be used to update control variables in loops.

Inaccurate results

Regarding the program output, you will note that there is a little arithmetic inaccuracy when this program is executed using JDK 1.3. *(The same is still true with JDK version 1.7.)*

Ideally, the output value 7.300000000000001 should simply be 7.3 without the very small additional fractional part, but that sort of thing often happens when using floating types.

Back to Question 6 (p. 74)

3.4.6.11 Answer 5

A. Compiler Error

3.4.6.11.1 Explanation 5

Cannot use > with reference variables

The only relational operator that can be applied to reference variables is the == operator.

As discussed in the previous questions, even then it can only be used to determine if two reference variables refer to the same object.

This program produces the following compiler error under JDK 1.3:

NOTE:

```
Ap029.java:14: operator > cannot be applied to Dummy,Dummy
  System.out.println(x > y);
```

Back to Question 5 (p. 73)

3.4.6.12 Answer 4

E. true true

3.4.6.12.1 Explanation 4

Two references to the same object

In this case, the reference variables named **x** and **y** both refer to the same object. Therefore, when tested for equality, using either the == operator or the default **equals** method, the result is true.

Back to Question 4 (p. 72)

3.4.6.13 Answer 3

D. false

3.4.6.13.1 Explanation 3

Read question 2

In case you skipped it, you need to read the explanation for the answer to Question 2 (p. 70) before reading this explanation.

Objects appear to be equal

These two objects are of the same type and contain the same values. Why are they reported as not being equal?

Did not override the equals method

When I defined the class named **Dummy** used in the programs for Question 2 (p. 70) and Question 3 (p. 71) , I did not override the method named **equals** .

Therefore, my class named **Dummy** simply inherited the default version of the method named **equals** that is defined in the class named **Object** .

Default behavior of equals method

The default version of the **equals** method behaves essentially the same as the == operator.

That is to say, the inherited default version of the **equals** method will return true if the two objects being compared are actually the same object, and will return false otherwise.

As a result, this program displays false.

Overridden equals is required for valid testing

If you want to be able to determine if two objects instantiated from a class that you define are "equal", you must override the inherited **equals** method for your new class. You cannot depend on the inherited version of the **equals** method to do that job for you.

Overriding may not be easy

That is not to say that overriding the **equals** method is easy. In fact, it may be quite difficult in those cases where the class declares instance variables that refer to other objects. In this case, it may be necessary to test an entire tree of objects for equality.

Back to Question 3 (p. 71)

3.4.6.14 Answer 2

D. false

3.4.6.14.1 Explanation 2

Use of the == operator with references to objects

This program illustrates an extremely important point about the use of the == operator with objects and reference variables containing references to objects.

You cannot determine...

You cannot determine if two objects are "equal" by applying the == operator to the reference variables containing references to those objects.

Rather, that test simply determines if two reference variables refer to the same object.

Two references to the same object

Obviously, if there is only one object, referred to by two different reference variables, then it is "equal" to itself.

Objects of same type containing same instance values

On the other hand, two objects of the same type could contain exactly the same data values, but this test would not indicate that they are "equal." *(In fact, that is the case in this program.)*

So, how do you test two objects for equal?

In order to determine if two objects are "equal", you must devise a way to compare the types of the two objects and actually compare the contents of one object to the contents of the other object. Fortunately, there is a standard framework for doing this.

The equals method

In particular, the class named **Object** defines a default version of a method named **equals** that is inherited by all other classes.

Class author can override the equals method

The intent is that the author of a new class can override the **equals** method so that it can be called to determine if two objects instantiated from that class are "equal."

What does "equal" mean for objects?

Actually, that is up to the author of the class to decide.

After having made that decision, the author of the class writes that behavior into her overridden version of the method named **equals** .

Back to Question 2 (p. 70)

3.4.6.15 Answer 1

The answer is True.

3.4.6.15.1 Explanation 1

Not much to explain here

There isn't much in the way of an explanation to provide for this program.

Evaluate seven relational expressions

Each of the seven relational expressions in the argument list for the **println** method is evaluated and returns either true or false as a **boolean** value.

Concatenate the individual results, separated by a space

The seven **boolean** results are concatenated, separated by space characters, and displayed on the computer screen.

Brief description of the relational operators

Just in case your aren't familiar with the relational operators, here is a brief description.

Each of these operators returns the **boolean** value true if the specified condition is met. Otherwise, it returns false.

NOTE:

```
== Left operand equals the right operand

!= Left operand is not equal to the right operand

< Left operand is less than the right operand

<= Left operand is less than or equal to the right operand

> Left operand is greater than the right operand

>= Left operand is greater than or equal to the right operand
```

Back to Question 1 (p. 69)

-end-

3.5 Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method[9]

3.5.1 Table of Contents

[9] This content is available online at <http://cnx.org/content/m45260/1.4/>.

3.5.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 103) to easily find and view the listings while you are reading about them.

3.5.3 Questions

3.5.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 94) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 1: Listing for Question 1.

```
    public class Ap039{
  public static void main(
                        String args[]){
    new Worker().doLogical();
  }//end main()
}//end class definition

class Worker{
  public void doLogical(){
    int x = 5, y = 6;
    if((x > y) || (y < x/0)){
      System.out.println("A");
    }else{
      System.out.println("B");
    }//end else
  }//end doLogical()
}//end class definition
```

3.55

Answer and Explanation (p. 111)

3.5.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 95) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 2: Listing for Question 2.

```
    public class Ap040{
  public static void main(
                        String args[]){
    new Worker().doLogical();
  }//end main()
}//end class definition

class Worker{
  public void doLogical(){
    int x = 5, y = 6;
    if((x < y) || (y < x/0)){
      System.out.println("A");
    }else{
      System.out.println("B");
    }//end else
  }//end doLogical()
}//end class definition
```

3.56

Answer and Explanation (p. 110)

3.5.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 96) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. B
- E. None of the above

Listing 3: Listing for Question 3.

```
    public class Ap041{
  public static void main(
                        String args[]){
    new Worker().doLogical();
  }//end main()
}//end class definition

class Worker{
  public void doLogical(){
    int x = 5, y = 6;
    if(!(x < y) && !(y < x/0)){
      System.out.println("A");
    }else{
      System.out.println("B");
    }//end else
  }//end doLogical()
}//end class definition
```

3.57

Answer and Explanation (p. 109)

3.5.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 96) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. 1
- E. None of the above

Listing 4: Listing for Question 4.

```
    public class Ap042{
  public static void main(
                        String args[]){
    new Worker().doCast();
  }//end main()
}//end class definition

class Worker{
  public void doCast(){
    boolean x = true;
    int y = (int)x;
    System.out.println(y);
  }//end doCast()
}//end class definition
```

3.58

Answer and Explanation (p. 108)

3.5.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 98) ?

- A. Compiler Error
- B. Runtime Error
- C. 4 -4
- D. 3 -3
- E. None of the above

Listing 5: Listing for Question 5.

```
    public class Ap043{
  public static void main(
                        String args[]){
    new Worker().doCast();
  }//end main()
}//end class definition

class Worker{
  public void doCast(){
    double w = 3.7;
    double x = -3.7;
    int y = (int)w;
    int z = (int)x;
    System.out.println(y + " " + z);
  }//end doCast()
}//end class definition
```

3.59

Answer and Explanation (p. 108)

3.5.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 99) ?

- A. Compiler Error
- B. Runtime Error
- C. 4 -3
- D. 3 -4
- E. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap044{
  public static void main(
                       String args[]){
    new Worker().doCast();
  }//end main()
}//end class definition

class Worker{
  public void doCast(){
    double w = 3.5;
    double x = -3.499999999999;

    System.out.println(doIt(w) +
                       " " +
                       doIt(x));
  }//end doCast()

  private int doIt(double arg){
    if(arg > 0){
      return (int)(arg + 0.5);
    }else{
      return (int)(arg - 0.5);
    }//end else
  }//end doIt()
}//end class definition
```

3.60

Answer and Explanation (p. 108)

3.5.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 100) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.5/9/true
- D. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap045{
  public static void main(
                        String args[]){
    new Worker().doConcat();
  }//end main()
}//end class definition

class Worker{
  public void doConcat(){
    double w = 3.5;
    int x = 9;
    boolean y = true;
    String z = w + "/" + x + "/" + y;
    System.out.println(z);
  }//end doConcat()
}// end class
```

3.61

Answer and Explanation (p. 106)

3.5.3.8 Question 8

Which of the following best approximates the output from the program shown in Listing 8 (p. 101) ?

- A. Compiler Error
- B. Runtime Error
- C. Dummy@273d3c
- D. Joe 35 162.5

Listing 8: Listing for Question 8.

```
    public class Ap046{
  public static void main(
                        String args[]){
    new Worker().doConcat();
  }//end main()
}//end class definition

class Worker{
  public void doConcat(){
    Dummy y = new Dummy();
    System.out.println(y);
  }//end doConcat()
}// end class

class Dummy{
  private String name = "Joe";
  private int age = 35;
  private double weight = 162.5;
}//end class dummy
```

3.62

Answer and Explanation (p. 105)

3.5.3.9 Question 9

Which of the following best approximates the output from the program shown in Listing 9 (p. 102) ?

- A. Compiler Error
- B. Runtime Error
- C. C. Dummy@273d3c
- D. Joe Age = 35 Weight = 162.5

Listing 9: Listing for Question 9.

```
    public class Ap047{
  public static void main(
                        String args[]){
    new Worker().doConcat();
  }//end main()
}//end class definition

class Worker{
  public void doConcat(){
    Dummy y = new Dummy();
    System.out.println(y);
  }//end doConcat()
}// end class

class Dummy{
  private String name = "Joe";
  private int age = 35;
  private double weight = 162.5;

  public String toString(){
    String x = name + " " +
               " Age = " + age + " " +
               " Weight = " + weight;
    return x;
  }
}//end class dummy
```

3.63

Answer and Explanation (p. 105)

3.5.3.10 Question 10

Which of the following best approximates the output from the program shown in Listing 10 (p. 103) ? *(Note the use of the constructor for the* ***Date*** *class that takes no parameters.)*

- A. Compiler Error
- B. Runtime Error
- C. Sun Dec 02 17:35:00 CST 2012 1354491300781
- D. Thur Jan 01 00:00:00 GMT 1970
- 0
- None of the above

Listing 10: Listing for Question 10.

```
    import java.util.*;
public class Ap048{
  public static void main(
                        String args[]){
    new Worker().doConcat();
  }//end main()
}//end class definition

class Worker{
  public void doConcat(){
    Date w = new Date();
    String y = w.toString();
    System.out.print(y);
    System.out.println(" " + w.getTime());
  }//end doConcat()
}// end class
```

3.64

Answer and Explanation (p. 104)

3.5.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.5.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method
- File: Ap0040.htm

- Originally published: 2002
- Published at cnx.org: 12/02/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.5.6 Answers

3.5.6.1 Answer 10

C. Sun Dec 02 17:35:00 CST 2012 1354491300781

3.5.6.1.1 Explanation 10

The noarg constructor for the Date class

The **Date** class has a constructor that takes no parameters and is described in the documentation as follows:

"Allocates a ***Date*** *object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond."*

In other words, this constructor can be used to instantiate a **Date** object that represents the current date and time according to the system clock.

A property named time of type long

The actual date and time information encapsulated in a **Date** object is apparently stored in a property named **time** as a **long** integer.

Milliseconds since the epoch

The **long** integer encapsulated in a **Date** object represents the total number of milliseconds for the encapsulated date and time, relative to the epoch, which was Jan 01 00:00:00 GMT 1970.

Earlier dates are represented as negative values. Later dates are represented as positive values.

An overridden toString method

An object of the **Date** class has an overridden **toString** method that converts the value in milliseconds to a form that is more useful for a human observer, such as:

Sun Dec 02 17:35:00 CST 2012

Instantiate a Date object with the noarg constructor

This program instantiates an object of the **Date** class using the constructor that takes no parameters.

Call the overridden toString method

Then it calls the overridden **toString** method to populate a **String** object that represents the **Date** object.

Following this, it displays that **String** object by calling the **print** method, producing the first part of the output shown above. *(The actual date and time will vary depending on when the program is executed.)*

Get the time property value

Then it calls the **getTime** method to get and display the value of the **time** property.

This is a representation of the same date and time shown above (p. 104) , but in milliseconds:

1354491300781

Back to Question 10 (p. 102)

3.5.6.2 Answer 9

D. Joe Age = 35 Weight = 162.5

3.5.6.2.1 Explanation 9

Upgraded program from Question 8

The program used for this question is an upgrade to the program that was used for Question 8 (p. 100) .

Dummy class overrides the toString method

In particular, in this program, the class named **Dummy** overrides the **toString** method in such a way as to return a **String** representing the object that would be useful to a human observer.

The **String** that is returned contains the values of the instance variables of the object: name, age, and weight.

Overridden toString method code

The overridden **toString** method for the **Dummy** class is shown below for easy reference.

NOTE: **Overridden *toString* method**

```
  public String toString(){
String x = name + " " +
           " Age = " + age + " " +
           " Weight = " + weight;
return x;
}//end toString()
```

The code in the overridden **toString** method is almost trivial.

The important thing is not the specific code in a specific overridden version of the **toString** method.

Why override the toString method?

Rather, the important thing is to understand why you should probably override the **toString** method in most of the new classes that you define.

In fact, you should override the **toString** method in all new classes that you define if a **String** representation of an instance of that class will ever be needed for any purpose.

The code will vary

The code required to override the **toString** method will vary from one class to another. The important point is that the code must return a reference to a **String** object. The **String** object should encapsulate information that represents the original object in a format that is meaningful to a human observer.

Back to Question 9 (p. 101)

3.5.6.3 Answer 8

C. Dummy@273d3c

3.5.6.3.1 Explanation 8

Display an object of the Dummy class

This program instantiates a new object of the **Dummy** class, and passes that object's reference to the method named **println** .

The purpose of the **println** method is to display a representation of the new object that is meaningful to a human observer. In order to do so, it requires a **String** representation of the object.

The toString method

The class named **Object** defines a default version of a method named **toString** .

All classes inherit the **toString** method.

A child of the Object class

Those classes that extend directly from the class named **Object** inherit the default version of the **toString** method.

Grandchildren of the Object class

Those classes that don't directly extend the class named **Object** also inherit a version of the **toString** method.

May be default or overridden version

The inherited **toString** method may be the default version, or it may be an overridden version, depending on whether the method has been overridden in a superclass of the new class.

The purpose of the toString method

The purpose of the **toString** method defined in the **Object** class is to be overridden in new classes.

The body of the overridden version should return a reference to a **String** object that represents an object of the new class.

Whenever a String representation of an object is required

Whenever a **String** representation of an object is required for any purpose in Java, the **toString** method is called on a reference to the object.

The **String** that is returned by the **toString** method is taken to be a **String** that represents the object.

When toString has not been overridden

When the **toString** method is called on a reference to an object for which the method has not been overridden, the default version of the method is called.

The default **String** representation of an object

The **String** returned by the default version consists of the following:

- The name of the class from which the object was instantiated
- The @ character
- A hexadecimal value that is the **hashcode** value for the object

As you can see, this does not include any information about the values of the data stored in the object.

Other than the name of the class from which the object was instantiated, this is not particularly useful to a human observer.

Dummy class does not override toString method

In this program, the class named **Dummy** extends the **Object** class directly, and doesn't override the **toString** method.

Therefore, when the **toString** method is called on a reference to an object of the **Dummy** class, the **String** that is returned looks something like the following:

Dummy@273d3c

Note that the six hexadecimal digits at the end will probably be different from one program to the next.

Back to Question 8 (p. 100)

3.5.6.4 Answer 7

C. 3.5/9/true

3.5.6.4.1 Explanation 7

More on String concatenation

This program illustrates **String** concatenation.

The plus (+) operator is what is commonly called an *overloaded operator* .

What is an overloaded operator?

An overloaded operator is an operator whose behavior depends on the types of its operands.

Plus (+) as a unary operator

The plus operator can be used as either a **unary** operator or a **binary** operator. However, as a unary operator, with only one operand to its right, it doesn't do anything useful. This is illustrated by the following two statements, which are functionally equivalent.

x = y;

x = +y;

Plus (+) as a binary operator

As a binary operator, the plus operator requires two operands, one on either side. *(This is called infix notation.)* When used as a binary operator, its behavior depends on the types of its operands.

Two numeric operands

If both operands are numeric operands, the plus operator performs arithmetic addition.

If the two numeric operands are of different types, the narrower operand is converted to the type of the wider operand, and the addition is performed as the wider type.

Two String operands

If both operands are references to objects of type **String** , the plus operator creates and returns a new **String** object that contains the concatenated values of the two operands.

One String operand and one of another type

If one operand is a reference to an object of type **String** and the other operand is of some type other than **String** , the plus operator causes a new **String** object to come into existence.

This new **String** object is a **String** representation of the *non-String* operand *(such as a value of type **int**)* ,

Then it concatenates the two **String** objects, producing another new **String** object, which is the concatenation of the two.

How is the new String operand representing the non-string operand created?

The manner in which it creates the new **String** object that represents the non-String operand varies with the actual type of the operand.

A primitive operand

The simplest case is when the non-String operand is one of the primitive types. In these cases, the capability already exists in the core programming language to produce a **String** object that represents the value of the primitive type.

A boolean operand

For example, if the operand is of type **boolean** , the new **String** object that represents the operand will either contain the word true or the word false.

A numeric operand

If the operand is one of the numeric types, the new **String** object will be composed of some of the following:

- numeric characters
- a decimal point character
- minus characters
- plus character
- other characters such as E or e

These characters will be arranged in such a way as to represent the numeric value of the operand to a human observer.

In this program ...

In this program, a numeric **double** value, a numeric **int** value, and a **boolean** value were concatenated with a pair of slash characters to produce a **String** object containing the following:

3.5/9/true

When a reference to this **String** object was passed as a parameter to the **println** method, the code in that method extracted the character string from the **String** object, and displayed that character string on the screen.

The toString method

If one of the operands to the plus operator is a reference to an object, the **toString** method is called on the reference to produce a string that represents the object. The **toString** method may be overridden by the author of the class from which the object was instantiated to produce a **String** that faithfully represents the object.

Back to Question 7 (p. 99)

3.5.6.5 Answer 6

C. 4 -3

3.5.6.5.1 Explanation 6

A rounding algorithm

The method named **doIt** in this program illustrates an algorithm that can be used with a numeric cast operator **(int)** to cause **double** values to be rounded to the nearest integer.

Different than truncation toward zero

Note that this is different from simply truncating to the next integer closer to zero *(as was illustrated in Question 5 (p. 97))* .

Back to Question 6 (p. 98)

3.5.6.6 Answer 5

D. 3 -3

3.5.6.6.1 Explanation 5

Truncates toward zero

When a **double** value is cast to an **int** , the fractional part of the **double** value is discarded.

This produces a result that is the next integer value closer to zero.

This is true regardless of whether the **double** is positive or negative. This is sometimes referred to as its *"truncation toward zero"* behavior.

Not the same as rounding

If each of the values assigned to the variables named **w** and **x** in this program were rounded to the nearest integer, the result would be 4 and -4, not 3 and -3 as produced by the program.

Back to Question 5 (p. 97)

3.5.6.7 Answer 4

A. Compiler Error

3.5.6.7.1 Explanation 4

Cannot cast a boolean type

A **boolean** type cannot be cast to any other type. This program produces the following compiler error:

NOTE:

```
Ap042.java:13: inconvertible types
found   : boolean
required: int
    int y = (int)x;
```

Back to Question 4 (p. 96)

3.5.6.8 Answer 3

D. B

3.5.6.8.1 Explanation 3

The logical and operator

The logical and operator shown below

NOTE: **The *logical and* operator**

```
&&
```

performs an ***and*** operation between its two operands, both of which must be of type **boolean** . If both operands are true, the operator returns true. Otherwise, it returns false.

The boolean negation operator

The boolean negation operator shown below

NOTE: **The *boolean negation* operator** !

is a *unary* operator, meaning that it always has only one operand. That operand must be of type **boolean** , and the operand always appears immediately to the right of the operator.

The behavior of this operator is to change its right operand from true to false, or from false to true.

Evaluation from left to right

Now, consider the following code fragment from this program.

NOTE:

```
    int x = 5, y = 6;
if(!(x < y) && !(y < x/0)){
  System.out.println("A");
}else{
  System.out.println("B");
}//end else
```

The individual operands of the *logical and* operator are evaluated from left to right.

Consider the left operand of the *logical and* operator that reads:

NOTE:

```
!(x<y)
```

The following expression is true

NOTE:

```
(x < y)
```

In this case, **x** is less than **y** , so the expression inside the parentheses evaluates to true.

The following expression is false

NOTE:

```
!(x < y)
```

The true result becomes the right operand for the *boolean negation* operator at this point.

You might think of the state of the evaluation process at this point as being something like **not true** .

When the **!** operator is applied to the **true** result, the combination of the two become a **false** result.

Short-circuit evaluation applies

This, in turn, causes the left operand of the *logical and* operator to be **false** .

At that point, the final outcome of the logical expression has been determined. It doesn't matter whether the right operand is true or false. The final result will be false regardless.

No attempt is made to evaluate the right operand

Therefore, no attempt is made to evaluate the right operand of the *logical and* operator in this case.

No attempt is made to divide the integer variable **x** by zero, no exception is thrown, and the program doesn't terminate abnormally. It runs to completion and displays a B on the screen.

Back to Question 3 (p. 95)

3.5.6.9 Answer 2

C. A

3.5.6.9.1 Explanation 2

Short-circuit evaluation

Question 1 (p. 94) was intended to set the stage for this question.

This Question, in combination with Question 1 (p. 94) , is intended to help you understand and remember the concept of short-circuit evaluation.

What is short-circuit evaluation?

Logical expressions are evaluated from left to right. That is, the left operand of a logical operator is evaluated before the right operand of the same operator is evaluated.

When evaluating a logical expression, the final outcome can often be determined without the requirement to evaluate all of the operands.

Once the final outcome is determined, no attempt is made to evaluate the remainder of the expression. This is short-circuit evaluation.

Code from Question 1

Consider the following code fragment from Question 1 (p. 94) :

NOTE:

```
    int x = 5, y = 6;
if((x > y) || (y < x/0)){
  ...
```

The (||) operator is the *logical or* operator.

Boolean operands required

This operator requires that its left and right operands both be of type **boolean** . This operator performs an *inclusive or* on its left and right operands. The rules for an inclusive or are:

If either of its operands is true, the operator returns true. Otherwise, it returns false.

Left operand is false

In this particular expression, the value of **x** is not greater than the value of **y** . Therefore, the left operand of the *logical or* operator is not true.

Right operand must be evaluated

This means that the right operand must be evaluated in order to determine the final outcome.

Right operand attempts to divide by zero

However, when an attempt is made to evaluate the right operand, an attempt is made to divide **x** by zero. This throws an exception, which is not caught and handled by the program, so the program terminates as described in Question 1 (p. 94) .

Similar code from Question 2

Now consider the following code fragment from Question 2 (p. 94) .

NOTE:

```
    int x = 5, y = 6;
if((x < y) || (y < x/0)){
  System.out.println("A");
...
```

Note that the right operand of the *logical or* operator still contains an expression that attempts to divide the integer x by zero.

No runtime error in this case

This program does not terminate with a runtime error. Why not?

And the answer is ...

In this case, **x** is less than **y** . Therefore, the left operand of the *logical or* operator is true.

Remember the rule for inclusive or

It doesn't matter whether the right operand is true or false. The final outcome is determined as soon as it is determined that the left operand is true.

The bottom line

Because the final outcome has been determined as soon as it is determined that the left operand is true, no attempt is made to evaluate the right operand.

Therefore, no attempt is made to divide **x** by zero, and no runtime error occurs.

Short-circuit evaluation

This behavior is often referred to as *short-circuit evaluation* .

Only as much of a logical expression is evaluated as is required to determine the final outcome.

Once the final outcome is determined, no attempt is made to evaluate the remainder of the logical expression.

This is not only true for the *logical or* operator, it is also true for the *logical and* operator, which consists of two ampersand characters with no space between them.

Back to Question 2 (p. 94)

3.5.6.10 Answer 1

B. Runtime Error

3.5.6.10.1 Explanation 1

Divide by zero

Whenever a Java program attempts to evaluate an expression requiring that a value of one of the integer types be divided by zero, it will throw an **ArithmeticException** . If this exception is not caught and handled by the program, it will cause the program to terminate.

Attempts to divide x by 0
This program attempts to evaluate the following expression:

NOTE:

```
(y < x/0)
```

This expression attempts to divide the variable named **x** by zero. This causes the program to terminate with the following error message when running under JDK 1.3:

NOTE:

```
java.lang.ArithmeticException: / by zero
at Worker.doLogical(Ap039.java:13)
at Ap039.main(Ap039.java:6)
```

Back to Question 1 (p. 94)
-end-

3.6 Ap0050: Self-assessment, Escape Character Sequences and Arrays[10]

3.6.1 Table of Contents

3.6.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 127) to easily find and view the listings while you are reading about them.

3.6.3 Questions

3.6.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 113) ?

NOTE:

[10] This content is available online at <http://cnx.org/content/m45280/1.4/>.

```
       A. Compiler Error
   B. Runtime Error
   C. \"Backslash\"->\\\nUnderstand
   D. "Backslash"->\
      Understand
```

Listing 1: Listing for Question 1.

```
   public class Ap049{
  public static void main(
                       String args[]){
    new Worker().doEscape();
  }//end main()
}//end class definition

class Worker{
  public void doEscape(){
    System.out.println(
      "\"Backslash\"->\\\nUnderstand");
  }//end doEscape()
}// end class
```

3.65

Answer and Explanation (p. 140)

3.6.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 114) ?

- A. Compiler Error
- B. Runtime Error
- C. St@273d3c St@256a7c St@720eeb
- D. Tom Dick Harry
- E. None of the above

Listing 2: Listing for Question 2.

```
    public class Ap050{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    St[] myArray = {new St("Tom"),
                    new St("Dick"),
                    new St ("Harry")};
  for(int cnt = 0;
           cnt < myArray.length;cnt++){
    System.out.print(
                   myArray[cnt] + " ");

  }//end for loop
  System.out.println("");
  }//end doArrays()
}// end class

class St{
  private String name;

  public St(String name){
    this.name = name;
  }//end constructor

  public String toString(){
    return name;
  }//end toString()
}//end class
```

3.66

Answer and Explanation (p. 138)

3.6.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 115) ?

NOTE:

A. Compiler Error

```
B.  Runtime Error

C.  0 0 0 0 0
    0 1 2 3 4
    0 2 4 6 8

D.  None of the above
```

Listing 3: Listing for Question 3.

```
    public class Ap051{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[3][5];

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        myArray[i][j] = i*j;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        System.out.print(
                  myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

3.67

Answer and Explanation (p. 136)

3.6.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 116) ?

NOTE:

A. Compiler Error

B. Runtime Error

C. 1 1 1 1 1
1 2 3 4 5
1 3 5 7 9

D. None of the above

Listing 4: Listing for Question 4.

```
    public class Ap052{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[][];
    myArray = new int[3][5];

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        myArray[i][j] = i*j + 1;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        System.out.print(
                  myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

3.68

Answer and Explanation (p. 135)

3.6.3.5 Question 5

What output is produced by program shown in Listing 5 (p. 117) ?

NOTE:

A. Compiler Error

B. Runtime Error

C. -1 -1 -1 -1 -1
-1 -2 -3 -4 -5
-1 -3 -5 -7 -9

D. None of the above

Listing 5: Listing for Question 5.

```
    public class Ap053{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[][];
    myArray = new int[3][5];

    for(int i = 0;i < 3;i++){
      for(int j = 0;j < 5;j++){
        myArray[i][j] = (i*j+1)*(-1);
      }//end inner for loop
    }//end outer for loop

    for(int i = 0;i < 3;i++){
      for(int j = 0;j < 6;j++){
        System.out.print(
                  myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

3.69

Answer and Explanation (p. 134)

3.6.3.6 Question 6

What output is produced by program shown in Listing 6 (p. 118) ?

- A. Compiler Error
- B. Runtime Error
- C. 3
- D. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap054{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;
    System.out.println(A[0] + A[1]);

  }//end doArrays()
}// end class
```

3.70

Answer and Explanation (p. 134)

3.6.3.7 Question 7

What output is produced by program shown in Listing 7 (p. 119) ?

- A. Compiler Error
- B. Runtime Error
- C. OK
- D. None of the above

Listing 7: Listing for Question 7.

```
    import java.awt.Label;
public class Ap055{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Label[] A = new Label[2];
    A[0] = new Label("O");
    A[1] = new Label("K");
    System.out.println(A[0] + A[1]);
  }//end doArrays()
}// end class
```

3.71

Answer and Explanation (p. 132)

3.6.3.8 Question 8

What output is produced by program shown in Listing 8 (p. 120) ?

- A. Compiler Error
- B. Runtime Error
- C. OK
- D. None of the above

Listing 8: Listing for Question 8.

```
import java.awt.Label;
public class Ap056{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Label[] A = new Label[2];
    A[0] = new Label("O");
    A[1] = new Label("K");
    System.out.println(A[0].getText() + A[1].getText());
  }//end doArrays()
}// end class
```

3.72

Answer and Explanation (p. 131)

3.6.3.9 Question 9

What output is produced by program shown in Listing 9 (p. 121) ?

- A. Compiler Error
- B. Runtime Error
- C. 1
- D. None of the above

Listing 9: Listing for Question 9.

```
    public class Ap057{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Integer[] A = new Integer[2];
    A[0] = new Integer(1);
    System.out.println(
                      A[1].intValue());
  }//end doArrays()
}// end class
```

3.73

Answer and Explanation (p. 131)

3.6.3.10 Question 10

What output is produced by program shown in Listing 10 (p. 122) ?

NOTE:

A. Compiler Error

B. Runtime Error

C. 0
0 1
0 2 4

D. None of the above

Listing 10: Listing for Question 10.

```
    public class Ap058{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int A[][] = new int[3][];
    A[0] = new int[1];
    A[1] = new int[2];
    A[2] = new int[3];

    for(int i = 0;i < A.length;i++){
      for(int j=0;j < A[i].length;j++){
        A[i][j] = i*j;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<A.length;i++){
      for(int j=0;j < A[i].length;j++){
        System.out.print(
                        A[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

3.74

Answer and Explanation (p. 130)

3.6.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 123) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero One Two
- D. None of the above

Listing 11: Listing for Question 11.

```
    public class Ap059{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new String("One");
    A[2] = new String("Two");

    System.out.println(A[0] + " " +
                       A[1] + " " +
                       A[2]);
  }//end doArrays()
}// end class
```

3.75

Answer and Explanation (p. 130)

3.6.3.12 Question 12

What output is produced by program shown in Listing 12 (p. 124) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero 1 2.0
- D. None of the above.

Listing 12: Listing for Question 12.

```
    public class Ap060{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new Integer(1);
    A[2] = new Double(2.0);

    System.out.println(A[0] + "  " +
                       A[1] + "  " +
                       A[2]);
  }//end doArrays()
}// end class
```

3.76

Answer and Explanation (p. 129)

3.6.3.13 Question 13

What output is produced by program shown in Listing 13 (p. 125) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero 1 2.0
- D. None of the above.

Listing 13: Listing for Question 13.

```
    public class Ap061{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new Integer(1);
    A[2] = new MyClass(2.0);

    System.out.println(A[0] + "  " +
                       A[1] + "  " +
                       A[2]);
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor
}// end MyClass
```

3.77

Answer and Explanation (p. 129)

3.6.3.14 Question 14

What output is produced by program shown in Listing 14 (p. 126) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 14: Listing for Question 14.

```
    public class Ap062{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[2];
    A[0] = new MyClass(1.0);
    A[1] = new MyClass(2.0);

    System.out.println(
      A[0].getData() + "  " +
      A[1].getData());
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor

  public double getData(){
    return data;
  }//end getData()
}// end MyClass
```

3.78

Answer and Explanation (p. 129)

3.6.3.15 Question 15

What output is produced by program shown in Listing 15 (p. 127) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 15: Listing for Question 15.

```
    public class Ap063{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[2];
    A[0] = new MyClass(1.0);
    A[1] = new MyClass(2.0);

    System.out.println(
      ((MyClass)A[0]).getData() + "  "
      + ((MyClass)A[1]).getData());
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor

  public double getData(){
    return data;
  }//end getData()
}// end MyClass
```

3.79

Answer and Explanation (p. 128)

3.6.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 1 (p. 113) . Listing for Question 1.
- Listing 2 (p. 114) . Listing for Question 2.
- Listing 3 (p. 115) . Listing for Question 3.
- Listing 4 (p. 116) . Listing for Question 4.
- Listing 5 (p. 117) . Listing for Question 5.
- Listing 6 (p. 118) . Listing for Question 6.

- Listing 7 (p. 119) . Listing for Question 7.
- Listing 8 (p. 120) . Listing for Question 8.
- Listing 9 (p. 121) . Listing for Question 9.
- Listing 10 (p. 122) . Listing for Question 10.
- Listing 11 (p. 123) . Listing for Question 11.
- Listing 12 (p. 124) . Listing for Question 12.
- Listing 13 (p. 125) . Listing for Question 13.
- Listing 14 (p. 126) . Listing for Question 14.
- Listing 15 (p. 127) . Listing for Question 15.

3.6.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0050: Self-assessment, Escape Character Sequences and Arrays
- File: Ap0050.htm
- Originally published: 2002
- Published at cnx.org: 12/03/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.6.6 Answers

3.6.6.1 Answer 15

C. 1.0 2.0

3.6.6.1.1 Explanation 15

This is an upgrade of the program from Question 14 (p. 125) . This program applies the proper downcast operator to the references extracted from the array of type **Object** before attempting to call the method named **getData** on those references. *(For more information, see the discussion of Question 14* (p. 125) *.)*

As a result of applying a proper downcast, the program compiles and runs successfully.
Back to Question 15 (p. 126)

3.6.6.2 Answer 14

A. Compiler Error

3.6.6.2.1 Explanation 14

Storing references in a generic array of type Object

This program stores references to two objects instantiated from a new class named **MyClass** in the elements of an array object of declared type **Object** . That is OK.

Calling a method on the references

Then the program extracts the references to the two objects and attempts to call the method named **getData** on each of the references. That is not OK.

Downcast is required

Because the method named **getData** is not defined in the class named **Object** , in order to call this method on references extracted from an array of type **Object** , it is necessary to downcast the references to the class in which the method is defined. In this case, the method is defined in the new class named **MyClass** *(but it could be defined in an intermediate class in the class hierarchy if the new class extended some class further down the hierarchy)* .

Here is a partial listing of the compiler error produced by this program:

NOTE:

```
Ap062.java:15: error: cannot find symbol
  A[0].getData() + "  " +
       ^
symbol:   method getData()
location: class Object
```

Back to Question 14 (p. 125)

3.6.6.3 Answer 13

D. None of the above.

3.6.6.3.1 Explanation 13

The array object of type **Object** in this program is capable of storing a reference to a new object instantiated from the new class named **MyClass** . However, because the new class does not override the **toString** method, when a string representation of the new object is displayed, the string representation is created using the version of the **toString** method that is inherited from the **Object** class. That causes this program to produce an output similar to the following:

Zero 1 MyClass@273d3c

Back to Question 13 (p. 124)

3.6.6.4 Answer 12

C. Zero 1 2.0

3.6.6.4.1 Explanation 12

A type-generic array object

As explained in Question 11 (p. 122) , an array object of the type **Object** is a generic array that can be used to store references to objects instantiated from any class.

Storing mixed reference types

This program instantiates objects from the classes **String** , **Integer** , and **Double** , and stores those object's references in the elements of an array of type **Object** . Then the program accesses the references and uses them to display string representations of each of the objects.

Polymorphic behavior applies

Once again, polymorphic behavior involving overridden versions of the **toString** method were involved and it was not necessary to downcast the references to their true type to display string representations of the objects.

Back to Question 12 (p. 123)

3.6.6.5 Answer 11

C. Zero One Two

3.6.6.5.1 Explanation 11

Storing references to subclass types

When you create an array object for a type defined by a class definition, the elements of the array can be used to store references to objects of that class or any subclass of that class.

A type-generic array object

All classes in Java are subclasses of the class named **Object** . This program creates an array object with the declared type being type **Object** . An array of type **Object** can be used to store references to objects instantiated from any class.

After creating the array object, this program instantiates three objects of the class **String** and stores those object's references in the elements of the array. *(As I pointed out in the comments, there is a simpler and better way to instantiate* ***String*** *objects, but it wouldn't illustrate my point as well as doing the way that I did.)*

Sometimes you need to downcast

Although an array of type **Object** can be used to store references to objects of any type *(including mixed types and references to other array objects)* , you will sometimes need to downcast those references back to their true type once you extract them from the array and attempt to use them for some purpose.

Polymorphic behavior applies here

For this case, however, because the **toString** method is defined in the **Object** class and overridden in the **String** class, polymorphic behavior applies and it is not necessary to downcast the references to type **String** in order to be able to convert them to strings and display them.

Back to Question 11 (p. 122)

3.6.6.6 Answer 10

NOTE:

```
C.  0
0 1
0 2 4
```

3.6.6.6.1 Explanation 10

Defer size specification for secondary arrays

It is not necessary to specify the sizes of the secondary arrays when you create a multi-dimensional array in Java. Rather, since the elements in the primary array simply contain references to other array objects *(or null by default)* , you can defer the creation of those secondary array objects until later.

Independent array objects

When you do finally create the secondary arrays, they are essentially independent array objects *(except for the requirement for type commonality among them)* .

Ragged arrays

Each individual secondary array can be of any size, and this leads to the concept of a *ragged array* . *(On a two-dimensional basis, a ragged array might be thought of as a two-dimensional array where each row can have a different number of columns.)*

This program creates, populates, and displays the contents of such a two-dimensional ragged array. Although this program creates a two-dimensional array that is triangular in shape, even that is not a requirement. The number of elements in each of the secondary arrays need have no relationship to the number of elements in any of the other secondary arrays.

Back to Question 10 (p. 121)

3.6.6.7 Answer 9

B. Runtime Error

3.6.6.7.1 Explanation 9

NullPointerException

The following code fragment shows that this program attempts to perform an illegal operation on the value accessed from the array object at index 1.

NOTE:

```
    Integer[] A = new Integer[2];
A[0] = new Integer(1);
System.out.println(
                  A[1].intValue());
```

You can't call methods on null references

The reference value that was returned by accessing **A[1]** is the default value of null. This is the value that was deposited in the element when the array object was created *(no other value was ever stored there)* . When an attempt was made to call the **intValue** method on that reference value, the following runtime error occurred

NOTE:

```
    java.lang.NullPointerException
 at Worker.doArrays(Ap057.java:14)
 at Ap057.main(Ap057.java:6)
```

This is a common programming error, and most Java programmers have seen an error message involving a **NullPointerException** several **(perhaps many)** times during their programming careers.

Back to Question 9 (p. 120)

3.6.6.8 Answer 8

C. OK

3.6.6.8.1 Explanation 8

Success at last

This program finally gets it all together and works properly. In particular, after accessing the reference values stored in each of the elements, the program does something legal with those values.

Call methods on the object's references

In this case, the code calls one of the public methods belonging to the objects referred to by the reference values stored in the array elements.

NOTE:

```
System.out.println(A[0].getText() + A[1].getText());
```

The **getText** method that is called, returns the contents of the **Label** object as type **String** . This makes it possible to perform **String** concatenation on the values returned by the method, so the program compiles and executes properly.

Back to Question 8 (p. 119)

3.6.6.9 Answer 7

A. Compiler Error

3.6.6.9.1 Explanation 7

Java arrays may seem different to you

For all types other than the primitive types, you may find the use of arrays in Java to be considerably different from what you are accustomed to in other programming languages. There are a few things that you should remember.

Array elements may contain default values

If the declared type of an array is one of the primitive types, the elements of the array contain values of the declared type. If you have not initialized those elements or have not assigned specific values to the elements, they will contain default values.

The default values

You need to know that:

- The default for numeric primitive types is the zero value for that type
- The default for the **boolean** type is false
- The default for the **char** type is a 16-bit unsigned integer, all of whose bits have a zero value *(sometimes called a null character)*
- The default value for reference types is *null* , not to be confused with the *null character* above. *(An array element that contains null doesn't refer to an object.)*

Arrays of references

If the declared type for the array is not one of the primitive types, the elements in the array are actually reference variables. Objects are never stored directly in a Java array. Only references to objects are stored in a Java array.

If the array type is the name of a class ...

If the declared type is the name of a class, references to objects of that class or any subclass of that class can be stored in the elements of the array.

If the array type is the name of an interface ...

If the declared type is the name of an interface, references to objects of any class that implements the interface, or references to objects of any subclass of a class that implements the interface can be stored in the elements of the array.

Why did this program fail to compile?

Now back to the program at hand. Why did this program fail to compile? To begin with, this array was not designed to store any of the primitive types. Rather, this array was designed to store references to objects instantiated from the class named **Label**, as indicated in the following fragment.

NOTE:

```
Label[] A = new Label[2];
```

Elements initialized to null

This is a two-element array. When first created, it contains two elements, each having a default value of *null*. What this really means is that the reference values stored in each of the two elements don't initially refer to any object.

Populate the array elements

The next fragment creates two instances *(objects)* of the **Label** class and assigns those object's references to the two elements in the array object. This is perfectly valid.

NOTE:

```
A[0] = new Label("0");
A[1] = new Label("K");
```

You cannot add reference values

The problem arises in the next fragment. Rather than dealing with the object's references in an appropriate manner, this fragment attempts to access the text values of the two reference variables and concatenate those values.

NOTE:

```
System.out.println(A[0] + A[1]);
```

The compiler produces the following error message:

NOTE:

```
Ap055.java:14: error: bad operand types for binary operator '+'
    System.out.println(A[0] + A[1]);
                               ^
  first type:  Label
  second type: Label
1 error
```

This error message is simply telling us that it is not legal to add the values of reference variables.

Not peculiar to arrays

This problem is not peculiar to arrays. You would get a similar error if you attempted to add two reference variables even when they aren't stored in an array. In this case, the code to access the values of the elements is good. The problem arises when we attempt to do something illegal with those values after we access them.

Usually two steps are required

Therefore, except in some special cases such as certain operations involving the wrapper classes, to use Java arrays with types other than the primitive types, when you access the value stored in an element of the array *(a reference variable)* you must perform only those operations on that reference variable that are legal for an object of that type. That usually involves two steps. The first step accesses the reference to an object. The second step performs some operation on the object.

Back to Question 7 (p. 118)

3.6.6.10 Answer 6

C. 3

3.6.6.10.1 Explanation 6

Once you create an array object for a primitive type in Java, you can treat the elements of the array pretty much as you would treat the elements of an array in other programming languages. In particular, a statement such the following can be used to assign a value to an indexed element in an array referred to by a reference variable named **A** .

NOTE:

```
A[1] = 2;
```

Similarly, when you reference an indexed element in an expression such as the following, the value stored in the element is used to evaluate the expression.

NOTE:

```
System.out.println(A[0] + A[1]);
```

For all Java arrays, you must remember to create the new array object and to store the array object's reference in a reference variable of the correct type. Then you can use the reference variable to gain access to the elements in the array.

Back to Question 6 (p. 118)

3.6.6.11 Answer 5

B. Runtime Error

3.6.6.11.1 Explanation 5

Good fences make good neighbors

One of the great things about an array object in Java is that it knows how to protect its boundaries.

Unlike some other currently popular programming languages, if your program code attempts to access a Java array element outside its boundaries, an exception will be thrown. If your program doesn't catch and handle the exception, the program will be terminated.

Abnormal termination

While experiencing abnormal program termination isn't all that great, it is better than the alternative of using arrays whose boundaries aren't protected. Programming languages that don't protect the array boundaries simply overwrite other data in memory whenever the array boundaries are exceeded.

Attempt to access out of bounds element

The code in the **for** loop in the following fragment attempts to access the array element at the index value 5. That index value is out of bounds of the array.

NOTE:

```
    for(int j = 0;j < 6;j++){
  System.out.print(
            myArray[i][j] + " ");
}//end inner for loop
```

Because that index value is outside the boundaries of the array, an **ArrayIndexOutOfBoundsException** is thrown. The exception isn't caught and handled by program code, so the program terminates abnormally at runtime.

This program also illustrates that it is usually better to use the **length** property of an array to control iterative loops than to use hard-coded limit values, which may be coded erroneously.

Back to Question 5 (p. 117)

3.6.6.12 Answer 4

NOTE:

```
C.  1 1 1 1 1
1 2 3 4 5
1 3 5 7 9
```

3.6.6.12.1 Explanation 4

A two-dimensional array

This program illustrates how to create, populate, and process a two-dimensional array with three rows and five columns.

(As mentioned earlier, a Java programmer who understands the fine points of the language probably wouldn't call this a two-dimensional array. Rather, this is a one-dimensional array containing three elements. Each of those elements is a reference to a one-dimensional array containing five elements. That is the more general way to think of Java arrays.)

The following code fragment creates the array, using one of the acceptable formats discussed in Question 3 (p. 114) .

NOTE:

```
    int myArray[][];
myArray = new int[3][5];
```

Populating the array

The next code fragment uses a pair of nested **for** loops to populate the elements in the array with values of type **int** .

NOTE:

```
    for(int i=0;i<myArray.length;i++){
  for(int j=0;
          j<myArray[0].length;j++){
    myArray[i][j] = i*j + 1;
  }//end inner for loop
}//end outer for loop
```

This is where the analogy of a two-dimensional array falls apart. It is much easier at this point to think in terms of a three-element primary array, each of whose elements contains a reference to a secondary array containing five elements. *(Note that in Java, the secondary arrays don't all have to be of the same size. Hence, it is possible to create odd-shaped multi-dimensional arrays in Java.)*

Using the length property

Pay special attention to the two chunks of code that use the length properties of the arrays to determine the number of iterations for each of the **for** loops.

The first chunk determines the number of elements in the primary array. In this case, the length property contains the value 3.

The second chunk determines the number of elements in the secondary array that is referred to by the contents of the element at index 0 in the primary array. *(Think carefully about what I just said.)*

In this case, the length property of the secondary array contains the value 5.

Putting data into the secondary array elements

The code interior to the inner loop simply calculates some numeric values and stores those values in the elements of the three secondary array objects.

Let's look at a picture

Here is a picture that attempts to illustrate what is really going on here. I don't know if it will make sense to you or not, but hopefully, it won't make the situation any more confusing than it might already be.

NOTE:

```
[->]  [1][1][1][1][1]

[->]  [1][2][3][4][5]

[->]  [1][3][5][7][9]
```

The primary array

The three large boxes on the left represent the individual elements of the three-element primary array. The length property for this array has a value of 3. The arrows in the boxes indicate that the content of each of these three elements is a reference to one of the five-element arrays on the right.

The secondary arrays

Each of the three rows of five boxes on the right represents a separate five-element array object. Each element in each of those array objects contains the **int** value shown. The length property for each of those arrays has a value of 5.

Access and display the array data

The code in the following fragment is another pair of nested **for** loops.

NOTE:

```
  for(int i=0;i<myArray.length;i++){
for(int j=0;
       j<myArray[0].length;j++){
  System.out.print(
            myArray[i][j] + " ");
  }//end inner for loop
  System.out.println("");
}//end outer for loop
```

In this case, the code in the inner loop accesses the contents of the individual elements in the three five-element arrays and displays those contents. If you understand the earlier code in this program, you shouldn't have any difficulty understanding the code in this fragment.

Back to Question 4 (p. 115)

3.6.6.13 Answer 3

A. Compiler Error

3.6.6.13.1 Explanation 3

An incorrect statement

The following statement is not the proper way to create an array object in Java.

NOTE:

```
int myArray[3][5];
```

This statement caused the program to fail to compile, producing several error messages.

What is the correct syntax?

There are several different formats that can be used to create an array object in Java. One of the acceptable ways was illustrated by the code used in Question 2 (p. 113) . Three more acceptable formats are shown below.

NOTE:

```
int[][] myArrayA = new int[3][5];

int myArrayB[][] = new int[3][5];

int myArrayC[][];
myArrayC = new int[3][5];
```

Two steps are required

The key thing to remember is that an array is an object in Java. Just like all other *(non-anonymous)* objects in Java, there are two steps involved in creating and preparing an object for use.

Declare a reference variable

The first step is to declare a reference variable capable of holding a reference to the object.

The second step

The second step is to create the object and to assign the object's reference to the reference variable. From that point on, the reference variable can be used to gain access to the object.

Two steps can often be combined

Although there are two steps involved, they can often be combined into a single statement, as indicated by the first two acceptable formats shown above.

In both of these formats, the code on the left of the assignment operator declares a reference variable. The code on the right of the assignment operator creates a new array object and returns the array object's reference. The reference is assigned to the new reference variable declared on the left.

A two-dimensional array object

In the code fragments shown above, the array object is a two-dimensional array object that can be thought of as consisting of three rows and five columns.

(Actually, multi-dimensional array objects in Java can be much more complex than this. In fact, although I have referred to this as a two-dimensional array object, there is no such thing as a multi-dimensional array object in Java. The concept of a multi-dimensional array in Java is achieved by creating a tree structure of single-dimensional array objects that contain references to other single-dimensional array objects.)

The square brackets in the declaration

What about the placement and the number of matching pairs of empty square brackets? As indicated in the first two acceptable formats shown above, the empty square brackets can be next to the name of the type or next to the name of the reference variable. The end result is the same, so you can use whichever format you prefer.

How many pairs of square brackets are required?

Also, as implied by the acceptable formats shown above, the number of matching pairs of empty square brackets must match the number of so-called *dimensions* of the array. *(This tells the compiler to create*

a reference variable capable of holding a reference to a one-dimensional array object, whose elements are capable of holding references to other array objects.)

Making the two steps obvious

A third acceptable format, also shown above, separates the process into two steps.

One statement in the third format declares a reference variable capable of holding a reference to a two-dimensional array object containing data of type **int** . When that statement finishes executing, the reference variable exists, but it doesn't refer to an actual array object. The next statement creates an array object and assigns that object's reference to the reference variable.

Back to Question 3 (p. 114)

3.6.6.14 Answer 2

D. Tom Dick Harry

3.6.6.14.1 Explanation 2

An array is an object in Java

An array is a special kind of object in Java. Stated differently, all array structures are encapsulated in objects in Java. Further. all array structures are one-dimensional. I often refer to this special kind of object as an *array object* .

An array object always has a property named **length** . The value of the **length** property is always equal to the number of elements in the array. Thus, a program can always determine the size of an array be examining its **length** property.

Instantiating an array object

An array object can be instantiated in at least two different ways:

1. By using the **new** operator in conjunction with the type of data to be stored in the array.
2. By specifying an initial value for every element in the array, in which case the **new** operator is not used.

This program uses the second of the two ways listed above.

Declaring a reference variable for an array object

The following code fragment was extracted from the method named doArrays().

NOTE:

```
St[] myArray = {new St("Tom"),
                new St("Dick"),
                new St ("Harry")};
```

The code to the left of the assignment operator declares a reference variable named **myArray** . This reference variable is capable of holding a reference to an array object that contains an unspecified number of references to objects instantiated from the class named **St** *(or any subclass of the class named St)* .

Note the square brackets

You should note the square brackets in the declaration of the reference variable in the above code *(the declaration of a reference variable to hold a reference to an ordinary object doesn't include square brackets)* .

Create the array object

The code to the right of the assignment operator in the above fragment causes the new array object to come into being. Note that the **new** operator is not used to create the array object in this case. *(This is one of the few cases in Java, along with a literal* ***String*** *object, where it is possible to create a new object without using either the* ***new*** *operator or the* ***newInstance*** *method of the class whose name is* ***Class*** *.)*

Populate the array object

This syntax not only creates the new array object, it also populates it. The new array object created by the above code contains three elements, because three initial values were provided. The initial values are separated by commas in the initialization syntax.

Also instantiates three objects of the St class

The code in the above fragment also instantiates three objects of the class named **St** . Once the array object has come into being, each of the three elements in the array contains a reference to a new object of the class **St** . Each of those objects is initialized to contain the name of a student by using a parameterized constructor that is defined in the class.

The length property value is 3

Following execution of the above code, the **length** property of the array object will contain a value of 3, because the array contains three elements, one for each initial value that was provided.

Using the length property

The code in the following fragment uses the **length** property of the array object in the conditional clause of a **for** loop to display a **String** representation of each of the objects.

NOTE:

```
  for(int cnt = 0;
  cnt < myArray.length;
  cnt++){
System.out.print(
              myArray[cnt] + " ");
```

Overridden toString method

The class named **St** , from which each of the objects was instantiated, defines an overridden **toString** method that causes the string representation of an object of that class to consist of the **String** stored in an instance variable of the object.

Thus, the **for** loop shown above displays the student names that were originally encapsulated in the objects when they were instantiated.

The class named St

The code in the following fragment shows the beginning of the class named **St** including one instance variable and a parameterized constructor.

NOTE:

```
  class St{
private String name;

public St(String name){
  this.name = name;
}//end constructor
```

A very common syntax

This constructor makes use of a very common syntax involving the reference named **this** . Basically, this syntax says to get the value of the incoming parameter whose name is **name** and to assign that value to the instance variable belonging to *this object* whose name is also **name** .

Initializing the object of type St

Each time a new object of the **St** class is instantiated, that object contains an instance variable of type **String** whose value matches the **String** value passed as a parameter to the constructor.

Overridden toString method

The overridden **toString** method for the class named **St** is shown in the following code fragment.

NOTE:

```
      public String toString(){
    return name;
  }//end toString()
```

This version causes the value in the **String** object, referred to by the instance variable named **name**, to be returned when it is necessary to produce a **String** representation of the object.

Back to Question 2 (p. 113)

3.6.6.15 Answer 1

The answer is item D, which reads as follows:

NOTE:

```
    "Backslash"->\
Understand
```

3.6.6.15.1 Explanation 1

Don't confuse the compiler

If you include certain characters inside a literal **String**, you will confuse the compiler. For example, if you simply include a quotation mark (") inside a literal **String**, the compiler will interpret that as the end of the string. From that point on, everything will be out of synchronization. Therefore, in order to include a quotation mark inside a literal string, you must precede it with a backslash character like this:

\"

Multiple lines

If you want your string to comprise two or more physical lines, you can include a newline code inside a **String** by including the following in the string:

\n

Escape character sequences

These character sequences are often referred to as *escape character sequences*. Since the backslash is used as the first character in such a sequence, if you want to include a backslash in a literal string, you must do it like this:

\\

There are some other escape sequences used in Java as well. You would do well to learn how to use them before going to an interview for a job as a Java programmer.

Back to Question 1 (p. 112)

-end-

3.7 Ap0060: Self-assessment, More on Arrays[11]

3.7.1 Table of Contents

[11] This content is available online at <http://cnx.org/content/m45264/1.4/>.

3.7.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 158) to easily find and view the listings while you are reading about them.

3.7.3 Questions

3.7.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 141) ?

- A. Compiler Error
- B. Runtime Error
- C. I'm OK
- D. None of the above

Listing 1: Listing for Question 1.

```
    public class Ap064{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println("I'm OK");
  }//end doArrays()
}// end class
```

3.80

Answer and Explanation (p. 167)

3.7.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 142) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap065{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class
```

3.81

Answer and Explanation (p. 166)

3.7.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 143) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 3: Listing for Question 3.

```
    public class Ap066{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    double C = (double)B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

3.82

Answer and Explanation (p. 166)

3.7.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 144) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 4: Listing for Question 4.

```
    public class Ap067{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    double[] C = (double[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

3.83

Answer and Explanation (p. 166)

3.7.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 145) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 5: Listing for Question 5.

```
    public class Ap068{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    String[] C = (String[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

3.84

Answer and Explanation (p. 165)

3.7.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 146) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap069{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    System.out.println(
                    A[0] + " " + A[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

3.85

Answer and Explanation (p. 165)

3.7.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 148) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap070{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Superclass[] B = A;
    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

3.86

Answer and Explanation (p. 164)

3.7.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 150) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 8: Listing for Question 8.

```
    public class Ap071{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Superclass[] A = new Superclass[2];
    A[0] = new Superclass(1);
    A[1] = new Superclass(2);

    Subclass[] B = (Subclass[])A;
    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

3.87

Answer and Explanation (p. 164)

3.7.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 152) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 9: Listing for Question 9.

```
    public class Ap072{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Superclass[] B = A;
    Subclass[] C = (Subclass[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

3.88

Answer and Explanation (p. 163)

3.7.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 153) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. D. None of the above

Listing 10: Listing for Question 10.

```
    public class Ap073{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println(
              ((double[])B)[0] + " " +
              ((double[])B)[1]);
  }//end doArrays()
}// end class
```

3.89

Answer and Explanation (p. 163)

3.7.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 154) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 11: Listing for Question 11.

```
    public class Ap074{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;

    double[] B = (double[])A;

    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class
```

3.90

Answer and Explanation (p. 162)

3.7.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 155) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 12: Listing for Question 12.

```
    public class Ap075{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] B = returnArray();
    for(int i = 0; i < B.length;i++){
      System.out.print(B[i] + " ");
    }//end for loop
    System.out.println();
  }//end doArrays()

  public int[] returnArray(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;
    return A;
  }//end returnArray()
}// end class
```

3.91

Answer and Explanation (p. 162)

3.7.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 156) ?

NOTE:

A. Compiler Error

B. Runtime Error

C. 0 0 0
0 1 2

D. None of the above

Listing 13: Listing for Question 13.

```
    public class Ap076{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A[];
    A = new int[2][3];

    for(int i=0; i<A.length;i++){
      for(int j=0;j<A[0].length;j++){
        A[i][j] = i*j;
      }//end inner loop
    }//end outer loop

    for(int i=0; i<A.length;i++){
      for(int j=0;j<A[0].length;j++){
        System.out.print(
                        A[i][j] + " ");
      }//end inner loop
      System.out.println();
    }//end outer loop

  }//end doArrays()
}// end class
```

3.92

Answer and Explanation (p. 161)

3.7.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 157) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 14: Listing for Question 14.

```
    public class Ap077{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Object X = A;
    Superclass B = A;
    Subclass[] C = (Subclass[])B;
    Subclass[] Y = (Subclass[])X;
    System.out.println(
                   C[0] + " " + Y[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

3.93

Answer and Explanation (p. 160)

3.7.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 158) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 0.0 false 0
- D. None of the above

Listing 15: Listing for Question 15.

```
    public class Ap078{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[1];
    double[] B = new double[1];
    boolean[] C = new boolean[1];
    int[] D = new int[0];

    System.out.println(A[0] + " " +
                       B[0] + " " +
                       C[0] + " " +
                       D.length);

  }//end doArrays()
}// end class
```

3.94

Answer and Explanation (p. 159)

3.7.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.7.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0060: Self-assessment, More on Arrays
- File: Ap0060.htm
- Originally published: 2002
- Published at cnx.org: 12/03/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.7.6 Answers

3.7.6.1 Answer 15

C. 0 0.0 false 0

3.7.6.1.1 Explanation 15

You can initialize array elements

You can create a new array object and initialize its elements using statements similar to the following:

NOTE:

```
    int[] A = {22, 43, 69};
X[] B = {new X(32), new X(21)};
```

What if you don't initialize array elements?

If you create a new array object without initializing its elements, the value of each element in the array is automatically initialized to a default value.

Illustrating array element default initialization

This program illustrates default initialization of **int** , **double** , and **boolean** arrays.

The default values are as follows:

- zero for all numeric values
- false for all **boolean** values
- all zero bits for char values
- null for object references

An array with no elements ...

This program also illustrates that it is possible to have an array object in Java that has no elements. In this case, the value of the **length** property for the array object is 0.

Give me an example

For example, when the user doesn't enter any arguments on the command line for a Java application, the incoming **String** array parameter to the **main** method has a length value of 0.

Another example

It is also possible that methods that return a reference to an array object may sometimes return a reference to an array whose length is 0. The method must satisfy the return type requirement by returning a reference to an array object. Sometimes, there is no data to be used to populate the array, so the method will simply return a reference to an array object with a **length** property value of 0.

Back to Question 15 (p. 158)

3.7.6.2 Answer 14

A. Compiler Error

3.7.6.2.1 Explanation 14

Assigning array reference to type Object

As you learned in an earlier module, you can assign an array object's reference to an ordinary reference variable of the type **Object** . It is not necessary to indicate that the reference variable is a reference to an array by appending square brackets to the type name or the variable name.

Only works with type Object

However, you cannot assign an array object's reference to an ordinary reference variable of any other type. For any type other than **Object** , the reference variable must be declared to hold a reference to an array object by appending empty square brackets onto the type name or the variable name.

The first statement in the following fragment compiles successfully.

NOTE:

```
    Object X = A;
Superclass B = A;
```

However, the second statement in the above fragment produces a compiler error under JDK 1.3, which is partially reproduced below.

NOTE:

```
    Ap077.java:22: incompatible types
found   : Subclass[]
required: Superclass
    Superclass B = A;
```

Both **Superclass** and **Object** are superclasses of the array type referred to by the reference variable named **A** . However, because of the above rule, in order to cause this program to compile successfully, you would need to modify it as shown below by adding the requisite empty square brackets to the **Superclass** type name.

NOTE:

```
        Object X = A;
    Superclass[] B = A;
```

Back to Question 14 (p. 156)

3.7.6.3 Answer 13

NOTE:

```
C.  0 0 0
0 1 2
```

3.7.6.3.1 Explanation 13

Syntactical ugliness

As I indicated in an earlier module, when declaring a reference variable that will refer to an array object, you can place the empty square brackets next to the name of the type or next to the name of the reference variable. In other words, either of the following formats will work.

NOTE:

```
        int[][] A;
    int B[][];
```

What I may not have told you at that time is that you can place some of the empty square brackets in one location and the remainder in the other location.

Really ugly syntax

This is indicated by the following fragment, which declares a reference variable for a two-dimensional array of type **int** . Then it creates the two-dimensional array object and assigns the array object's reference to the reference variable.

NOTE:

```
        int[] A[];
    A = new int[2][3];
```

While it doesn't matter which location you use for the square brackets in the declaration, it does matter how many pairs of square brackets you place in the two locations combined. The number of dimensions on the array *(if you want to think of a Java array as having dimensions)* will equal the total number of pairs of empty square brackets in the declaration of the reference variable. Thus, in this case, the array is a

two-dimensional array because there is one pair of square brackets next to the type and another pair next to the variable name.

This program goes on to use nested for loops to populate the array and then to display the contents of the elements.

I personally don't use this syntax, and I hope that you don't either. However, even if you don't use it, you need to be able to recognize it when used by others.

Back to Question 13 (p. 155)

3.7.6.4 Answer 12

C. 1 2

3.7.6.4.1 Explanation 12

The length property

This program illustrates the use of the array property named **length** , whose value always matches the number of elements in the array.

As a Java programmer, you will frequently call methods that will return a reference to an array object of a specified type, but of an unknown length. *(See, for example, the method named* ***getEventSetDescriptors*** *that is declared in the interface named* ***BeanInfo*** *.)* This program simulates that situation.

Returning a reference to an array

The method named **returnArray** returns a reference to an array of type **int** having two elements. Although I fixed the size of the array in this example, I could just as easily have used a random number to set a different size for the array each time the method is called. Therefore, the **doArrays** method making the call to the method named **returnArray** has no way of knowing the size of the array referred to by the reference that it receives as a return value.

All array objects have a length property

This could be a problem, but Java provides the solution to the problem in the **length** property belonging to all array objects.

The **for** loop in the method named **doArrays** uses the **length** property of the array to determine how many elements it needs to display. This is a very common scenario in Java.

Back to Question 12 (p. 154)

3.7.6.5 Answer 11

A. Compiler Error

3.7.6.5.1 Explanation 11

You cannot cast primitive array references

You cannot cast an array reference from one primitive type to another primitive type, even if the individual elements in the array are of a type that can normally be converted to the new type.

This program attempts to cast a reference to an array of type **int[]** and assign it to a reference variable of type **double** []. Normally, a value of type **int** will be automatically converted to type **double** whenever there is a need for such a conversion. However, this attempted cast produces the following compiler error under JDK 1.3.

NOTE:

```
    Ap074.java:19: inconvertible types
found   : int[]
required: double[]
    double[] B = (double[])A;
```

Why is this cast not allowed?

I can't give you a firm reason why such a cast is not allowed, but I believe that I have a good idea why. I speculate that this is due to the fact that the actual primitive values are physically stored in the array object, and primitive values of different types require different amounts of storage. For example, the type **int** requires 32 bits of storage while the type **double** requires 64 bits of storage.

Would require reconstructing the array object

Therefore, to convert an array object containing **int** values to an array object containing **double** values would require reconstructing the array object and allocating twice as much storage space for each element in the array.

Restriction doesn't apply to arrays of references

As you have seen from previous questions, such a casting restriction does not apply to arrays containing references to objects. This may be because the amount of storage required to store a reference to an object is the same, regardless of the type of the object. Therefore, the allowable casts that you have seen in the previous questions did not require any change to the size of the array. All that changed was some supplemental information regarding the type of objects to which the elements in the array refer.

Back to Question 11 (p. 153)

3.7.6.6 Answer 10

C. 1.0 2.0

3.7.6.6.1 Explanation 10

Assigning array reference to variable of type Object

A reference to an array can be assigned to a non-array reference of the class named **Object**, as in the following statement extracted from the program, where A is a reference to an array object of type **double**.

NOTE:

```
Object B = A;
```

Note that there are no square brackets anywhere in the above statement. Thus, the reference to the array object is not being assigned to an array reference of the type **Object[]**. Rather, it is being assigned to an ordinary reference variable of the type **Object**.

Downcasting to an array type

Once the array reference has been assigned to the ordinary reference variable of the type **Object**, that reference variable can be downcast and used to access the individual elements in the array as illustrated in the following fragment. Note the empty square brackets in the syntax of the cast operator **(double[])**.

NOTE:

```
System.out.println(
       ((double[])B)[0] + " " +
       ((double[])B)[1]);
```

Placement of parentheses is critical

Note also that due to precedence issues, the placement of both sets of parentheses is critical in the above code fragment. You must downcast the reference variable before applying the index to that variable.

Back to Question 10 (p. 153)

3.7.6.7 Answer 9

C. 1 2

3.7.6.7.1 Explanation 9

General array casting rule

The general rule for casting array references *(for arrays whose declared type is the name of a class or an interface)* is:

A reference to an array object can be cast to another array type if the elements of the referenced array are of a type that can be cast to the type of the elements of the specified array type.

Old rules apply here also

Thus, the general rules covering conversion and casting up and down the inheritance hierarchy and among classes that implement the same interfaces also apply to the casting of references to array objects.

A reference to an object can be cast down the inheritance hierarchy to the actual class of the object. Therefore, an array reference can also be cast down the inheritance hierarchy to the declared class for the array object.

This program declares a reference to, creates, and populates an array of the class type **Subclass** . This reference is assigned to an array reference of a type that is a superclass of the actual class type of the array. Then the superclass reference is downcast to the actual class type of the array and assigned to a different reference variable. This third reference variable is used to successfully access and display the contents of the elements in the array.

Back to Question 9 (p. 151)

3.7.6.8 Answer 8

B. Runtime Error

3.7.6.8.1 Explanation 8

Another ClassCastException

While it is allowable to assign an array reference to an array reference variable declared for a class that is further up the inheritance hierarchy *(as illustrated earlier)* , it is not allowable to cast an array reference down the inheritance hierarchy to a subclass of the original declared class for the array.

This program declares a reference for, creates, and populates a two-element array for a class named **Superclass** . Then it downcasts that reference to a subclass of the class named **Superclass** . The compiler is unable to determine that this is a problem. However, the runtime system throws the following exception, which terminates the program at runtime.

NOTE:

```
    java.lang.ClassCastException: [LSuperclass;
 at Worker.doArrays(Ap071.java:19)
 at Ap071.main(Ap071.java:9)
```

Back to Question 8 (p. 149)

3.7.6.9 Answer 7

C. 1 2

3.7.6.9.1 Explanation 7

Assignment to superclass array reference variable

This program illustrates that, if you have a reference to an array object containing references to other objects, you can assign the array object's reference to an array reference variable whose type is a superclass of the declared class of the array object. *(As we will see later, this doesn't work for array objects containing primitive values.)*

What can you do then?

Having made the assignment to the superclass reference variable, whether or not you can do anything useful with the elements in the array *(without downcasting)* depends on many factors.

No downcast required in this case

In this case, the ability to display the contents of the objects referred to in the array was inherited from the class named **Superclass** . Therefore, it is possible to access and display a **String** representation of the objects without downcasting the array object reference from **Superclass** to the actual type of the objects.

Probably need to downcast in most cases

However, that will often not be the case. In most cases, when using a reference of a superclass type, you will probably need to downcast in order to make effective use of the elements in the array object.

Back to Question 7 (p. 146)

3.7.6.10 Answer 6

C. 1 2

3.7.6.10.1 Explanation 6

Straightforward array application

This is a straightforward application of Java array technology for the storage and retrieval of references to objects.

The program declares a reference to, creates, and populates a two-element array of a class named **Subclass** . The class named **Subclass** extends the class named **Superclass** , which in turn, extends the class named **Object** by default.

The super keyword

The class named **Subclass** doesn't do anything particularly useful other than to illustrate extending a class.

However, it also provides a preview of the use of the **super** keyword for the purpose of causing a constructor in a subclass to call a parameterized constructor in its superclass.

Setting the stage for follow-on questions

The main purpose for showing you this program is to set the stage for several programs that will be using this class structure in follow-on questions.

Back to Question 6 (p. 145)

3.7.6.11 Answer 5

B. Runtime Error

3.7.6.11.1 Explanation 5

ClassCastException

There are some situations involving casting where the compiler cannot identify an erroneous condition that is later identified by the runtime system. This is one of those cases.

This program begins with an array of type **double** []. The reference to that array is converted to type **Object** . Then it is cast to type **String** []. All of these operations are allowed by the compiler.

However, at runtime, the runtime system expects to find references to objects of type **String** in the elements of the array. What it finds instead is values of type **double** stored in the elements of the array.

As a result, a **ClassCastException** is thrown. Since it isn't caught and handled by the program, the program terminates with the following error message showing on the screen.

NOTE:

```
    java.lang.ClassCastException: [D
  at Worker.doArrays(Ap068.java:17)
  at Ap068.main(Ap068.java:6)
```

Back to Question 5 (p. 144)

3.7.6.12 Answer 4

C. 1.0 2.0

3.7.6.12.1 Explanation 4

Finally, we got it right

Finally, we managed to get it all together. The program compiles and executes correctly. This program illustrates the assignment of an array object's reference to a reference variable of type **Object** , and the casting of that reference of type **Object** back to the correct array type in order to gain access to the elements in the array.

But don't go away, there is a lot more that you need to know about arrays in Java. We will look at some of those things in the questions that follow.

Back to Question 4 (p. 143)

3.7.6.13 Answer 3

A. Compiler Error

3.7.6.13.1 Explanation 3

Must use the correct cast syntax

While it is possible to store an array object's reference in a reference variable of type **Object** , and later cast it back to an array type to gain access to the elements in the array, you must use the correct syntax in performing the cast. This is not the correct syntax for performing that cast. It is missing the empty square brackets required to indicate a reference to an array object.

A portion of the compiler error produced by JDK 1.3 is shown below:

NOTE:

```
    Ap066.java:17: inconvertible types
found   : java.lang.Object
required: double
    double C = (double)B;
```

Back to Question 3 (p. 142)

3.7.6.14 Answer 2

A. Compiler Error

3.7.6.14.1 Explanation 2

Must cast back to an array type

This program illustrates another very important point. Although you can assign an array object's reference to a reference variable of type **Object** , you cannot gain access to the elements in the array while treating it as type **Object** . Instead, you must cast it back to an array type before you can gain access to the elements in the array object.

A portion of the compiler error produced by JDK 1.3 is shown below:

NOTE:

```
    Ap065.java:18: array required, but java.lang.Object found
B[0] + " " + B[1]);
```

Back to Question 2 (p. 142)

3.7.6.15 Answer 1

C. I'm OK

3.7.6.15.1 Explanation 1

Assigning array reference to type Object

This program illustrates a very important point. You can assign an array object's reference to an ordinary reference variable of type **Object** . Note that I didn't say **Object[]** . The empty square brackets are not required when the type is **Object** .

Standard containers or collections

Later on, when we study the various containers in the Java class libraries *(see the Java Collections Framework)* , we will see that they store references to all objects, including array objects, as type **Object** . Thus, if it were not possible to store a reference to an array object in a reference variable of type **Object** , it would not be possible to use the standard containers to store references to array objects.

Because it is possible to assign an array object's reference to a variable of type **Object** , it is also possible to store array object references in containers of type **Object** .

Back to Question 1 (p. 141)

-end-

3.8 Ap0070: Self-assessment, Method Overloading[12]

3.8.1 Table of Contents

3.8.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 175) to easily find and view the listings while you are reading about them.

[12]This content is available online at <http://cnx.org/content/m45276/1.4/>.

3.8.3 Questions

3.8.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 168) ?

- A. Compiler Error
- B. Runtime Error
- C. 9 17.64
- D. None of the above

Listing 1: Listing for Question 1.

```
    public class Ap079{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 3;
    double y = 4.2;
    System.out.println(square(x) + " "
                         + square(y));
  }//end doOverLoad()

  public int square(int y){
    return y*y;
  }//end square()

  public double square(double y){
    return y*y;
  }//end square()
}// end class
```

3.95

Answer and Explanation (p. 182)

3.8.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 169) ?

- A. Compiler Error
- B. Runtime Error
- C. float 9.0 double 17.64
- D. None of the above

Listing 2: Listing for Question 2.

```
    public class Ap080{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 3;
    double y = 4.2;

    System.out.print(square(x) + " ");
    System.out.print(square(y));
    System.out.println();
  }//end doOverLoad()

  public float square(float y){
    System.out.print("float ");
    return y*y;
  }//end square()

  public double square(double y){
    System.out.print("double ");
    return y*y;
  }//end square()
}// end class
```

3.96

Answer and Explanation (p. 181)

3.8.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 170) ?

- A. Compiler Error
- B. Runtime Error
- C. 10 17.64
- D. None of the above

Listing 3: Listing for Question 3.

```
    public class Ap081{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    double w = 3.2;
    double x = 4.2;

    int y = square(w);
    double z = square(x);

    System.out.println(y + " " + z);
  }//end doOverLoad()

  public int square(double y){
    return (int)(y*y);
  }//end square()

  public double square(double y){
    return y*y;
  }//end square()

}// end class
```

3.97

Answer and Explanation (p. 181)

3.8.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 171) ?

- A. Compiler Error
- B. Runtime Error
- C. 9 17.64
- D. None of the above

Listing 4: Listing for Question 4.

```
    public class Ap083{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int w = 3;
    double x = 4.2;

    System.out.println(
      new Subclass().square(w) + " "
      + new Subclass().square(x));
  }//end doOverLoad()
}// end class

class Superclass{
  public double square(double y){
    return y*y;
  }//end square()
}//end class Superclass

class Subclass extends Superclass{
  public int square(int y){
    return y*y;
  }//end square()
}//end class Subclass
```

3.98

Answer and Explanation (p. 180)

3.8.3.5 Question 5

Which of the following is produced by the program shown in Listing 5 (p. 172) ?

NOTE:

A. Compiler Error

B. Runtime Error

C. float 2.14748365E9
float 9.223372E18

```
        double 4.2

    D.  None of the above
```

Listing 5: Listing for Question 5.

```
    public class Ap084{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 2147483647;
    square(x);
    long y = 9223372036854775807L;
    square(y);
    double z = 4.2;
    square(z);

    System.out.println();
  }//end doOverLoad()

  public void square(float y){
    System.out.println("float" + " " +
                               y + " ");
  }//end square()

  public void square(double y){
    System.out.println("double" + " " +
                               y + " ");
  }//end square()
}// end class
```

3.99

Answer and Explanation (p. 178)

3.8.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 173) ?

- A. Compiler Error
- B. Runtime Error
- C. Test DumIntfc

- D. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap085{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    Test a = new Test();
    DumIntfc b = new Test();
    overLoadMthd(a);
    overLoadMthd(b);
    System.out.println();
  }//end doOverLoad()

  public void overLoadMthd(Test x){
    System.out.print("Test ");
  }//end overLoadMthd

  public void overLoadMthd(DumIntfc x){
    System.out.print("DumIntfc ");
  }//end overLoadMthd
}// end class

interface DumIntfc{
}//end DumIntfc

class Test implements DumIntfc{
}//end class Test
```

3.100

Answer and Explanation (p. 177)

3.8.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 174) ?

- A. Compiler Error
- B. Runtime Error
- C. Test Object
- D. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap086{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    Test a = new Test();
    Object b = new Test();
    overLoadMthd(a);
    overLoadMthd(b);
    System.out.println();
  }//end doOverLoad()

  public void overLoadMthd(Test x){
    System.out.print("Test ");
  }//end overLoadMthd

  public void overLoadMthd(Object x){
    System.out.print("Object ");
  }//end overLoadMthd

}// end class

class Test{
}//end class Test
```

3.101

Answer and Explanation (p. 177)

3.8.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 175) ?

- A. Compiler Error
- B. Runtime Error
- C. SubC SuperC
- D. None of the above

Listing 8: Listing for Question 8.

```
    public class Ap087{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    SubC a = new SubC();
    SuperC b = new SubC();

    SubC obj = new SubC();
    obj.overLoadMthd(a);
    obj.overLoadMthd(b);

    System.out.println();
  }//end doOverLoad()

}// end class

class SuperC{
  public void overLoadMthd(SuperC x){
    System.out.print("SuperC ");
  }//end overLoadMthd
}//end SuperC

class SubC extends SuperC{
  public void overLoadMthd(SubC x){
    System.out.print("SubC ");
  }//end overLoadMthd
}//end class SubC
```

3.102

Answer and Explanation (p. 176)

3.8.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 1 (p. 168) . Listing for Question 1.
- Listing 2 (p. 169) . Listing for Question 2.
- Listing 3 (p. 170) . Listing for Question 3.
- Listing 4 (p. 171) . Listing for Question 4.

- Listing 5 (p. 172) . Listing for Question 5.
- Listing 6 (p. 173) . Listing for Question 6.
- Listing 7 (p. 174) . Listing for Question 7.
- Listing 8 (p. 175) . Listing for Question 8.

3.8.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0070: Self-assessment, Method Overloading
- File: Ap0070.htm
- Originally published: 2002
- Published at cnx.org: 12/04/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.8.6 Answers

3.8.6.1 Answer 8

C. SubC SuperC

3.8.6.1.1 Explanation 8

While admittedly a little convoluted, this is another relatively straightforward application of method overloading using types from the class hierarchy.

Type **SubC** , **SuperC** , or **Object?**

This method defines a class named **SuperC** , which extends **Object** , and a class named **SubC** , which extends **SuperC** . Therefore, an object instantiated from the class named **SubC** can be treated as any of the following types: **SubC** , **SuperC** , or **Object** .

Two overloaded methods in different classes

Two overloaded methods named **overLoadMthd** are defined in two classes in the inheritance hierarchy. The class named **SuperC** defines a version that requires an incoming parameter of type **SuperC** . The class named **SubC** defines a version that requires an incoming parameter of type **SubC** . When called, each of these overloaded methods prints the type of its formal argument.

Two objects of type SubC

The program instantiates two objects of the **SubC** class, storing the reference to one of them in a reference variable of type **SubC**, and storing the reference to the other in a reference variable of type **SuperC**.

Call the overloaded method twice

The next step is to call the overloaded method named **overLoadMthd** twice in succession, passing each of the reference variables of type **SubC** and **SuperC** to the method.

Instance methods require an object

Because the two versions of the overloaded method are instance methods, it is necessary to have an object on which to call the methods. This is accomplished by instantiating a new object of the **SubC** class, storing the reference to that object in a reference variable named **obj**, and calling the overloaded method on that reference.

Overloaded methods not in same class

The important point here is that the two versions of the overloaded method were not defined in the same class. Rather, they were defined in two different classes in the inheritance hierarchy. However, they were defined in such a way that both overloaded versions were contained as instance methods in an object instantiated from the class named **SubC**.

No surprises

There were no surprises. When the overloaded method was called twice in succession, passing the two different reference variables as parameters, the output shows that the version that was called in each case had a formal argument type that matched the type of the parameter that was passed to the method.

Back to Question 8 (p. 174)

3.8.6.2 Answer 7

C. Test Object

3.8.6.2.1 Explanation 7

Another straightforward application

This is another straightforward application of method overloading, which produces no surprises.

This program defines a new class named **Test**, which extends the **Object** class by default. This means that an object instantiated from the class named **Test** can be treated either as type **Test**, or as type **Object**.

The program defines two overloaded methods named **overLoadMthd**. One requires an incoming parameter of type **Test**. The other requires an incoming parameter of type **Object**. When called, each of these methods prints the type of its incoming parameter.

The program instantiates two different objects of the class **Test**, storing a reference to one of them in a reference variable of type **Test**, and storing a reference to the other in a reference variable of type **Object**.

No surprises here

Then it calls the overloaded **overLoadMthd** method twice in succession, passing the reference of type **Test** during the first call, and passing the reference of type **Object** during the second call.

As mentioned above, the output produces no surprises. The output indicates that the method selected for execution during each call is the method with the formal argument type that matches the type of parameter passed to the method.

Back to Question 7 (p. 173)

3.8.6.3 Answer 6

C. Test DumIntfc

3.8.6.3.1 Explanation 6

Overloaded methods with reference parameters

This is a fairly straightforward application of method overloading. However, rather than requiring method parameters of primitive types as in the previous questions in this module, the overloaded methods in this program require incoming parameters of class and interface types respectively.

Type Test or type DumIntfc?

The program defines an interface named **DumIntfc** and defines a class named **Test** that implements that interface. The result is that an object instantiated from the **Test** class can be treated either as type **Test** or as type **DumIntfc** *(it could also be treated as type Object as well)* .

Two overloaded methods

The program defines two overloaded methods named **overLoadMthd** . One requires an incoming parameter of type **Test** , and the other requires an incoming parameter of type **DumIntfc** . When called, each of the overloaded methods prints a message indicating the type of its argument.

Two objects of the class Test

The program instantiates two objects of the class **Test** . It assigns one of the object's references to a reference variable named **a** , which is declared to be of type **Test** .

The program assigns the other object's reference to a reference variable named **b** , which is declared to be of type **DumIntfc** . *(Remember, both objects were instantiated from the class **Test** .)*

No surprises here

Then it calls the overloaded method named **overLoadMthd** twice in succession, passing first the reference variable of type **Test** and then the reference variable of type **DumIntfc** .

The program output doesn't produce any surprises. When the reference variable of type **Test** is passed as a parameter, the overloaded method requiring that type of parameter is selected for execution. When the reference variable of type **DumIntfc** is passed as a parameter, the overloaded method requiring that type of parameter is selected for execution.

Back to Question 6 (p. 172)

3.8.6.4 Answer 5

NOTE:

```
C.  float 2.14748365E9
float 9.223372E18
double 4.2
```

3.8.6.4.1 Explanation 5

Another subtle method selection issue

This program illustrates a subtle issue in the automatic selection of an overloaded method based on assignment compatibility.

This program defines two overloaded methods named **square** . One requires an incoming parameter of type **float** , and the other requires an incoming parameter of type **double** .

When called, each of these methods prints the type of its formal argument along with the value of the incoming parameter as represented by its formal argument type. In other words, the value of the incoming parameter is printed after it has been automatically converted to the formal argument type.

Printout identifies the selected method

This printout makes it possible to determine which version is called for different types of parameters. It also makes it possible to determine the effect of the automatic conversion on the incoming parameter. What we are going to see is that the conversion process can introduce serious accuracy problems.

Call the method three times

The **square** method is called three times in succession, passing values of type **int** , **long** , and **double** during successive calls.

(Type ***long*** *is a 64-bit integer type capable of storing integer values that are much larger than can be stored in type* ***int*** *. The use of this type here is important for illustration of data corruption that occurs through automatic type conversion.)*

The third invocation of the **square** method, passing a **double** as a parameter, is not particularly interesting. There is a version of **square** with a matching argument type, and everything behaves as would be expected for this invocation. The interesting behavior occurs when the **int** and **long** values are passed as parameters.

Passing an int parameter

The first thing to note is the behavior of the program produced by the following code fragment.

NOTE:

```
    int x = 2147483647;
square(x);
```

The above fragment assigns a large integer value (2147483647)to the **int** variable and passes that variable to the **square** method. This fragment produces the following output on the screen:

NOTE:

```
    float 2.14748365E9
```

As you can see, the system selected the overloaded method that requires an incoming parameter of type **float** for execution in this case *(rather than the version that requires type* ***double*** *).*

Conversion from int to float loses accuracy

Correspondingly, it converted the incoming **int** value to type **float** , losing one decimal digit of accuracy in the process. *(The original* ***int*** *value contained ten digits of accuracy. This was approximated by a nine-digit* ***float*** *value with an exponent value of 9.)*

This seems like an unfortunate choice of overloaded method. Selecting the other version that requires a **double** parameter as input would not have resulted in any loss of accuracy.

A more dramatic case

Now, consider an even more dramatic case, as illustrated in the following fragment where a very large **long** integer value(9223372036854775807) is passed to the **square** method.

NOTE:

```
    long y = 9223372036854775807L;
square(y);
```

The above code fragment produced the following output:

NOTE:

```
    float 9.223372E18
```

A very serious loss of accuracy

Again, unfortunately, the system selected the version of the **square** method that requires a **float** parameter for execution. This caused the **long** integer to be converted to a **float** . As a result, the **long** value containing 19 digits of accuracy was converted to an estimate consisting of only seven digits plus an exponent. *(Even if the overloaded* ***square*** *method requiring a* ***double*** *parameter had been*

selected, the conversion process would have lost about three digits of accuracy, but that would have been much better than losing twelve digits of accuracy.)

The moral to the story is ...

Don't assume that just because the system knows how to automatically convert your integer data to floating data, it will protect the integrity of your data. Oftentimes it won't.

To be really safe ...

To be really safe, whenever you need to convert either **int** or **long** types to floating format, you should write your code in such a way as to ensure that it will be converted to type **double** instead of type **float** .

For example, the following modification would solve the problem for the **int** data and would greatly reduce the magnitude of the problem for the **long** data. Note the use of the **(double)** cast to force the **double** version of the **square** method to be selected for execution.

NOTE:

```
    int x = 2147483647;
square((double)x);
long y = 9223372036854775807L;
square((double)y);
```

The above modification would cause the program to produce the following output:

NOTE:

```
    double 2.147483647E9
double 9.223372036854776E18
double 4.2
```

This output shows no loss of accuracy for the **int** value, and the loss of three digits of accuracy for the long value.

*(Because a **long** and a **double** both store their data in 64 bits, it is not possible to convert a very large **long** value to a **double** value without some loss in accuracy, but even that is much better than converting a 64-bit **long** value to a 32-bit **float** value.)*

Back to Question 5 (p. 171)

3.8.6.5 Answer 4

C. 9 17.64

3.8.6.5.1 Explanation 4

When the **square** method is called on an object of the **Subclass** type passing an **int** as a parameter, there is an exact match to the required parameter type of the **square** method defined in that class. Thus, the method is properly selected and executed.

When the **square** method is called on an object of the **Subclass** type passing a **double** as a parameter, the version of the **square** method defined in the **Subclass** type is not selected. The **double** value is not assignment compatible with the required type of the parameter *(an **int** is narrower than a **double**)*.

Having made that determination, the system continues searching for an overloaded method with a required parameter that is either type **double** or assignment compatible with **double** . It finds the version inherited from **Superclass** that requires a **double** parameter and calls it.

The bottom line is, overloaded methods can occur up and down the inheritance hierarchy.

Back to Question 4 (p. 170)

3.8.6.6 Answer 3

A. Compiler Error

3.8.6.6.1 Explanation 3

Return type is not a differentiating feature

This is not a subtle issue. This program illustrates the important fact that the return type does not differentiate between overloaded methods having the same name and formal argument list.

For a method to be overloaded, two or more versions of the method must have the same name and different formal arguments lists.

The return type can be the same, or it can be different *(it can even be void)* . It doesn't matter.

These two methods are not a valid overload

This program attempts to define two methods named **square** , each of which requires a single incoming parameter of type **double** . One of the methods casts its return value to type **int** and returns type **int** . The other method returns type **double** .

The JDK 1.3 compiler produced the following error:

```
NOTE:

    Ap081.java:28: square(double) is already defined
in Worker

  public double square(double y){
```

Back to Question 3 (p. 169)

3.8.6.7 Answer 2

C. float 9.0 double 17.64

3.8.6.7.1 Explanation 2

This program is a little more subtle

Once again, the program defines two overloaded methods named **square** . However, in this case, one of the methods requires a single incoming parameter of type **float** and the other requires a single incoming parameter of type **double** . *(Suffice it to say that the* ***float*** *type is similar to the* ***double*** *type, but with less precision. It is a floating type, not an integer type. The* ***double*** *type is a 64-bit floating type and the* ***float*** *type is a 32-bit floating type.)*

Passing a type int as a parameter

This program does not define a method named **square** that requires an incoming parameter of type **int** . However, the program calls the **square** method passing a value of type **int** as a parameter.

What happens to the int parameter?

The first question to ask is, will this cause one of the two overloaded methods to be called, or will it cause a compiler error? The answer is that it will cause one of the overloaded methods to be called because a value of type **int** is assignment compatible with both type **float** and type **double** .

Which overloaded method will be called?

Since the type **int** is assignment compatible with type **float** and also with type **double** , the next question is, which of the two overloaded methods will be called when a value of type **int** is passed as a parameter?

Learn through experimentation

I placed a print statement in each of the overloaded methods to display the type of that method's argument on the screen when the method is called. By examining the output, we can see that the method

with the **float** parameter was called first *(corresponding to the parameter of type* ***int*** *)*. Then the method with the **double** parameter was called *(corresponding to the parameter of type* ***double*** *)*.

Converted int to float

Thus, the system selected the overloaded method requiring an incoming parameter of type **float** when the method was called passing an **int** as a parameter. The value of type **int** was automatically converted to type **float** .

In this case, it wasn't too important which method was called to process the parameter of type **int** , because the two methods do essentially the same thing – compute and return the **square** of the incoming value.

However, if the behavior of the two methods were different from one another, it could make a lot of difference, which one gets called on an assignment compatible basis. *(Even in this case, it makes some difference. As we will see later, when a very large* ***int*** *value is converted to a* ***float*** *, there is some loss in accuracy. However, when the same very large* ***int*** *value is converted to a* ***double*** *, there is no loss in accuracy.)*

Avoiding the problem

One way to avoid this kind of subtle issue is to avoid passing assignment-compatible values to overloaded methods.

Passing assignment-compatible values to overloaded methods allows the system to resolve the issue through automatic type conversion. Automatic type conversion doesn't always provide the best choice.

Using a cast to force your choice of method

Usually, you can cast the parameter values to a specific type before calling the method and force the system to select your overloaded method of choice.

For example, in this problem, you could force the method with the **double** parameter to handle the parameter of type **int** by using the following cast when the method named **square** is called:

square((double)x)

However, as we will see later, casting may not be the solution in every case.

Back to Question 2 (p. 168)

3.8.6.8 Answer 1

C. 9 17.64

3.8.6.8.1 Explanation 1

What is method overloading?

A rigorous definition of method overloading is very involved and won't be presented here. However, from a practical viewpoint, a method is overloaded when two or more methods having the same name and different formal argument lists are defined in the class from which an object is instantiated, or are inherited into an object by way of superclasses of that class.

How does the compiler select among overloaded methods?

The exact manner in which the system determines which method to call in each particular case is also very involved. Basically, the system determines which of the overloaded methods to execute by matching the types of parameters passed to the method to the types of arguments defined in the formal argument list.

Assignment compatible matching

However, there are a number of subtle issues that arise, particularly when there isn't an exact match. In selecting the version of the method to call, Java supports the concept of an "assignment compatible" match *(or possibly more than one assignment compatible match)* .

Briefly, assignment compatibility means that it would be allowable to assign a value of the type that is passed as a parameter to a variable whose type matches the specified argument in the formal argument list.

Selecting the best match

According to *Java Language Reference* by Mark Grand,

"If more than one method is compatible with the given arguments, the method that most closely matches the given parameters is selected. If the compiler cannot select one of the methods as a better match than the others, the method selection process fails and the compiler issues an error message."

Understanding subtleties

If you plan to be a Java programmer, you must have some understanding of the subtle issues involving overloaded methods, and the relationship between *overloaded* methods and *overridden* methods. Therefore, the programs in this module will provide some of that information and discuss some of the subtle issues that arise.

Even if you don't care about the subtle issues regarding method overloading, many of those issues really involve automatic type conversion. You should study these questions to learn about the problems associated with automatic type conversion.

This program is straightforward

However, there isn't anything subtle about the program for Question 1 (p. 168) . This program defines two overloaded methods named **square** . One requires a single incoming parameter of type **int** . The other requires a single incoming parameter of type **double** . Each method calculates and returns the square of the incoming parameter.

The program calls a method named **square** twice in succession, and displays the values returned by those two invocations. In the first case, an **int** value is passed as a parameter. This causes the method with the formal argument list of type **int** to be called.

In the second case, a **double** value is passed as a parameter. This causes the method with the formal argument list of type **double** to be called.

Overloaded methods may have different return types

Note in particular that the overloaded methods have different return types. One method returns its value as type **int** and the other returns its value as type **double** . This is reflected in the output format for the two return vales as shown below:

9 17.64

Back to Question 1 (p. 168)

-end-

3.9 Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods[13]

3.9.1 Table of Contents

3.9.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

[13] This content is available online at <http://cnx.org/content/m45279/1.4/>.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 193) to easily find and view the listings while you are reading about them.

3.9.3 Questions

3.9.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 184) ?

- A. Compiler Error
- B. Runtime Error
- C. An Object
- D. None of the above

Listing 1: Listing for Question 1.

```
    public class Ap090{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = NewClass();
    System.out.println(obj);

  }//end makeObj()


}// end class

class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

3.103

Answer and Explanation (p. 202)

3.9.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 185) ?

- A. Compiler Error
- B. Runtime Error
- C. An Object
- D. None of the above

Listing 2: Listing for Question 2.

```
    public class Ap091{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);

  }//end makeObj()

}// end class

Class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

3.104

Answer and Explanation (p. 201)

3.9.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 186) ?

- A. Compiler Error
- B. Runtime Error
- C. An Object
- D. None of the above

Listing 3: Listing for Question 3.

```
    public class Ap092{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);

  }//end makeObj()

}// end class

class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

3.105

Answer and Explanation (p. 200)

3.9.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 187) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 2
- D. None of the above

Listing 4: Listing for Question 4.

```
    public class Ap093{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()
}//end NewClass
```

3.106

Answer and Explanation (p. 198)

3.9.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 188) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 2
- D. None of the above

Listing 5: Listing for Question 5.

```
    public class Ap094{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class Superclass{
  private int x;

  public Superclass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()

  public void setX(int x){
    this.x = x;
  }//end setX()
}//end Superclass

class Subclass extends Superclass{
  public Subclass(){
    setX(2);
  }//end noarg constructor
}//end Subclass
```

3.107

Answer and Explanation (p. 198)

3.9.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 189) ?

- A. Compiler Error
- B. Runtime Error

- C. Object containing 5
- D. Object containing 2
- E. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap095{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()
}//end NewClass
```

3.108

Answer and Explanation (p. 197)

3.9.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 190) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 0, 0.0, false
- D. Object containing 0.0, 0, true

- E. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap096{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x;
  private double y;
  private boolean z;

  public String toString(){
    return "Object containing " +
                          x + ", " +
                          y + ", " + z;
  }//end toString()
}//end NewClass
```

Answer and Explanation (p. 197)

3.9.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 191) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 5
- E. None of the above

Listing 8: Listing for Question 8.

```
    public class Ap097{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    System.out.println(obj.getX());
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end NewClass
```

3.110

Answer and Explanation (p. 195)

3.9.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 192) ?

- A. Compiler Error
- B. Runtime Error
- C. 10
- D. None of the above

Listing 9: Listing for Question 9.

```
    public class Ap098{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){

    NewClass obj = new NewClass();
    obj.setX(10);
    System.out.println(obj.getX());

  }//end makeObj()
}// end class

class NewClass{
  private int y;

  public void setX(int y){
    this.y = y;
  }//end setX()

  public int getX(){
    return y;
  }//end getX()
}//end NewClass
```

3.111

Answer and Explanation (p. 195)

3.9.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 193) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 5
- E. 10
- F. None of the above

Listing 10: Listing for Question 10.

```
    public class Ap099{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    obj.x = 10;
    System.out.println(obj.x);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public void setX(int x){
    this.x = x;
  }//end setX()

  public int getX(){
    return x;
  }//end getX()
}//end NewClass
```

3.112

Answer and Explanation (p. 194)

3.9.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 4 (p. 187) . Listing for Question 4.
- Listing 5 (p. 188) . Listing for Question 5.
- Listing 6 (p. 189) . Listing for Question 6.
- Listing 7 (p. 190) . Listing for Question 7.
- Listing 8 (p. 191) . Listing for Question 8.
- Listing 9 (p. 192) . Listing for Question 9.
- Listing 10 (p. 193) . Listing for Question 10.

3.9.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods
- File: Ap0080.htm
- Originally published: 2002
- Published at cnx.org: 12/05/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.9.6 Answers

3.9.6.1 Answer 10

A. Compiler Error

3.9.6.1.1 Explanation 10

Variables declared private really are private

The code in the following fragment attempts to ignore the setter and getter methods and directly access the **private** instance variable named **x** in the object referred to by the reference variable named **obj** .

NOTE:

```
    obj.x = 10;
System.out.println(obj.x);
```

This produces a compiler error. The compiler error produced by JDK 1.3 is reproduced below.

NOTE:

```
    Ap099.java:19: x has private access in
NewClass
    obj.x = 10;
       ^
Ap099.java:20: x has private access in
NewClass
    System.out.println(obj.x);
                       ^
```

Back to Question 10 (p. 192)

3.9.6.2 Answer 9

C. 10

3.9.6.2.1 Explanation 9

A setter and a getter

This is a very simple program that uses a setter *(modifier or mutator)* method named **setX** to set the value 10 in a property named **x** that is stored in an instance variable named **y** in an object instantiated from the class named **NewClass** ..

The program also uses a getter *(accessor)* method named **getX** to get and display the value of the property named **x** . *(Note that according to JavaBeans design patterns, the name of the property is unrelated to the name of variable in which the property value is stored.)*

Back to Question 9 (p. 191)

3.9.6.3 Answer 8

D. 5

3.9.6.3.1 Explanation 8

Hide your data and expose your methods

For reasons that I won't go into here, good object-oriented design principles state that in almost all cases where an instance variable is not declared to be **final** , it should be declared **private** . *(A final variable behaves like a constant.)*

What is private access?

When an instance variable is declared **private** , it is accessible only by methods of the class in which it is defined. Therefore, the only way that the "outside world" can gain access to a **private** instance variable is by going through an *(usually **public**)* instance method of the object.

Accessor, modifier, mutator, setter, and getter methods

Historically, methods that have been defined for the purpose of exposing **private** instance variables to the outside world have been referred to as *accessor* and *modifier* methods. *(Modifier methods are also sometimes called mutator methods.)*

(Note that since the advent of Sun's JavaBeans Component design patterns, these methods have also come to be known as getter methods and setter methods in deference to the design-pattern naming conventions for the methods.)

A private instance variable with an initializer

The class named **NewClass** declares a **private** instance variable named **x** and initializes its value to 2, as shown in the following code fragment:

NOTE:

```
private int x = 2;
```

Two constructors

The class contains both a *noarg* constructor and a *parameterized* constructor as shown in the following fragment:

NOTE:

```
public NewClass(){
}//end constructor

public NewClass(int x){
  this.x = x;
}//end constructor
```

Calling the noarg constructor

If an object of the class is instantiated by calling the *noarg* constructor, the initial value of 2 remains intact, and that object contains an instance variable with an initial value of 2.

Calling the parameterized constructor

If an object of the class is instantiated by calling the parameterized constructor, the initial value of 2 is overwritten by the value of the incoming parameter to the parameterized constructor. In this case, that value is 5, because the object is instantiated by the following code fragment that passes the literal value 5 to the parameterized constructor. Thus, the initial value of the instance variable in that object is 5.

NOTE:

```
NewClass obj = new NewClass(5);
```

A getter method

Because the instance variable named **x** is **private**, it cannot be accessed directly for display by the code in the **makeObj** method of the **Worker** class. However, the **NewClass** class provides the following public *getter* or *accessor* method that can be used to get the value stored in the instance variable.

(The name of this method complies with JavaBeans design patterns. If you examine the name carefully, you will see why Java programmers often refer to methods like this as getter methods.)

NOTE:

```
public int getX(){
  return x;
}//end getX()
```

Calling the getter method

Finally, the second statement in the following code fragment calls the getter method on the **NewClass** object to get and display the value of the instance variable named **x**.

NOTE:

```
NewClass obj = new NewClass(5);
System.out.println(obj.getX());
```

Back to Question 8 (p. 190)

3.9.6.4 Answer 7

C. Object containing 0, 0.0, false

3.9.6.4.1 Explanation 7

Default initialization values

The purpose of this question is to confirm that you understand the default initialization of instance variables in an object when you don't write code to cause the initialization of the instance variable to differ from the default.

By default, all instance variables in a new object are initialized with default values if you don't provide a constructor *(or other mechanism)* that causes them to be initialized differently from the default.

- All instance variables of the numeric types are initialized to the value of zero for the type. This program illustrates default initialization to zero for **int** and **double** types.
- Instance variables of type **boolean** are initialized to false.
- Instance variables of type **char** are initialized to a 16-bit Unicode character for which all sixteen bits have been set to zero. I didn't include initialization of the **char** type in the output of this program because the **default** char value is not printable.
- Instance variables of reference types are initialized to null.

Back to Question 7 (p. 189)

3.9.6.5 Answer 6

C. Object containing 5

3.9.6.5.1 Explanation 6

A parameterized constructor

This program illustrates the straightforward use of a parameterized constructor.

The class named **NewClass** defines a parameterized constructor that requires an incoming parameter of type **int** .

(For good design practice, the class also defines a noarg constructor, even though it isn't actually used in this program. This makes it available if needed later when someone extends the class.)

Both constructors are shown in the following code fragment.

NOTE:

```
    public NewClass(){
}//end constructor

public NewClass(int x){
  this.x = x;
}//end constructor
```

The parameterized constructor stores its incoming parameter named **x** in an instance variable of the class, also named **x** .

*(The use of the keyword **this** is required in this case to eliminate the ambiguity of having a local parameter with the same name as an instance variable. This is very common Java programming style that you should recognize and understand.)*

Call the parameterized constructor

The following code fragment calls the parameterized constructor, passing the literal **int** value of 5 as a parameter.

NOTE:

```
NewClass obj = new NewClass(5);
```

Hopefully you will have no difficulty understanding the remaining code in the program that causes the value stored in the instance variable named **x** to be displayed on the computer screen.

Back to Question 6 (p. 188)

3.9.6.6 Answer 5

A. Compiler Error

3.9.6.6.1 Explanation 5

If you define any constructors, ...

The discussion for Question 4 (p. 186) explained that if you define any constructor in a new class, you must define all constructors that will ever be needed for that class. When you define one or more constructors, the default *noarg* constructor is no longer provided by the system on your behalf.

Question 4 (p. 186) illustrated a simple manifestation of a problem arising from the failure to define a *noarg* constructor that would be needed later. The reason that it was needed later was that the programmer attempted to explicitly use the non-existent *noarg* constructor to create an instance of the class.

A more subtle problem

The problem in this program is more subtle. Unless you *(or the programmer of the superclasses)* specifically write code to cause the system to behave otherwise, each time you instantiate an object of a class, the system automatically calls the *noarg* constructor on superclasses of that class up to and including the class named **Object** . If one or more of those superclasses don't have a *noarg* constructor, unless the author of the subclass constructor has taken this into account, the program will fail to compile.

Calling a non-existing *noarg* constructor

This program attempts to instantiate an object of a class named **Subclass** , which extends a class named **Superclass** . By default, when attempting to instantiate the object, the system will attempt to call a *noarg* constructor defined in **Superclass** .

Superclass has no *noarg* constructor

The **Superclass** class defines a parameterized constructor that requires a single incoming parameter of type **int** . However, it does not also define a *noarg* constructor. Because the parameterized constructor is defined, the default *noarg* constructor does not exist. As a result, JDK 1.3 produces the following compiler error:

NOTE:

```
    Ap094.java:40: cannot resolve symbol

symbol  : constructor Superclass  ()
location: class Superclass
  public Subclass(){
```

Back to Question 5 (p. 187)

3.9.6.7 Answer 4

A. Compiler Error

3.9.6.7.1 Explanation 4

Constructors

Java uses the following kinds of constructors:

- Those that take arguments, often referred to as *parameterized constructors* , which typically perform initialization on the new object using parameter values.
- Those that don't take arguments, often referred to as *default* or *noarg* constructors, which perform default initialization on the new object.
- Those that don't take arguments but perform initialization on the new object in ways that differ from the default initialization.

Constructor definition is optional

You are not required to define a constructor when you define a new class. If you don't define a constructor for your new class, a default constructor will be provided on your behalf. This constructor requires no argument, and it is typically used in conjunction with the new operator to create an instance of the class using statements such as the following:

NOTE:

```
NewClass obj = new NewClass();
```

The default constructor

The default constructor typically does the following:

- Calls the *noarg* constructor of the superclass
- Assists in the process of allocating and organizing memory for the new object
- Initializes all instance variables of the new object with the following four default values:
  - numeric = 0,
  - boolean = false,
  - char = all zero bits
  - reference = null

Are you satisfied with default values?

As long as you are satisfied with the default initialization of all instance variables belonging to the object, there is no need for you to define a constructor of your own.

However, in the event that you have initialization needs that are not satisfied by the default constructor, you can define your own constructor. Your new constructor may or may not require arguments. *(In case you have forgotten, the name of the constructor is always the same of the name of the class in which it is defined.)*

A non-default *noarg* constructor

If your new constructor doesn't require arguments, you may need to write code that performs initialization in ways that differ from the default initialization. For example, you might decide that a particular **double** instance variable needs to be initialized with a random number each time a new object is instantiated. You could do that with a constructor of your own design that doesn't take arguments by defining the constructor to get initialization values from an object of the **Random** class.

A parameterized constructor

If your new constructor does take arguments, *(a parameterized constructor)* you can define as many overloaded versions as you need. Each overloaded version must have a formal argument list that differs from the formal argument list of all of the other overloaded constructors for that class.

(The rules governing the argument list for overloaded constructors are similar to the rules governing the argument list for overloaded methods, which were discussed in a previous module.)

Use parameter values for initialization

In this case, you will typically define your parameterized constructors to initialize some or all of the instance variables of the new object using values passed to the constructor as parameters.

What else can a constructor do?

You can also cause your new constructor to do other things if you so choose. For example, if you know how to do so, you could cause your constructor *(with or without parameters)* to play an audio clip each time a new object is instantiated. You could use a parameter to determine which audio clip to play in each particular instance.

The punch line

So far, everything that I have said is background information for this program. Here is the punch line insofar as this program is concerned.

If you define any constructor in your new class, you must define all constructors that your new class will ever need.

If you define any constructor, the default constructor is no longer provided on your behalf. If your new class needs a *noarg* constructor *(and it probably does, but that may not become apparent until later when you or someone else extends your class)* you must define the *noarg* version in addition to the other overloaded versions that you define.

A violation of the rule

This program violated the rule given above. It defined the parameterized constructor for the class named **NewClass** shown below

NOTE:

```
  public NewClass(int x){
this.x = x;
}//end constructor
```

However, the program did not also define a *noarg* constructor for the **NewClass** class.

Calling the *noarg* constructor

The code in the **makeObj** method of the **Worker** class attempted to instantiate a new object using the following code:

NOTE:

```
NewClass obj = new NewClass();
```

Since the class definition didn't contain a definition for a *noarg* constructor, the following compiler error was produced by JDK 1.3.

NOTE:

```
    Ap093.java:18: cannot resolve symbol
symbol  : constructor NewClass
()
location: class NewClass
    NewClass obj = new NewClass();
```

Back to Question 4 (p. 186)

3.9.6.8 Answer 3

C. An Object

3.9.6.8.1 Explanation 3

We finally got it right!

Did you identify the errors in the previous two programs before looking at the answers?

This program declares the class named **NewClass** correctly and uses the **new** operator correctly in conjunction with the default *noarg* constructor for the **NewClass** class to create a new instance of the class *(an object)* .

Making the class public

One of the things that I could do differently would be to make the declaration for the **NewClass** class public *(as shown in the following code fragment)* .

NOTE:

```
  public class NewClass{
public String toString(){
  return "An Object";
}//end toString()
}//end NewClass
```

I am a little lazy

The reason that I didn't declare this class **public** *(and haven't done so throughout this series of modules)* is because the source code for all **public** classes and interfaces must be in separate files. While that is probably a good requirement for large programming projects, it is overkill for simple little programs like I am presenting in this group of self-assessment modules.

Dealing with multiple files

Therefore, in order to avoid the hassle of having to deal with multiple source code files for every program, I have been using *package-private* access for class definitions other than the controlling class *(the controlling class is declared public)* . Although I won't get into the details at this point, when a class is not declared public, it is common to say that it has *package-private* access instead of **public** access.

Back to Question 3 (p. 185)

3.9.6.9 Answer 2

A. Compiler Error

3.9.6.9.1 Explanation 2

Java is a case-sensitive language

Java keywords must be written exactly as specified. The keyword **class** cannot be written as *Class* , which is the problem with this program.

The inappropriate use of the upper-case C in the word *Class* caused the following compiler error.

NOTE:

```
Ap091.java:25: 'class' or 'interface' expected


Class NewClass{
```

The solution to the problem

This problem can be resolved by causing the first character in the keyword **class** to be a lower-case character as shown in the following code fragment.

NOTE:

```
    class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

Back to Question 2 (p. 184)

3.9.6.10 Answer 1

A. Compiler Error

3.9.6.10.1 Explanation 1

Instantiating an object

There are several ways to instantiate an object in Java:

- Use the **newInstance** method of the class named **Class** .
- Reconstruct a serialized object using an I/O **readObject** method.
- Create an initialized array object such as {1,2,3}.
- Create a **String** object from a literal string such as "A String".
- Use the **new** operator.

Of all of these, the last two are by far the most common.

What you cannot do!

You cannot instantiate a new object using code like the following code fragment that was extracted from this program.

NOTE:

```
    NewClass obj = NewClass();
```

This program produces the following compiler error:

NOTE:

```
    Ap090.java:18: cannot resolve symbol
symbol  : method NewClass  ()
location: class Worker
    NewClass obj = NewClass();
```

The solution to the problem

This problem can be solved by inserting the **new** operator to the left of the constructor as shown in the following code fragment.

NOTE:

```
        NewClass obj = new NewClass();
```

Back to Question 1 (p. 184)

-end-

3.10 Ap0090: Self-assessment, the super keyword, final keyword, and static methods[14]

3.10.1 Table of Contents

3.10.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 214) to easily find and view the listings while you are reading about them.

3.10.3 Questions

3.10.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 204) ?

- A. Compiler Error
- B. Runtime Error
- C. 1, 2
- D. 5, 10
- E. None of the above

[14] This content is available online at <http://cnx.org/content/m45270/1.4/>.

Listing 1: Listing for Question 1.

```
    public class Ap100{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj.getX() +
                    ", " + obj.getY());
  }//end makeObj()
}// end class

class Superclass{
  private int x = 1;

  public Superclass(){
    x = 5;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end Superclass

class Subclass extends Superclass{
  private int y = 2;

  public Subclass(){
    super();
    y = 10;
  }//end constructor

  public int getY(){
    return y;
  }//end getY()
}//end Subclass
```

3.113

Answer and Explanation (p. 222)

3.10.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 206) ?

- A. Compiler Error
- B. Runtime Error
- C. 1, 2
- D. 5, 2
- E. 5, 10
- F. 20, 10
- G. None of the above

Listing 2: Listing for Question 2.

```
    public class Ap101{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj.getX() +
                    ", " + obj.getY());
  }//end makeObj()
}// end class

class Superclass{
  private int x = 1;

  public Superclass(){
    x = 5;
  }//end constructor

  public Superclass(int x){
    this.x = x;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end Superclass

class Subclass extends Superclass{
  private int y = 2;

  public Subclass(){
    super(20);
    y = 10;
  }//end constructor

  public int getY(){
    return y;
  }//end getY()
}//end Subclass
```

3.114

Answer and Explanation (p. 221)

3.10.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 207) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 3: Listing for Question 3.

```
    public class Ap102{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  public void finalStuff(){
    final int x = 5;
    x = 10;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

3.115

Answer and Explanation (p. 220)

3.10.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 208) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 4: Listing for Question 4.

```
    public class Ap103{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  public void finalStuff(){
    public final int x = 5;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

3.116

Answer and Explanation (p. 219)

3.10.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 209) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 5: Listing for Question 5.

```
    public class Ap104{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  void finalStuff(){
    final int x = 5;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

3.117

Answer and Explanation (p. 218)

3.10.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 210) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.141592653589793
- D. 3.1415927
- E. None of the above

Listing 6: Listing for Question 6.

```
    public class Ap105{
  public static void main(
                       String args[]){
    System.out.println(Worker.fPi);
  }//end main()
}//end class definition

class Worker{
  public static final float fPi =
                       (float)Math.PI;
}// end class
```

3.118

Answer and Explanation (p. 218)

3.10.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 211) ?

- A. Compiler Error
- B. Runtime Error
- C. A static method
- D. None of the above

Listing 7: Listing for Question 7.

```
    public class Ap106{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class definition

class Worker{
  public static void staticMethod(){
    System.out.println(
                    "A static method");
  }//end staticMethod()

}// end class
```

3.119

Answer and Explanation (p. 217)

3.10.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 212) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 8: Listing for Question 8.

```
    public class Ap107{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class Ap107

class Worker{
  private int x = 5;
  public static void staticMethod(){
    System.out.println(x);
  }//end staticMethod()

}// end class
```

3.120

Answer and Explanation (p. 217)

3.10.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 213) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 9: Listing for Question 9.

```
    public class Ap108{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class Ap108

class Worker{
  private int x = 5;
  public static void staticMethod(){
    System.out.println(
                  new Worker().getX());
  }//end staticMethod()

  public int getX(){
    return x;
  }//end getX()

}// end class
```

3.121

Answer and Explanation (p. 216)

3.10.3.10 Question 10

Which output shown below is produced by the program shown in Listing 10 (p. 214) ?

NOTE:

A. Compiler Error
B. Runtime Error
C. 38.48451000647496
12.566370614359172
D. None of the above

Listing 10: Listing for Question 10.

```
    public class Ap109{
  public static void main(String args[]){
    System.out.println(Worker.area(3.5));
    System.out.println(Worker.area(2.0));
    System.out.println();
  }//end main()
}//end class Ap109

class Worker{
  public static double area(double r){
    return r*r*Math.PI;
  }//end area()

}// end class
```

3.122

Answer and Explanation (p. 215)

3.10.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.10.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0090: Self-assessment, the super keyword, final keyword, and static methods

- File: Ap0090.htm
- Originally published: 2002
- Published at cnx.org: 12/05/12

- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.10.6 Answers

3.10.6.1 Answer 10

NOTE:

```
C.  38.48451000647496
12.566370614359172
```

3.10.6.1.1 Explanation 10

Use static methods sparingly

Good object-oriented design dictates that **static** methods be used sparingly, and only in those situations where they are appropriate. As you might guess, not all authors will agree on the issue of appropriateness in all cases.

Is this an appropriate use of a static method?

However, I believe that most authors will agree that this program illustrates an appropriate use of a **static** method.

No persistence requirement

This **static** method computes and returns a result on a non-persistent basis. That is to say, there is no attempt by the **static** method to save any historical information from one call of the method to the next. *(Of course, the method that calls the* ***static*** *method can save whatever it chooses to save.)*

Avoiding wasted computer resources

In situations such as this, it would often be a waste of computer resources to require a program to instantiate an object and call an instance method on that object just to be able to delegate a non-persistent computation to that method. *(This is just about as close to a global method as you can get in Java.)*

Computing the area of a circle

In this program, the **Worker** class provides a **static** method named **area** that receives a **double** parameter representing the radius of a circle. It computes and returns the area of the circle as a **double** value. The **static** method named area is shown in the following code fragment.

NOTE:

```
  class Worker{
public static double area(double r){
  return r*r*Math.PI;
}//end area()

}// end class
```

As a driver, the **main** method of the controlling class calls the **area** method twice in succession, passing different values for the radius of a circle. In each case, the **main** method receives and displays the value that is returned by the **area** method representing the area of a circle.

Static methods in the class libraries

If you examine the Java API documentation carefully, you will find numerous examples of **static** methods that produce and return something on a non-persistent basis. *(Again, non-persistent in this context means that no attempt is made by the* ***static*** *method to store any historical information. It does a job, forgets it, and goes on to the next job when it is called again.)*

Factory methods

For example, the alphabetical index of the JDK 1.3 API lists several dozen **static** methods named **getInstance** , which are defined in different classes. These methods, which usually produce and return a reference to an object, are often called *factory methods* .

Here is the text from the API documentation describing one of them:

NOTE: getInstance(int)

Static method in class java.awt.AlphaComposite

Creates an AlphaComposite object with the specified rule.

Back to Question 10 (p. 213)

3.10.6.2 Answer 9

C. 5

3.10.6.2.1 Explanation 9

Going through a reference to ...

This program illustrates a rather convoluted methodology by which a **static** method can gain access to an instance member of an object.

NOTE:

```
  class Worker{
private int x = 5;
public static void staticMethod(){
  System.out.println(
                new Worker().getX());
}//end staticMethod()

public int getX(){
  return x;
}//end getX()

}// end class
```

In this example, the **static** method calls a getter method on a reference to an object to gain access to an instance variable belonging to that object. This is what I meant in the discussion in the previous question when I said *"going through a reference to an object of the class."*

Back to Question 9 (p. 212)

3.10.6.3 Answer 8

A. Compiler Error

3.10.6.3.1 Explanation 8

A static method cannot access ...

A **static** method cannot access non-static or instance members of its class without going through a reference to an object of the class.

In this program, the **static** method attempts to directly access the instance variable named **x** . As a result, JDK 1.3 produces the following compiler error:

NOTE:

```
    Ap107.java:17: non-static variable x
cannot be referenced from a static context

    System.out.println(x);
```

Back to Question 8 (p. 211)

3.10.6.4 Answer 7

C. A static method

3.10.6.4.1 Explanation 7

Using a static method

This is a very straightforward example of the use of a **static** method.

When a method is declared **static** , it is not necessary to instantiate an object of the class containing the method in order to access the method *(although it is possible to do so unless the class is declared abstract)* . All that is necessary to access a **public** **static** method is to refer to the name of the class in which it is defined and the name of the method joined by a period.

(A method that is declared ***static*** *is commonly referred to as a class method. If the method is not declared* ***public*** *, it may not be accessible from your code.)*

Accessing the static method

This is illustrated by the following fragment from the program, with much of the code deleted for brevity.

NOTE:

```
    //...
        Worker.staticMethod();
  //...

class Worker{
  public static void staticMethod(){
    //...
  }//end staticMethod()

}// end class
```

The class named **Worker** defines a **public** **static** method named **staticMethod** . A statement in the **main** method of the controlling class calls the method by referring to the name of the class and the name of the method joined by a period.

When should you use static methods?

Static methods are very useful as utility methods *(getting the absolute value of a number, for example)* .

In my opinion, you should almost never use a **static** method in any circumstance that requires the storage and use of data from one call of the method to the next. In other words, a **static** method may be appropriate for use when it performs a specific task that is completed each time it is called without the requirement for data to persist between calls.

The **Math** class contains many good examples of the use of **static** methods, such as **abs** , **acos** , **asin** , etc.

Back to Question 7 (p. 210)

3.10.6.5 Answer 6

D. 3.1415927

3.10.6.5.1 Explanation 6

Using a public static final member variable

The class named **Worker** declares and initializes a member variable named **fPi** .

final

Because it is declared **final** , it is not possible to write code that will change its value after it has been initialized.

static

Because it is declared **static** , it can be accessed without a requirement to instantiate an object of the **Worker** class. All that is necessary to access the variable is to refer to the name of the class and the name of the variable joined by a period.

Because it is **static** , it can also be accessed by **static** methods.

public

Because it is declared **public** , it can be accessed by any code in any method in any object that can locate the class.

Type float is less precise than type double

Because the initialized value is cast from the type **double** that is returned by **Math.PI** to type **float** , an 8-digit approximation is stored in the variable named **fPi** .

The **double** value returned by **Math.PI** is 3.141592653589793

The cast to type **float** reduces the precision down to 3.1415927

Back to Question 6 (p. 209)

3.10.6.6 Answer 5

C. 5

3.10.6.6.1 Explanation 5

Using a final local variable

Well, I finally got rid of all the bugs. This program uses a **final** local variable properly. The program compiles and executes without any problems.

Back to Question 5 (p. 208)

3.10.6.7 Answer 4

A. Compiler Error

3.10.6.7.1 Explanation 4

The purpose of this question is to see if you are still awake.

What caused the compiler error?

The statement that caused the compiler error in this program is shown below. Now that you know that there was a compiler error, and you know which statement caused it, do you know what caused it?

NOTE:

```
public final int x = 5;
```

Using public static final member variables

As I mentioned in an earlier question, the **final** keyword can be applied either to local variables or to member variables. When applying the **final** keyword to member variables, it is common practice to declare them to be both **public** and **static** in order to make them as accessible as possible. For example, the math class has a **final** variable that is described as follows:

NOTE: public static final double PI

The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

The constant named PI

You may recognize the constant named **PI** from your high school geometry class.

Whenever you need the value for the constant **PI**, you shouldn't have to instantiate an object just to get access to it. Furthermore, your class should not be required to have any special package relationship with the **Math** class just to get access to **PI**.

The good news ...

Because **PI** is declared to be both **public** and **static** in the **Math** class, it is readily available to any code in any method in any Java program that has access to the standard Java class library.

How is PI accessed?

PI can be accessed by using an expression as simple as that shown below, which consists simply of the name of the class and the name of the variable joined by a period *(Math.PI)*.

NOTE:

```
double piRSquare = Math.PI * R * R;
```

No notion of public local variables

As a result of the above, many of you may have become accustomed to associating the keyword **public** with the keyword **final**. However, if you missed this question and you have read the explanation to this point, you must also remember that there is no notion of **public** or **private** for local variables. Therefore, when this program was compiled under JDK 1.3, a compiler error was produced. That compiler error is partially reproduced below:

NOTE:

```
Ap103.java:16: illegal start of
expression
public final int x = 5;
```

Back to Question 4 (p. 207)

3.10.6.8 Answer 3

A. Compiler Error

3.10.6.8.1 Explanation 3

The **final** keyword

The **final** keyword can be applied in a variety of ways in Java. This includes:

- final parameters
- final methods
- final classes
- final variables *(constants)*

Behaves like a constant

When the **final** keyword is applied to a variable in Java, that causes the variable to behave like a constant. In other words, the value of the variable must be initialized when it is declared, and it cannot be changed thereafter *(see the exception discussed below)* .

Apply to local or member variables

The **final** keyword can be applied to either local variables or member variables. *(In case you have forgotten, local variables are declared inside a method or constructor, while member variables are declared inside a class, but outside a method.)*

So, what is the problem?

The problem with this program is straightforward. As shown in the following code fragment, after declaring a **final** local variable and initializing its value to 5, the program attempts to change the value stored in that variable to 10. This is not allowed.

NOTE:

```
        final int x = 5;
    x = 10;
```

A compiler error

JDK 1.3 produces the following error message:

NOTE:

```
    Ap102.java:17: cannot assign a value to
final
variable x


        x = 10;
```

An interesting twist - blank finals

An interesting twist of the use of the **final** keyword with local variables is discussed below.

Background information

Regardless of whether or not the local variable is declared **final** , the compiler will not allow you to access the value in a local variable if that variable doesn't contain a value. This means that you must always either initialize a local variable or assign a value to it before you can access it.

So, what is the twist?

Unlike **final** member variables of a class, the Java compiler and runtime system do not require you to initialize a **final** local variable when you declare it. Rather, you can wait and assign a value to it later.

(Some authors refer to this as a blank final.) However, once you have assigned a value to a **final** local variable, you cannot change that value later.

The bottom line

Whether you initialize the **final** local variable when you declare it, or assign a value to it later, the result is the same. It behaves as a constant. The difference is that if you don't initialize it when you declare it, you cannot access it until after you assign a value to it.

Back to Question 3 (p. 207)

3.10.6.9 Answer 2

F. 20, 10

3.10.6.9.1 Explanation 2

Calling a parameterized constructor

This is a relatively straightforward implementation of the use of the **super** keyword in a subclass constructor to call a parameterized constructor in the superclass.

The interesting code in the program is highlighted in the following fragment. Note that quite a lot of code was deleted from the fragment for brevity.

NOTE:

```
    class Superclass{
  //...

  public Superclass(int x){
    //...
  }//end constructor

  //...
}//end Superclass

class Subclass extends Superclass{
  //...

  public Subclass(){
    super(20);
    //...
  }//end constructor

  //...
}//end Subclass
```

Using the super keyword

The code that is of interest is the use of **super(20)** as the first executable statement in the **Subclass** constructor to call the parameterized constructor in the superclass, passing a value of 20 as a parameter to the parameterized constructor.

Note that when the **super** keyword is used in this fashion in a constructor, it must be the ***first*** executable statement in the constructor.

As before, the program plays around a little with initial values for instance variables to see if you are alert, but the code that is really of interest is highlighted in the above fragment.

Back to Question 2 (p. 205)

3.10.6.10 Answer 1

D. 5, 10

3.10.6.10.1 Explanation 1

The execution of constructors

The purpose of this question and the associated answer is to illustrate explicitly what happens automatically by default regarding the execution of constructors.

The Subclass constructor

This program defines a class named **Subclass** , which extends a class named **Superclass** . A portion of the **Subclass** definition, including its *noarg* constructor is shown in the following code fragment. *(The class also defines a getter method, which was omitted here for brevity.)*

NOTE:

```
  class Subclass extends Superclass{
private int y = 2;

public Subclass(){
  super();
  y = 10;
}//end constructor

//...
}//end Subclass
```

The super keyword

The important thing to note in the above fragment is the statement containing the keyword **super** .

The **super** keyword has several uses in Java. As you might guess from the word, all of those uses have something to do with the superclass of the class in which the keyword is used.

Invoke the superclass constructor

When the **super** keyword *(followed by a pair of matching parentheses)* appears as the first executable statement in a constructor, this is an instruction to the runtime system to first call the constructor for the superclass, and then come back and finish executing the code in the constructor for the class to which the constructor belongs.

Call the *noarg* superclass constructor

If the parentheses following the **super** keyword are empty, this is an instruction to call the *noarg* constructor for the superclass.

Invoke a parameterized superclass constructor

If the parentheses are not empty, this is an instruction to find and call a parameterized constructor in the superclass whose formal arguments match the parameters in the parentheses.

Invoke the *noarg* superclass constructor by default

Here is an important point that is not illustrated above. If the first executable statement in your constructor is not an instruction to call the constructor for the superclass, an instruction to call the *noarg* constructor for the superclass will effectively be inserted into your constructor code before it is compiled.

Therefore, a constructor for the superclass is ***always called*** before the code in the constructor for your new class is executed.

You can choose the superclass constructor

The superclass constructor that is called may be the *noarg* constructor for the superclass, or you can force it to be a parameterized constructor by inserting something like

super(3,x,4.5);

as the first instruction in your constructor definition.

Always have a *noarg* constructor ...

Now you should understand why I told you in an earlier module that the classes you define should almost always have a *noarg* constructor, either the default *noarg* version, or a *noarg* version of your own design.

If your classes don't have a *noarg* constructor, then anyone who extends your classes will be required to put code in the constructor for their new class to call a parameterized constructor in your class.

In this program, the **super();** statement in the **Subclass** constructor causes the *noarg* constructor for the **Superclass** to be called. That *noarg* constructor is shown in the following code fragment.

NOTE:

```
  class Superclass{
private int x = 1;

public Superclass(){
  x = 5;
}//end constructor

//...
}//end Superclass
```

Additional code

Beyond an exposure and explanation of the use of the **super** keyword to call the superclass constructor, this program plays a few games with initial values of instance variables just to see if you are alert to that sort of thing. However, none of that should be new to you, so I won't discuss it further here.

Back to Question 1 (p. 203)

-end-

3.11 Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables[15]

3.11.1 Table of Contents

3.11.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 232) to easily find and view the listings while you are reading about them.

[15] This content is available online at <http://cnx.org/content/m45296/1.3/>.

3.11.3 Questions

3.11.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 224) ?

- A. Compiler Error
- B. Runtime Error
- C. 33
- D. None of the above

Listing 1: Listing for Question 1.

```
    public class Ap110{
  public static void main(
                        String args[]){
                new Worker().doThis();
  }//end main()
}//end class Ap110

class Worker{
  private int data = 33;

  public void doThis(){
    new Helper().helpMe(this);
  }//end area()

  public String getData(){
    return data;
  }//end getData()
}// end class Worker

class Helper{
  public void helpMe(Worker param){
    System.out.println(
                      param.getData());
  }//end helpMe()
}//end class Helper
```

Answer and Explanation (p. 245)

3.11.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 225) ?

- A. Compiler Error
- B. Runtime Error

- C. 33
- D. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap111{
  public static void main(
                        String args[]){
        new Worker().doThis();
  }//end main()
}//end class Ap111

class Worker{
  private int data = 33;

  public void doThis(){
    new Helper().helpMe(this);
  }//end area()

  public String getData(){
    return "" + data;
  }//end getData()
}// end class Worker

class Helper{
  public void helpMe(Worker param){
    System.out.println(
                      param.getData());
  }//end helpMe()
}//end class Helper
```

3.124

Answer and Explanation (p. 243)

3.11.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 226) ?

- A. Compiler Error
- B. Runtime Error
- C. 11
- D. 22
- E. 33
- F. 44
- G. None of the above.

Listing 3: Listing for Question 3.

```
    public class Ap112{
  public static void main(
                        String args[]){
    Worker obj1 = new Worker(11);
    Worker obj2 = new Worker(22);
    Worker obj3 = new Worker(33);
    Worker obj4 = new Worker(44);
    obj2.doThis();
  }//end main()
}//end class Ap112

class Worker{
  private int data;

  public Worker(int data){
    this.data = data;
  }//end constructor

  public void doThis(){
    System.out.println(this);
  }//end area()

  public String toString(){
    return "" + data;
  }//end toString()
}// end class Worker
```

3.125

Answer and Explanation (p. 241)

3.11.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 227) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 4: Listing for Question 4.

```
    public class Ap113{
  public static void main(
                        String args[]){
    System.out.println(
                   new Worker().twoPI);
  }//end main()
}//end class Ap113

class Worker{
  public static final double twoPI;

  public Worker(){
    twoPI = 2 * Math.PI;
  }//end constructor
}// end class Worker
```

3.126

Answer and Explanation (p. 241)

3.11.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 228) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 5: Listing for Question 5.

```
    public class Ap114{
  public static void main(
                        String args[]){
    System.out.println(
                   new Worker().twoPI);
  }//end main()
}//end class Ap114

class Worker{
  public static final double twoPI
                         = 2 * Math.PI;
}// end class Worker
```

3.127

Answer and Explanation (p. 240)

3.11.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 228) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 6: Listing for Question 6.

```
    public class Ap115{
  public static void main(
                        String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap115

class Worker{
  public static final double twoPI
                         = 2 * Math.PI;
}// end class Worker
```

3.128

Answer and Explanation (p. 239)

3.11.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 229) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. C. 6.283185307179586
- D. None of the above.

Listing 7: Listing for Question 7.

```
    public class Ap116{
  public static void main(
                        String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap116

class Worker{
  public static final double twoPI
                             = 2 * myPI;
  public static final double myPI
                              = Math.PI;
}// end class Worker
```

3.129

Answer and Explanation (p. 238)

3.11.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 230) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 0.0 false
- D. null null null
- E. None of the above.

Listing 8: Listing for Question 8.

```
    public class Ap117{
  public static void main(
                        String args[]){
    new Worker().display();
  }//end main()
}//end class Ap117

class Worker{
  private int myInt;
  private double myDouble;
  private boolean myBoolean;

  public void display(){
    System.out.print(myInt);
    System.out.print("  " + myDouble);
    System.out.println(
                     "  " + myBoolean);
  }//end display()

}// end class Worker
```

3.130

Answer and Explanation (p. 237)

3.11.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 231) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 false 5 true
- D. None of the above.

Listing 9: Listing for Question 9.

```
    public class Ap118{
  public static void main(
                        String args[]){
    new Worker().display();
    new Worker(5,true).display();
    System.out.println();
  }//end main()
}//end class Ap118

class Worker{
  private int myInt;
  private boolean myBoolean;

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  public void display(){
    System.out.print(myInt);
    System.out.print(
                " " + myBoolean + " ");
  }//end display()

}// end class Worker
```

3.131

Answer and Explanation (p. 236)

3.11.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 232) ?

- A. Compiler Error
- B. Runtime Error
- C. 20 222.0 false — 5 222.0 true
- D. None of the above.

Listing 10: Listing for Question 10.

```
    public class Ap119{
  public static void main(
                        String args[]){
    new Worker().display();
    System.out.print("--- ");
    new Worker(5,true).display();
    System.out.println();
  }//end main()
}//end class Ap119

class Worker{
  private int myInt = 100;
  private double myDouble = 222.0;
  private boolean myBoolean;

  public Worker(){
    myInt = 20;
  }//end noarg constructor

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  public void display(){
    System.out.print(myInt + " ");
    System.out.print(myDouble + " ");
    System.out.print(myBoolean + " ");
  }//end display()

}// end class Worker
```

3.132

Answer and Explanation (p. 233)

3.11.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 1 (p. 224) . Listing for Question 1.
- Listing 2 (p. 225) . Listing for Question 2.
- Listing 3 (p. 226) . Listing for Question 3.
- Listing 4 (p. 227) . Listing for Question 4.
- Listing 5 (p. 228) . Listing for Question 5.

- Listing 6 (p. 228) . Listing for Question 6.
- Listing 7 (p. 229) . Listing for Question 7.
- Listing 8 (p. 230) . Listing for Question 8.
- Listing 9 (p. 231) . Listing for Question 9.
- Listing 10 (p. 232) . Listing for Question 10.

3.11.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables
- File: Ap0100.htm
- Originally published: 2004
- Published at cnx.org: 12/08/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.11.6 Answers

3.11.6.1 Answer 10

C. 20 222.0 false — 5 222.0 true

3.11.6.1.1 Explanation 10

Four ways to initialize instance variables

There are at least four ways to establish initial values for instance variables *(you may be able to think of others)* :

1. Allow them to take on their default values.
2. Establish their values using initialization expressions.
3. Establish their values using hard-coded values within a constructor.
4. Establish their values using parameter values passed to parameterized constructors.

Using the first two ways

The following fragment illustrates the first two of those four ways.

NOTE:

```
  class Worker{
private int myInt = 100;
private double myDouble = 222.0;
private boolean myBoolean;

//...
```

In the above fragment, the instance variables named **myInt** and **myDouble** receive their initial values from initialization expressions. In these two cases, the initialization expressions are very simple. They are simply literal expressions. However, they could be much more complex if needed.

The variable named **myBoolean** in the above fragment is allowed to take on its default value of false.

Replacing the default *noarg* constructor

The next fragment shows one of the two overloaded constructors in the class named **Worker**. This constructor is a replacement for the default *noarg* constructor.

NOTE:

```
  class Worker{
private int myInt = 100;
private double myDouble = 222.0;
private boolean myBoolean;

public Worker(){
  myInt = 20;
}//end noarg constructor

//...
```

Using hard-coded values for initialization

This fragment illustrates the third of the four ways listed earlier to establish the initial value of the instance variables of an object of the class named **Worker**. In particular, this fragment assigns the hard-coded value 20 to the instance variable named **myInt**, thus overwriting the value of 100 previously established for that variable by an initialization expression.

*(All objects instantiated from the **Worker** class using this noarg constructor would have the same initial value for the variable named **myInt**.)*

Note, that this constructor does not disturb the initial values of the other two instance variables that were earlier established by an initialization expression, or by taking on the default value. Thus, the initial values of these two instance variables remain as they were immediately following the declaration of the variables.

Initial values using this *noarg* constructor

When an object of the **Worker** class is instantiated using this constructor and the values of the three instance variables are displayed, the results are as shown below:

20 222.0 false

The value of **myInt** is 20 as established by the constructor. The value of **myDouble** is 222.0 as established by the initialization expression, and the value of **myBoolean** is false as established by default.

Using constructor parameters for initialization

The next fragment shows the last of the four ways listed earlier for establishing the initial value of an instance variable.

NOTE:

```
    public class Ap119{
  public static void main(
                       String args[]){
    //...
    new Worker(5,true).display();
    //...
  }//end main()
}//end class Ap119

class Worker{
  private int myInt = 100;
  private double myDouble = 222.0;
  private boolean myBoolean;
  //...

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  //...
```

A parameterized constructor

The above fragment shows the second of two overloaded constructors for the class named **Worker** . This constructor uses two incoming parameter values to establish the values of two of the instance variables, overwriting whatever values may earlier have been established for those variables.

The above fragment uses this constructor to instantiate an object of the **Worker** class, assigning incoming parameter values of 5 and *true* to the instance variables named **myInt** and **myBoolean** respectively. This overwrites the value previously placed in the variable named **myInt** by the initialization expression. It also overwrites the default value previously placed in the instance variable named **myBoolean** .

(Note that this constructor doesn't disturb the value for the instance variable named ***myDouble*** *that was previously established through the use of an initialization expression.)*

Initial values using parameterized constructor

After instantiating the new object, this fragment causes the values of all three instance variables to be displayed. The result is:

5 222.0 true

As you can see, the values contained in the instance variables named **myInt** and **myBoolean** are the values of 5 and true placed there by the constructor, based on incoming parameter values. The value in the instance variable named **myDouble** is the value placed there by the initialization expression when the variable was declared.

Default initialization

If you don't take any steps to initialize instance variables, they will be automatically initialized. Numeric instance variables will be initialized with zero value for the type of variable involved. Instance variables of type **boolean** will be initialized to false. Instance variables of type **char** will be initialized to a Unicode value with all 16 bits set to zero. Reference variables will be initialized to null.

Initialization expression

If you provide an initialization expression for an instance variable, the value of the expression will overwrite the default value, and the value of the initialization expression will become the initial value for the instance variable.

Assignment in constructor code

If you use an assignment statement in a constructor to assign a value to an instance variable, that value will overwrite the value previously placed in the instance variable either by default, or by use of an initialization expression. The constructor has the "last word" on the matter of initialization of instance variables.

Back to Question 10 (p. 231)

3.11.6.2 Answer 9

A. Compiler Error

3.11.6.2.1 Explanation 9

The default constructor

When you define a class, you are not required to define a constructor for the class. If you do not define a constructor for the class, a default constructor that takes no arguments will be provided on your behalf. You can instantiate new objects of the class by applying the new operator to the default constructor as shown in the following code fragment from Question 8 (p. 229) .

NOTE:

```
new Worker().display();
```

Behavior of the default constructor

As illustrated in Question 8 (p. 229) , when you don't provide a constructor that purposely initializes the values of instance variables, or initialize them in some other manner, they will automatically be initialized to the default values described in Question 8 (p. 229) .

Defining overloaded constructors

You can also define one or more overloaded constructors having different formal argument lists. The typical intended purpose of such constructors is to use incoming parameter values to initialize the values of instance variables in the new object.

A parameterized constructor

This is illustrated in the following code fragment. This fragment receives two incoming parameters and uses the values of those two parameters to initialize the values of two instance variables belonging to the new object.

NOTE:

```
    class Worker{
  private int myInt;
  private boolean myBoolean;

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  //display() omitted for brevity

}// end class Worker
```

If you define any constructors ...

However, there is a pitfall that you must never forget.

If you define any constructors in your new class, you must define all constructors that will ever be required for your new class.

If you define any constructors, the default constructor will no longer be provided automatically. Therefore, if a constructor that takes no arguments will ever be needed for your new class, and you define one or more parameterized constructors, you must define the *noarg* constructor when you define your class.

A parameterized constructor for Worker

The class named **Worker** in this program defines a constructor that receives two incoming parameters, one of type **int** and the other of type **boolean** . It uses those two incoming parameters to initialize two instance variables of the new object.

Oops!

However, it does not define a constructor with no arguments in the formal argument list *(commonly called a noarg constructor)* .

Calling the missing *noarg* constructor

The following code in the **main** method of the controlling class attempts to instantiate two objects of the **Worker** class. The first call of the constructor passes no parameters to the constructor. Thus, it requires a *noarg* constructor in order to instantiate the object.

NOTE:

```
    public class Ap118{
  public static void main(
                       String args[]){
    new Worker().display();
    new Worker(5,true).display();
    System.out.println();
  }//end main()
}//end class Ap118
```

A compiler error

Since there is no constructor defined in the **Worker** class with an empty formal argument list *(and the default version is not provided)* , the program produces the following compiler error.

NOTE:

```
    Ap118.java:11: cannot resolve symbol
symbol  : constructor Worker
()
location: class Worker
    new Worker().display();
```

Back to Question 9 (p. 230)

3.11.6.3 Answer 8

C. 0 0.0 false

3.11.6.3.1 Explanation 8

All instance variables are initialized to default values

All instance variables are automatically initialized to default values if the author of the class doesn't take explicit steps to cause them to initialized to other values.

The default values

Numeric variables are automatically initialized to zero, while **boolean** variables are automatically initialized to false. Instance variables of type **char** are initialized to a Unicode value with all 16 bits set to zero. Reference variables are initialized to null.

Back to Question 8 (p. 229)

3.11.6.4 Answer 7

A. Compiler Error

3.11.6.4.1 Explanation 7

Pushing the compiler beyond its limits

Compared to many programming environments, the Java compiler is very forgiving. However, there is a limit to how far even the Java compiler is willing to go to keep us out of trouble.

Initializing the value of a static variable

We can initialize the value of a **static** variable using an initialization expression as follows:

NOTE:

```
public static final MY_CONSTANT
   = initialization expression;
```

Important point

It is necessary for the compiler to be able to evaluate the initialization expression when it is encountered.

Illegal forward reference

This program attempts to use an initialization expression that makes use of the value of another **static** variable *(myPI)* that has not yet been established at that point in the compilation process. As a result, the program produces the following compiler error under JDK 1.3.

NOTE:

```
Ap116.java:18: illegal forward reference
 = 2 * myPI;
       ^
```

Reverse the order of the variable declarations

The problem can be resolved by reversing the order of the two **static** variable declarations in the following revised version of the program.

NOTE:

```
    public class Ap116{
  public static void main(
                       String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap116

class Worker{
  public static final double myPI
                             = Math.PI;
  public static final double twoPI
```

```
                                = 2 * myPI;

  }// end class Worker
```

This revised version of the program compiles and executes successfully.

Back to Question 7 (p. 229)

3.11.6.5 Answer 6

C. 6.283185307179586

3.11.6.5.1 Explanation 6

Access via an object

Question 5 (p. 227) illustrated the fact that a **public static final** member variable of a class can be accessed via a reference to an object instantiated from the class.

Not the only way to access a *static* variable

However, that is not the only way in which **static** member variables can be accessed. More importantly, **public static** member variables of a class can be accessed simply by referring to the name of the class and the name of the member variable joined by a period.

(Depending on other factors, it may not be necessary for the ***static*** *variable to also be declared* ***public*** *, but that is the most general approach.)*

A *public static final* member variable

In this program, the **Worker** class declares and initializes a **public static final** member variable named **twoPI** as shown in the following fragment.

NOTE:

```
    class Worker{
  public static final double twoPI
                          = 2 * Math.PI;
}// end class Worker
```

Accessing the *static* variable

The single statement in the **main** method of the controlling class accesses and displays the value of the **public static final** member variable named **twoPI** as shown in the following fragment.

NOTE:

```
    public class Ap115{
  public static void main(
                         String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap115
```

Objects share one copy of *static* variables

Basically, when a member variable is declared **static** , no matter how many objects are instantiated from a class *(including no objects at all)* , they all share a single copy of the variable.

Sharing can be dangerous

This sharing of a common variable leads to the same kind of problems that have plagued programs that use ***global*** variables for years. If the code in any object changes the value of the **static** variable, it is changed insofar as all objects are concerned.

Should you use non-final *static* variables?

Most authors will probably agree that in most cases, you probably should not use **static** variables unless you also make them **final** .

*(There are some cases, such as counting the number of objects instantiated from a class, where a non-final **static** variable may be appropriate. However, the appropriate uses of non-final **static** variables are few and far between.)*

Should you also make *static* variables *public* ?

If you make your variables **static** and **final** , you will often also want to make them **public** so that they are easy to access. There are numerous examples in the standard Java class libraries where variables are declared as **public** , **static** , and **final** . This is the mechanism by which the class libraries create constants and make them available for easy access on a widespread basis.

The Color class

For example, the **Color** class defines a number of **public static final** variables containing the information that represents generic colors such as ORANGE, PINK, and MAGENTA. *(By convention, constants in Java are written with all upper-case characters, but that is not a technical requirement.)*

If you need generic colors and not custom colors, you can easily access and use these color values without the requirement to mix red, green, and blue to produce the desired color values.

Back to Question 6 (p. 228)

3.11.6.6 Answer 5

C. 6.283185307179586

3.11.6.6.1 Explanation 5

A *public static final* variable

This program declares a **public static final** member variable named **twoPI** in the class named **Worker** , and properly initializes it when it is declared as shown in the following code fragment.

NOTE:

```
class Worker{  public static final double twoPI
                        = 2 * Math.PI;
}/}// end class Worker
```

From that point forward in the program, this member variable named **twoPI** behaves like a constant, meaning that any code that attempts to change its value will cause a compiler error *(as in the program in Question 4 (p. 226))* ..

Accessing the static variable

The following single statement that appears in the **main** method of the controlling class instantiates a new object of the **Worker** class, accesses, and displays the **public static final** member variable named **twoPI** .

NOTE:

```
 public static void main(
                      String args[]){
System.out.println(
                new Worker().twoPI);
}  }//end main()
```

*(Note for future discussion that the variable named **twoPI** is accessed via a reference to an object instantiated from the class named **Worker** .)*

This causes the **double** value 6.283185307179586 to be displayed on the standard output device.

Back to Question 5 (p. 227)

3.11.6.7 Answer 4

A. Compiler Error

3.11.6.7.1 Explanation 4

A **final** variable

When a member variable of a class *(not a local variable)* is declared **final** , its value must be established when the variable is declared. This program attempts to assign a value to a **final** member variable after it has been declared, producing the following compiler error under JDK 1.3.

NOTE:

```
    Ap113.java:20: cannot assign a value to
final variable twoPI
    twoPI = 2 * Math.PI;
```

Back to Question 4 (p. 226)

3.11.6.8 Answer 3

D. 22

3.11.6.8.1 Explanation 3

Two uses of the *this* keyword

This program illustrates two different uses of the **this** keyword.

Disambiguating a reference to a variable

Consider first the use of **this** that is shown in the following code fragment.

NOTE:

```
    class Worker{
  private int data;

  public Worker(int data){
    this.data = data;
  }//end constructor
```

Very common usage

The code in the above fragment is commonly used by many Java programmers. All aspiring Java programmers need to know how to read such code, even if they elect not to use it. In addition, understanding this code should enhance your overall understanding of the use and nature of the **this** keyword.

A parameterized constructor

The above fragment shows a parameterized constructor for the class named **Worker** . This constructor illustrates a situation where there is a local parameter named **data** that has the same name as an instance variable belonging to the object.

Casting a shadow

The existence of the local parameter named **data** casts a shadow on the instance variable having the same name, making it inaccessible by using its name alone.

(A local variable having the same name as an instance variable casts a similar shadow on the instance variable.)

In this shadowing circumstance, when the code in the constructor refers simply to the name **data** , it is referring to the local parameter having that name. In order for the code in the constructor to refer to the instance variable having the name data, it must refer to it as **this.data** .

In other words ...

In other words, **this.data** is a reference to an instance variable named **data** belonging to the object being constructed by the constructor *(this object)* .

Not always necessary

You could always use this syntax to refer to an instance variable of the object being constructed if you wanted to. However, the use of this syntax is necessary only when a local parameter or variable has the same name as the instance variable and casts a shadow on the instance variable. When this is not the case, you can refer to the instance variable simply by referring to its name without the keyword **this** .

Finally, the main point ...

Now consider the main point of this program. The following fragment shows the **main** method of the controlling class for the application.

NOTE:

```
  public class Ap112{
public static void main(
                      String args[]){
  Worker obj1 = new Worker(11);
  Worker obj2 = new Worker(22);
  Worker obj3 = new Worker(33);
  Worker obj4 = new Worker(44);
  obj2.doThis();
 }//end main()
}//end class Ap112
```

Four different objects of type Worker

The code in the above fragment instantiates four different objects from the class named **Worker** , passing a different value to the constructor for each object. Thus, individual instance variable in each of the four objects contain the **int** values 11, 22, 33, and 44 respectively.

Call an instance method on one object

Then the code in the **main** method calls the instance method named **doThis** on only one of the objects, which is the one referred to by the reference variable named **obj2** .

An overridden **toString** method of the **Worker** class is eventually called to return a **String** representation of the value stored in the instance variable named **data** for the purpose of displaying that value on the standard output device.

Overridden toString method

The next fragment shows the overridden **toString** method for the **Worker** class. As you can see, this overridden method constructs and returns a reference to a **String** representation of the **int** value stored in the instance variable named **data** . Thus, depending on which object the **toString** method is called on, different string values will be returned by the overridden method.

NOTE:

```
 public String toString(){
  return "" + data;
 }//end toString()
}// end class Worker
```

Passing reference to this object to println method

The next fragment shows the **doThis** instance method belonging to each object instantiated from the **Worker** class. When this method is called on a specific object instantiated from the **Worker** class, it uses the **this** keyword to pass that specific object's reference to the **println** method. The **println** method uses that reference to call the **toString** method on that specific object. This, in turn causes a **String** representation of the value of the instance variable named **data** belonging to that specific object to be displayed.

NOTE:

```
    public void doThis(){
  System.out.println(this);
}//end area()
```

The bottom line

In this program, the instance variable in the object referred to by **obj2** contains the value 22. The instance variables in the other three objects instantiated from the same class contain different values.

The bottom line is that the following statement in the **main** method causes the value 22 to be displayed on the standard output device. Along the way, the **this** keyword is used to cause the **println** method to get and display the value stored in a specific object, and to ignore three other objects that were instantiated from the same class.

NOTE:

```
      obj2.doThis();
```

Back to Question 3 (p. 225)

3.11.6.9 Answer 2

C. 33

3.11.6.9.1 Explanation 2

The **this** keyword

The key to an understanding of this program lies in an understanding of the single statement that appears in the method named **doThis**, as shown in the following fragment.

NOTE:

```
    public void doThis(){
  new Helper().helpMe(this);
}//end area()
```

The keyword named **this** has several uses in Java, some of which are explicit, and some of which take place behind the scenes.

What do you need to know about the this keyword?

One of the uses of the keyword **this** is passing the implicit parameter in its entirety to another method. That is exactly what this program does. But what is the implicit parameter named **this** anyway?

Every object holds a reference to itself

This implicit reference can be accessed using the keyword **this** in a non-static *(instance)* method belonging to the object. *(The implicit reference named **this** cannot be accessed from within a **static** method for reasons that won't be discussed here.)*

Calling an instance method

An instance method can only be called by referring to a specific object and joining that object's reference to the name of the instance method using a period as the joining operator. This is illustrated in the following statement, which calls the method named **doThis** on a reference to an object of the class named **Worker** .

NOTE:

```
new Worker().doThis();
```

An anonymous object

The above statement creates an anonymous object of the class named **Worker** . *(An anonymous object is an object whose reference is not assigned to a named reference variable.)*

The code to the left of the period returns a reference to the new object. Then the code calls the instance method named **doThis** on the reference to the object.

Which object is *this* object?

When the code in the instance method named **doThis** refers to the keyword **this** , it is a reference to the specific object on which the **doThis** method was called. The statement in the following fragment passes a reference to that specific instance of the **Worker** class to a method named **helpMe** in a new object of the **Helper** class.

NOTE:

```
 public void doThis(){
new Helper().helpMe(this);
}//end area()
```

A little help here please

The **helpMe** method is shown in the following fragment.

NOTE:

```
  class Helper{
public void helpMe(Worker param){
  System.out.println(
                    param.getData());
}//end helpMe()
}//end class Helper
```

Using the incoming reference

The code in the **helpMe** method uses the incoming reference to the object of the **Worker** class to call the **getData** method on that object.

Thus code in the **helpMe** method is able to call a method in the object containing the method that called the **helpMe** method in the first place.

A callback scenario

When a method in one object calls a method in another object, passing **this** as a parameter, that makes it possible for the method receiving the parameter to make a callback to the object containing the method that passed **this** as a parameter.

The **getData** method returns a **String** representation of the **int** instance variable named **data** with a value of 33 that is contained in the object of the **Worker** class.

Display the value

The code in the **helpMe** method causes that string to be displayed on the computer screen.

And the main point is ...

Any number of objects can be instantiated from a given class. A given instance method can be called on any of those objects. When the code in such an instance method refers to **this** , it is referring to the specific object on which it was called, and is not referring to any of the many other objects that may have been instantiated from the same class.

Back to Question 2 (p. 224)

3.11.6.10 Answer 1

A. Compiler Error

3.11.6.10.1 Explanation 1

A wakeup call

The purpose of this question is simply to give you a wakeup call. The declaration for the method named **getData** indicates that the method returns a reference to an object of the class **String** . However, the code in the method attempts to return an **int** . The program produces the following compiler error under JDK 1.3.

NOTE:

```
    found   : int
required: java.lang.String
    return data;
```

Back to Question 1 (p. 224)

-end-

3.12 Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior[16]

3.12.1 Table of Contents

3.12.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 255) to easily find and view the listings while you are reading about them.

[16] This content is available online at <http://cnx.org/content/m45308/1.3/>.

3.12.3 Questions

3.12.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 246) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. None of the above.

Listing 1: Listing for Question 1.

```
    public class Ap120{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap120

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Answer and Explanation (p. 265)

3.12.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 247) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap121{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap121

class Worker{
  void doIt(){
    Base myVar = new A();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.134

Answer and Explanation (p. 264)

3.12.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 248) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. None of the above.

Listing 3: Listing for Question 3.

```
    public class Ap122{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap122

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  abstract public void test();
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.135

Answer and Explanation (p. 263)

3.12.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 249) ?

- A. Compiler Error
- B. Runtime Error
- C. A
- D. None of the above.

Listing 4: Listing for Question 4.

```
    public class Ap123{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap123

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

abstract class Base{
  abstract public void test();
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.136

Answer and Explanation (p. 262)

3.12.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 250) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A
- E. None of the above.

Listing 5: Listing for Question 5.

```
    public class Ap124{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap124

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

abstract class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.137

Answer and Explanation (p. 262)

3.12.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 251) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A
- E. None of the above.

Listing 6: Listing for Question 6.

```
    public class Ap125{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap125

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.138

Answer and Explanation (p. 261)

3.12.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 252) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A
- E. None of the above.

Listing 7: Listing for Question 7.

```
    public class Ap126{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap126

class Worker{
  void doIt(){
    Base myVar = new Base();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.139

Answer and Explanation (p. 259)

3.12.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 253) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A
- E. None of the above.

Listing 8: Listing for Question 8.

```
    public class Ap127{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap127

class Worker{
  void doIt(){
    Base myVar = new A();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.140

Answer and Explanation (p. 258)

3.12.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 254) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A
- E. None of the above.

Listing 9: Listing for Question 9.

```
    public class Ap128{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap128

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

3.141

Answer and Explanation (p. 258)

3.12.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 255) ?

- A. Compiler Error
- B. Runtime Error
- C. Base
- D. A B
- E. None of the above.

Listing 10: Listing for Question 10.

```
    public class Ap129{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap129

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    myVar = new B();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B
```

3.142

Answer and Explanation (p. 256)

3.12.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 4 (p. 249) . Listing for Question 4.
- Listing 5 (p. 250) . Listing for Question 5.
- Listing 6 (p. 251) . Listing for Question 6.
- Listing 7 (p. 252) . Listing for Question 7.
- Listing 8 (p. 253) . Listing for Question 8.
- Listing 9 (p. 254) . Listing for Question 9.
- Listing 10 (p. 255) . Listing for Question 10.

3.12.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior
- File: Ap0110.htm
- Originally published: 2002
- Published at cnx.org: 12/08/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.12.6 Answers

3.12.6.1 Answer 10

D. A B

3.12.6.1.1 Explanation 10

Another illustration of simple polymorphic behavior

In this program, two classes named **A** and **B** extend the class named **Base** , each overriding the method named **test** to produce different behavior. *(Typically, overridden methods in different classes will produce different behavior, even though they have the same names.)*

Behavior appropriate for object on which method is called

In other words, the behavior of the method named **test** , when called on a reference to an object of type **A** , is different from the behavior of the method named **test** when called on a reference to an object of type **B** .

The method definitions

The definitions of the two classes named **A** and **B**, along with the two versions of the overridden method named **test** are shown in the following fragment.

NOTE:

```
    class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B
```

Store a subclass object's reference as a superclass type

The program declares a reference variable of type **Base**, instantiates a new object of the class named **A**, and assigns that object's reference to the reference variable of type **Base**. Then it calls the method named **test** on that reference as shown in the following fragment.

NOTE:

```
        Base myVar = new A();
    myVar.test();
```

Polymorphic behavior applies

Simple polymorphic behavior causes the overridden version of the method named **test**, defined in the class named **A**, *(as opposed to the versions defined in class **Base** or class **B**)* to be executed. This causes the letter **A** followed by a space character to be displayed on the standard output device.

Store another subclass object's reference as superclass type

Then the program instantiates a new object from the class named **BB**, and assigns that object's reference to the same reference variable, overwriting the reference previously stored there. *(This causes the object whose reference was previously stored in the reference variable to become eligible for garbage collection in this case.)*

Then the program calls the method named **test** on the reference as shown in the following fragment.

NOTE:

```
        myVar = new B();
    myVar.test();
```

Polymorphic behavior applies again

This time, simple polymorphic behavior causes the overridden version of the method named **test**, defined in the class named **B**, *(as opposed to the versions defined in class **Base** or class **A**)* to be executed. This causes the letter **B** followed by a space character to be displayed on the standard output device.

Once again, what is runtime polymorphic behavior?

With runtime polymorphic behavior, the method selected for execution is based, not on the type of the reference variable holding the reference to the object, but rather on the actual class from which the object was instantiated.

If the method was properly overridden, the behavior exhibited by the execution of the method is appropriate for an object of the class from which the object was instantiated.

Back to Question 10 (p. 254)

3.12.6.2 Answer 9

D. A

3.12.6.2.1 Explanation 9

Compiles and executes successfully

This program compiles and executes successfully causing the version of the method named **test** , which is overridden in the class named **A** to be executed. That overridden method is shown in the following fragment.

NOTE:

```
    class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

So, what is the issue here?

The purpose of this program is to determine if you understand polymorphic behavior and the role of downcasting. Consider the following fragment taken from the program in Question 8 (p. 252) .

NOTE:

```
        Base myVar = new A();
    ((A)myVar).test();
```

The downcast is redundant

As you learned in the discussion of Question 8 (p. 252) , the downcast isn't required, and it has no impact on the behavior of the program in Question 8 (p. 252) .

This program behaves exactly the same with the second statement in the above fragment replaced by the following statement, which does not contain a downcast.

NOTE:

```
        myVar.test();
```

Again, you need to know when downcasting is required, when it isn't required, and to make use of that knowledge to downcast appropriately.

Back to Question 9 (p. 253)

3.12.6.3 Answer 8

D. A

3.12.6.3.1 Explanation 8

Compiles and executes successfully

This program compiles and executes successfully causing the version of the method named **test** , which is overridden in the class named **A** to be executed. That overridden method is shown in the following fragment.

NOTE:

```
    class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

So, what is the issue here?

The purpose of this program is to determine if you understand polymorphic behavior and the role of downcasting, as shown in the following fragment.

NOTE:

```
        Base myVar = new A();
    ((A)myVar).test();
```

This would be a simple case of polymorphic behavior were it not for the downcast shown in the above fragment.

The downcast is redundant

Actually, the downcast was placed there to see if you could determine that it is redundant. It isn't required, and it has no impact on the behavior of this program. This program would behave exactly the same if the second statement in the above fragment were replaced with the following statement, which does not contain a downcast.

NOTE:

```
        myVar.test();
```

You need to know when downcasting is required, when it isn't required, and to make use of that knowledge to downcast appropriately.

Back to Question 8 (p. 252)

3.12.6.4 Answer 7

B. Runtime Error

3.12.6.4.1 Explanation 7

Storing a reference as a superclass type

You can store an object's reference in any reference variable whose declared type is a superclass of the actual class from which the object was instantiated.

May need to downcast later

Later on, when you attempt to make use of that reference, you may need to downcast it. Whether or not you will need to downcast will depend on what you attempt to do.

In order to call a method ...

For example, if you attempt to call a method on the reference, but that method is not defined in or inherited into the class of the reference variable, then you will need to downcast the reference in order to call the method on that reference.

Class Base defines method named test

This program defines a class named **Base** that defines a method named **test** .

Class A extends Base and overrides test

The program also defines a class named **A** that extends **Base** and overrides the method named **test** as shown in the following fragment.

NOTE:

```
    class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

A new object of the class Base

The program instantiates a new object of the class **Base** and stores a reference to that object in a reference variable of type **Base** , as shown in the following fragment.

NOTE:

```
        Base myVar = new Base();
    ((A)myVar).test();
```

Could call test directly on the reference

Having done this, the program could call the method named **test** directly on the reference variable using a statement such as the following, which is not part of this program.

NOTE:

```
        myVar.test();
```

This statement would cause the version of the method named **test** defined in the class named **Base** to be called, causing the word **Base** to appear on the standard output device.

This downcast is not allowed

However, this program attempts to cause the version of the method named **test** defined in the class named **A** to be called, by downcasting the reference to type **A** before calling the method named **test** . This is shown in the following fragment.

NOTE:

```
        ((A)myVar).test();
```

A runtime error occurs

This program compiles successfully. However, the downcast shown above causes the following runtime error to occur under JDK 1.3:

NOTE:

```
    Exception in thread "main" java.lang.ClassCastException: Base
        at Worker.doIt(Ap126.java:22)
        at Ap126.main(Ap126.java:15)
```

What you can do

You can store an object's reference in a reference variable whose type is a superclass of the class from which the object was originally instantiated. Later, you can downcast the reference back to the type *(class)* from which the object was instantiated.

What you cannot do

However, you cannot downcast an object's reference to a subclass of the class from which the object was originally instantiated.

Unfortunately, the compiler is unable to detect an error of this type. The error doesn't become apparent until the exception is thrown at runtime.

Back to Question 7 (p. 251)

3.12.6.5 Answer 6

C. Base

3.12.6.5.1 Explanation 6

Totally straightforward code

This rather straightforward program instantiates an object of the class named **Base** and assigns that object's reference to a reference variable of the type **Base** as shown in the **following fragment** .

NOTE:

```
        Base myVar = new Base();
    myVar.test();
```

Then it calls the method named **test** on the reference variable.

Class Base defines the method named test

The class named **Base** contains a concrete definition of the method named **test** as shown in the following fragment. This is the method that is called by the code shown in the above fragment (p. 261) .

NOTE:

```
    class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base
```

Class A is just a smokescreen

The fact that the class named **A** extends the class named **Base** , and overrides the method named **test** , as shown in the following fragment, is of absolutely no consequence in the behavior of this program. Hopefully you understand why this is so. If not, then you still have a great deal of studying to do on Java inheritance.

NOTE:

```
    class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Back to Question 6 (p. 250)

3.12.6.6 Answer 5

A. Compiler Error

3.12.6.6.1 Explanation 5

Cannot instantiate an abstract class

This program defines an **abstract** class named **Base** . Then it violates one of the rules regarding **abstract** classes, by attempting to instantiate an object of the **abstract** class as shown in the following code fragment.

NOTE:

```
Base myVar = new Base();
```

The program produces the following compiler error under JDK 1.3:

NOTE:

```
Ap124.java:19: Base is abstract; cannot be instantiated
Base myVar = new Base();
```

Back to Question 5 (p. 249)

3.12.6.7 Answer 4

C. A

3.12.6.7.1 Explanation 4

An abstract class with an abstract method

This program illustrates the use of an **abstract** class containing an **abstract** method to achieve *polymorphic behavior* .

The following code fragment shows an **abstract** class named **Base** that contains an **abstract** method named **test** .

NOTE:

```
  abstract class Base{
abstract public void test();
}//end class Base
```

Extending abstract class and overriding abstract method

The class named **A** , shown in the following fragment extends the **abstract** class named **Base** and overrides the **abstract** method named **test** .

NOTE:

```
  class A extends Base{
public void test(){
  System.out.print("A ");
}//end test()
}//end class A
```

Can store a subclass reference as a superclass type

Because the class named **A** extends the class named **Base** , a reference to an object instantiated from the class named **A** can be stored in a reference variable of the declared type **Base** . No cast is required in this case.

Polymorphic behavior

Furthermore, because the class named **Base** contains the method named **test** , *(as an* ***abstract*** *method)* , when the method named **test** is called on a reference to an object of the class named **A** , stored in a reference variable of type **Base** , the *overridden* version of the method as defined in the class named **A** will actually be called. This is polymorphic behavior.

(Note, however, that this example does little to illustrate the power of polymorphic behavior because only one class extends the class named ***Base*** *and only one version of the abstract method named* ***test*** *exists. Thus, the system is not required to select among two or more overridden versions of the method named* ***test*** *.)*

The important code

The following code fragment shows the instantiation of an object of the class named **A** and the assignment of that object's reference to a reference variable of type **Base** . Then the fragment calls the method named **test** on the reference variable.

NOTE:

```
    Base myVar = new A();
myVar.test();
```

This causes the overridden version of the method named **test** , shown in the following fragment, to be called, which causes the letter **A** to be displayed on the standard output device.

NOTE:

```
  public void test(){
System.out.print("A ");
}//end test()
```

Back to Question 4 (p. 248)

3.12.6.8 Answer 3

A. Compiler Error

3.12.6.8.1 Explanation 3

Classes can be final or abstract, but not both

A class in Java may be declared **final** . A class may also be declared **abstract** . A class cannot be declared both **final** and **abstract** .

Behavior of final and abstract classes

A class that is declared **final** cannot be extended. A class that is declared **abstract** cannot be instantiated. Therefore, it must be extended to be useful.

An **abstract** class is normally intended to be extended.

Methods can be final or abstract, but not both

A method in Java may be declared **final** . A method may also be declared **abstract** . However, a method cannot be declared both **final** and **abstract** .

Behavior of final and abstract methods

A method that is declared **final** cannot be overridden. A method that is declared **abstract** must be overridden to be useful.

An **abstract** method doesn't have a body.

Abstract classes and methods

A class that contains an **abstract** method must itself be declared **abstract** . However, an **abstract** class is not required to contain **abstract** methods.

Failed to declare the class abstract

In this program, the class named **Base** contains an **abstract** method named **test** , but the class is not declared **abstract** as required.

NOTE:

```
    class Base{
  abstract public void test();
}//end class Base
```

Therefore, the program produces the following compiler error under JDK 1.3:

NOTE:

```
    Ap122.java:24: Base should be declared abstract;
it does not define test in Base
class Base{
```

Back to Question 3 (p. 247)

3.12.6.9 Answer 2

C. A

3.12.6.9.1 Explanation 2

If you missed this ...

If you missed this question, you didn't pay attention to the explanation for Question 1 (p. 246) .

Define a method in a subclass

This program defines a subclass named **A** that extends a superclass named **Base** . A method named **test** is defined in the subclass named **A** but is not defined in any superclass of the class named **A** .

Store a reference as a superclass type

The program declares a reference variable of the superclass type, and stores a reference to an object of the subclass in that reference variable as shown in the following code fragment.

NOTE:

```
        Base myVar = new A();
```

Downcast and call the method

Then the program calls the method named **test** on the reference stored as the superclass type, as shown in the following fragment.

NOTE:

```
        ((A)myVar).test();
```

Unlike the program in Question 1 (p. 246) , the reference is downcast to the true type of the object before calling the method named **test** . As a result, this program does not produce a compiler error.

Why is the cast required?

As explained in Question 1 (p. 246) , it is allowable to store a reference to a subclass object in a variable of a superclass type. Also, as explained in Question 1 (p. 246) , it is not allowable to directly call, on that superclass reference, a method of the subclass object that is not defined in or inherited into the superclass.

However, such a call is allowable if the programmer purposely downcasts the reference to the true type of the object before calling the method.

Back to Question 2 (p. 246)

3.12.6.10 Answer 1

A. Compiler Error

3.12.6.10.1 Explanation 1

Define a method in a subclass

This program defines a subclass named **A** that extends a superclass named **Base** . A method named **test** , is defined in the subclass named **A** , which is not defined in any superclass of the class named **A** .

Store a reference as superclass type

The program declares a reference variable of the superclass type, and stores a reference to an object of the subclass in that reference variable as shown in the following code fragment.

NOTE:

```
Base myVar = new A();
```

Note that no cast is required to store a reference to a subclass object in a reference variable of a superclass type. The required type conversion happens automatically in this case.

Call a method on the reference

Then the program attempts to call the method named **test** on the reference stored as the superclass type, as shown in the following fragment. This produces a compiler error.

NOTE:

```
myVar.test();
```

The reason for the error

It is allowable to store a reference to a subclass object in a variable of a superclass type. However, it is not allowable to directly call, *(on that superclass reference)* , a method of the subclass object that is not defined in or inherited into the superclass.

The following error message is produced by JDK 1.3.

NOTE:

```
    Ap120.java:18: cannot resolve symbol
symbol  : method test  ()
location: class Base
    myVar.test();
```

The solution is ...

This error can be avoided by casting the reference to type **A** before calling the method as shown below:

NOTE:

```
((A)myVar).test();
```

Back to Question 1 (p. 246)
-end-

3.13 Ap0120: Self-assessment, Interfaces and polymorphic behavior[17]

3.13.1 Table of Contents

3.13.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 284) to easily find and view the listings while you are reading about them.

3.13.3 Questions

3.13.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 267) ?

- A. Compiler Error
- B. Runtime Error
- C. Base A-intfcMethod
- D. None of the above.

[17] This content is available online at <http://cnx.org/content/m45303/1.3/>.

Listing 1: Listing for Question 1.

```
    public class Ap131{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap131

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    X myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

3.143

Answer and Explanation (p. 299)

3.13.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 268) ?

- A. Compiler Error
- B. Runtime Error
- C. A-inherMethod A-intfcMethod
- D. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap132{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap132

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    Base myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

3.144

Answer and Explanation (p. 298)

3.13.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 270) ?

- A. Compiler Error
- B. Runtime Error
- C. Base A-intfcMethod
- D. None of the above.

Listing 3: Listing for Question 3.

```
    public class Ap133{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap133

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    A myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

3.145

Answer and Explanation (p. 296)

3.13.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 271) ?

- A. Compiler Error
- B. Runtime Error
- C. Base A-intfcMethod
- D. None of the above.

Listing 4: Listing for Question 4.

```
    public class Ap134{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap134

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    X myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

3.146

Answer and Explanation (p. 295)

3.13.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 273) ?

- A. Compiler Error
- B. Runtime Error
- C. A-intfcMethodX B-intfcMethodX
- D. None of the above.

Listing 5: Listing for Question 5.

```
    public class Ap135{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap135

class Worker{
  void doIt(){
    X myVar1 = new A();
    myVar1.intfcMethodX();
    X myVar2 = new B();
    myVar2.intfcMethodX();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B

interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 292)

3.13.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 275) ?

- A. Compiler Error
- B. Runtime Error
- C. A-intfcMethodX B-intfcMethodX
- D. None of the above.

Listing 6: Listing for Question 6.

```
    public class Ap136{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap136

class Worker{
  void doIt(){
    Object[] myArray = new Object[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      myArray[i].intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```



```
interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 289)

3.13.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 277) ?

- A. Compiler Error
- B. Runtime Error
- C. A-intfcMethodX B-intfcMethodX
- D. None of the above.

Listing 7: Listing for Question 7.

```
    public class Ap137{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap137

class Worker{
  void doIt(){
    Object[] myArray = new Object[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      ((X)myArray[i]).intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```



```
interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 288)

3.13.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 279) ?

- A. Compiler Error
- B. Runtime Error
- C. A-intfcMethodX B-intfcMethodX
- D. None of the above.

Listing 8: Listing for Question 8.

```
    public class Ap138{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap138

class Worker{
  void doIt(){
    X[] myArray = new X[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      myArray[i].intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```



```
interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 287)

3.13.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 281) ?

- A. Compiler Error
- B. Runtime Error
- C. Base A B
- D. None of the above.

Listing 9: Listing for Question 9.

```
    public class Ap139{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap139

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    myVar = new A();
    myVar.test();
    myVar = new B();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");
  }//end test()
}//end class Base

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base implements X,Y{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y
```

3.151

Answer and Explanation (p. 285)

3.13.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 283) ?

- A. Compiler Error
- B. Runtime Error
- C. Base A B B
- D. None of the above.

Listing 10: Listing for Question 10.

```
    public class Ap140{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap140

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.test();
    myVar1 = new A();
    myVar1.test();
    myVar1 = new B();
    myVar1.test();

    X myVar2 = (X)myVar1;
    myVar2.test();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");
  }//end test()
}//end class Base

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base implements X,Y{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y
```

Answer and Explanation (p. 284)

3.13.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.13.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0120: Self-assessment, Interfaces and polymorphic behavior
- File: Ap0120.htm
- Originally published: 2004
- Published at cnx.org: 12/08/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.13.6 Answers

3.13.6.1 Answer 10

C. Base A B B

3.13.6.1.1 Explanation 10

Expansion of the program from Question 9 (p. 280)

The class and interface definitions for the classes and interfaces named **Base** , **A** , **B** , **X** , and **Y** are the same as in Question 9 (p. 280) .

Call the test method differently

However, the call of the method named **test** in the object instantiated from the class named **B** is somewhat different. The difference is identified by the code in the following fragment.

NOTE:

```
  void doIt(){
Base myVar1 = new Base();
myVar1.test();
myVar1 = new A();
myVar1.test();
myVar1 = new B();
myVar1.test();

X myVar2 = (X)myVar1;
myVar2.test();

System.out.println("");
}//end doIt()
```

Calling test method on Base-type reference

In Question 9 (p. 280) , and in the above code fragment as well, the method named **test** was called on each of the objects using a reference stored in a reference variable of type **Base** .

Calling the overridden version of test method

This might be thought of as calling the overridden version of the method, through polymorphism, without regard for anything having to do with the interfaces.

Calling test method on interface-type reference

Then the code shown above calls the same method named **test** on one of the same objects using a reference variable of the interface type **X** .

Only one test method in each object

Keep in mind that each object defines only one method named **test** . This single method serves the dual purpose of overriding the method having the same signature from the superclass, and implementing a method with the same signature declared in each of the interfaces.

Implementing the interface method

Perhaps when the same method is called using a reference variable of the interface type, it might be thought of as implementing the interface method rather than overriding the method defined in the superclass. You can be the judge of that.

The same method is called regardless of reference type

In any event, in this program, the same method is called whether it is called using a reference variable of the superclass type, or using a reference variable of the interface type.

Illustrates the behavior of signature collisions

The purpose of this and Question 9 (p. 280) is not necessarily to illustrate a useful inheritance and implementation construct. Rather, these two questions are intended to illustrate the behavior of Java for the case of duplicated superclass and interface method signatures.

Back to Question 10 (p. 282)

3.13.6.2 Answer 9

C. Base A B

3.13.6.2.1 Explanation 9

A question regarding signature collisions

The question often arises in my classroom as to what will happen if a class inherits a method with a given signature and also implements one or more interfaces that declare a method with an identical signature.

The answer

The answer is that nothing bad happens, as long as the class provides a concrete definition for a method having that signature.

Only one method definition is allowed

Of course, only one definition can be provided for any given method signature, so that definition must satisfy the needs of overriding the inherited method as well as the needs of implementing the interfaces.

An example of signature collisions

The following fragment defines a class named **Base** that defines a method named **test** . The code also defines two interfaces named **X** and **Y** , each of which declares a method named **test** with an identical signature.

NOTE:

```
    class Base{
  public void test(){
    System.out.print("Base ");
  }//end test()
}//end class Base

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Classes A and B extend Base and implement X and Y

The code in the following fragment defines two classes, named **A** and **B** , each of which extends **Base** , and each of which implements both interfaces **X** and **Y** . Each class provides a concrete definition for the method named **test** , with each class providing a different definition.

NOTE:

```
    class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base implements X,Y{
  public void test(){
```

```
    System.out.print("B ");
  }//end test()
}//end class B
```

Override inherited method and define interface method

Each of the methods named **test** in the above fragment serves not only to override the method inherited from the class named **Base** , but also to satisfy the requirement to define the methods declared in the implemented interfaces named **X** and **Y** . *(This can also be thought of as overriding an inherited abstract method from an interface.)*

Store object's references as type Base and call test method

Finally, the code in the following fragment declares a reference variable of the type **Base** . Objects respectively of the classes **Base** , **A** , and **B** are instantiated and stored in the reference variable. Then the method named **test** is called on each of the references in turn.

NOTE:

```
  void doIt(){
Base myVar = new Base();
myVar.test();
myVar = new A();
myVar.test();
myVar = new B();
myVar.test();
System.out.println("");
}//end doIt()
}// end class Worker
```

As you probably expected, this causes the following text to appear on the screen:

NOTE:

```
Base A B
```

Back to Question 9 (p. 280)

3.13.6.3 Answer 8

C. A-intfcMethodX B-intfcMethodX

3.13.6.3.1 Explanation 8

Similar to previous two programs

This program is very similar to the programs in Question 6 (p. 274) and Question 7 (p. 276) . The program is Question 6 (p. 274) exposed a specific type mismatch problem. The program in Question 7 (p. 276) provided one solution to the problem.

A different solution

The following fragment illustrates a different solution to the problem.

NOTE:

```
  void doIt(){
X[] myArray = new X[2];
myArray[0] = new A();
```

```
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      myArray[i].intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
```

An array object of the interface type

In this case, rather than to declare the array object to be of type **Object**, the array is declared to be of the interface type **X**.

This is a less generic container than the one declared to be of type **Object**. Only references to objects instantiated from classes that implement the **X** interface, or objects instantiated from subclasses of those classes can be stored in the container. However, this is often adequate.

What methods can be called?

Since the references are stored as the interface type, any method declared in or inherited into the interface can be called on the references stored in the container. Of course, the objects referred to by those references must provide concrete definitions of those methods or the program won't compile.

(Although it isn't implicitly obvious, it is also possible to call any of the eleven methods defined in the ***Object*** *class on an object's reference being stored as an interface type. Those eleven methods can be called on any object, including array objects, regardless of how the references are stored.)*

Not the standard approach

If you are defining your own container, this is a satisfactory approach to implementation of the observer design pattern. However, you cannot use this approach when using containers from the standard collections framework, because those containers are designed to always store references as the generic type **Object**. In those cases, the casting solution of Question 7 (p. 276) *(or the use of generics)* is required.

Back to Question 8 (p. 278)

3.13.6.4 Answer 7

C. A-intfcMethodX B-intfcMethodX

3.13.6.4.1 Explanation 7

The correct use of an interface

This program illustrates the correct use of an interface. It uses a cast of the interface type in the following fragment to resolve the problem that was discussed at length in Question 6 (p. 274) earlier.

NOTE:

```
    void doIt(){
  Object[] myArray = new Object[2];
  myArray[0] = new A();
  myArray[1] = new B();

  for(int i=0;i<myArray.length;i++){
    ((X)myArray[i]).intfcMethodX();
  }//end for loop

  System.out.println("");
}//end doIt()
```

Back to Question 7 (p. 276)

3.13.6.5 Answer 6

A. Compiler Error

3.13.6.5.1 Explanation 6

What is a container?

The word container is often used in Java, with at least two different meaning. One meaning is to refer to the type of an object that is instantiated from a subclass of the class named **Container** . In that case, the object can be considered to be of type **Container** , and typically appears in a graphical user interface *(GUI)* . That is not the usage of the word in the explanation of this program.

A more generic meaning

In this explanation, the word container has a more generic meaning. It is common to store a collection of object references in some sort of Java container, such as an array object or a **Vector** object. In fact, there is a complete collections framework provided to facilitate that sort of thing *(**Vector** is one of the concrete classes in the Java Collections Framework)* .

Storing references as type Object

It is also common to declare the type of references stored in the container to be of the class **Object** . Because **Object** is a completely generic type, this means that a reference to any object instantiated from any class *(or any array object)* can be stored in the container. The standard containers such as **Vector** and **Hashtable** take this approach.

(Note that this topic became a little more complicated with the release of generics in jdk version 1.5.)

A class named Base and an interface named X

In a manner similar to several previous programs, this program defines a class named **Base** and an interface named **X** as shown in the following fragment.

NOTE:

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

interface X{
  public void intfcMethodX();
}//end X
```

Classes A and B extend Base and implement X

Also similar to previous programs, this program defines two classes named **A** and **B** . Each of these classes extends the class named **Base** and implements the interface named **X** , as shown in the next fragment.

NOTE:

```
    class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
```

```
                    "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```

Concrete definitions of the interface method

As before, these methods provide concrete definitions of the method named **intfcMethodX**, which is declared in the interface named **X**.

An array of references of type *Object*

The interesting portion of this program begins in the following fragment, which instantiates and populates a two-element array object *(container)* of type **Object**. *(In the sense of this discussion, an array object is a container, albeit a very simple one.)*

NOTE:

```
  void doIt(){
Object[] myArray = new Object[2];
myArray[0] = new A();
myArray[1] = new B();
```

Store object references of type *A* and B as type *Object*

Because the container is declared to be of type **Object**, references to objects instantiated from any class can be stored in the container. The code in the above fragment instantiates two objects, *(one of class **A** and the other of class **B**)*, and stores the two object's references in the container.

Cannot call interface method as type Object

The code in the **for** loop in the next fragment attempts to call the method named **intfcMethodX** on each of the two objects whose references are stored in the elements of the array.

NOTE:

```
    for(int i=0;i<myArray.length;i++){
  myArray[i].intfcMethodX();
}//end for loop

System.out.println("");
}//end doIt()
```

This produces the following compiler error under JDK 1.3:

NOTE:

```
    Ap136.java:24: cannot resolve symbol
symbol  : method intfcMethodX  ()
location: class java.lang.Object

myArray[i].intfcMethodX();
```

What methods can you call as type Object?

It is allowable to store the reference to an object instantiated from any class in a container of the type **Object** . However, the only methods that can be directly called *(without a cast and not using generics)* on that reference are the following eleven methods. These methods are defined in the class named **Object** :

- **clone** ()
- equals(Object obj)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()
- wait(long timeout)
- wait(long timeout,int nanos)

Overridden methods

Some, *(but not all)* , of the methods in the above list are defined with default behavior in the **Object** class, and are meant to be overridden in new classes that you define. This includes the methods named **equals** and **toString** .

Some of the methods in the above list, such as **getClass** , are simply utility methods, which are not meant to be overridden.

Polymorphic behavior applies

If you call one of these methods on an object's reference *(being stored as type Object)* , polymorphic behavior will apply. The version of the method overridden in, or inherited into, the class from which the object was instantiated will be identified and executed.

Otherwise, a cast is required

In order to call any method other than one of the eleven methods in the above list (p. 291) , *(on an object's reference being stored as type* ***Object*** *without using generics)* , you must cast the reference to some other type.

Casting to an interface type

The exact manner in which you write the cast will differ from one situation to the next. In this case, the problem can be resolved by rewriting the program using the interface cast shown in the following fragment.

NOTE:

```
  void doIt(){
Object[] myArray = new Object[2];
myArray[0] = new A();
myArray[1] = new B();

for(int i=0;i<myArray.length;i++){
  ((X)myArray[i]).intfcMethodX();
}//end for loop
```

```
    System.out.println("");
  }//end doIt()
```

The observer design pattern

By implementing an interface, and using a cast such as this, you can store references to many different objects, of many different actual types, each of which implements the same interface, but which have no required superclass-subclass relationship, in the same container. Then, when needed, you can call the interface methods on any of the objects whose references are stored in the container.

This is a commonly used design pattern in Java, often referred to as the observer design pattern.

Registration of observers

With this design pattern, none, one, or more observer objects, *(which implement a common observer interface)* are registered on an observable object. This means references to the observer objects are stored in a container by the observable object.

Making a callback

When the observable object determines that some interesting event has occurred, the observable object calls a specific interface method on each of the observer objects whose references are stored in the container.

The observer objects execute whatever behavior they were designed to execute as a result of having been notified of the event.

The model-view-control (MVC) paradigm

In fact, there is a class named **Observable** and an interface named **Observer** in the standard Java library. The purpose of these class and interface definitions is to make it easy to implement the observer design pattern.

(The Observer interface and the Observable class are often used to implement a programming style commonly referred to as the MVC paradigm.)

Delegation event model, bound properties of Beans, etc.

Java also provides other tools for implementing the observer design pattern under more specific circumstances, such as the Delegation Event Model, and in conjunction with bound and constrained properties in JavaBeans Components.

Back to Question 6 (p. 274)

3.13.6.6 Answer 5

C. A-intfcMethodX B-intfcMethodX

3.13.6.6.1 Explanation 5

More substantive use of an interface

This program illustrates a more substantive use of the interface than was the case in the previous programs.

The class named Base

The program defines a class named **Base** as shown in the following fragment.

NOTE:

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base
```

The interface named X

The program also defines an interface named **X** as shown in the next fragment. Note that this interface declares a method named **intfcMethodX** .

NOTE:

```
    interface X{
  public void intfcMethodX();
}//end X
```

Class A extends Base and implements X

The next fragment shows the definition of a class named **A** that extends **Base** and implements **X** .

NOTE:

```
    class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A
```

Defining interface method

Because the class named **A** implements the interface named **X** , it must provide a concrete definition of all the methods declared in **X** .

In this case, there is only one such method. That method is named **intfcMethodX** . A concrete definition for the method is provided in the class named **A** .

Class B also extends Base and implements X

The next fragment shows the definition of another class *(named* ***B***), which also extends **Base** and implements **X** .

NOTE:

```
    class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```

Defining the interface method

Because this class also implements **X** , it must also provide a concrete definition of the method named **intfcMethodX** .

Different behavior for interface method

However *(and this is extremely important)* , there is no requirement for this definition of the method to match the definition in the class named **A** , or to match the definition in any other class that implements **X** .

Only the method signature for the method named **intfcMethodX** is necessarily common among all the classes that implement the interface.

The definition of the method named **intfcMethodX** in the class named **A** is different from the definition of the method having the same name in the class named **B** .

The interesting behavior

The interesting behavior of this program is illustrated by the code in the following fragment.

NOTE:

```
  void doIt(){
X myVar1 = new A();
myVar1.intfcMethodX();
X myVar2 = new B();
myVar2.intfcMethodX();

System.out.println("");
}//end doIt()
```

Store object's references as interface type X

The code in the above fragment causes one object to be instantiated from the class named **A** , and another object to be instantiated from the class named **B** .

The two object's references are stored in two different reference variables, each declared to be of the type of the interface **X** .

Call the interface method on each reference

A method named **intfcMethodX** is called on each of the reference variables. Despite the fact that both object's references are stored as type **X** , the system selects and calls the appropriate method, *(as defined by the class from which each object was instantiated)* , on each of the objects. This causes the following text to appear on the screen:

NOTE:

```
A-intfcMethodX B-intfcMethodX
```

No subclass-superclass relationship exists

Thus, the use of an interface makes it possible to call methods having the same signatures on objects instantiated from different classes, without any requirement for a subclass-superclass relationship to exist among the classes involved.

In this case, the only subclass-superclass relationship between the classes named **A** and **B** was that they were both subclasses of the same superclass. Even that relationship was established for convenience, and was not a requirement.

Different behavior of interface methods

The methods having the same signature, *(declared in the common interface, and defined in the classes)* , need not have any similarity in terms of behavior.

A new interface relationship

The fact that both classes implemented the interface named **X** created a new relationship among the classes, which is not based on class inheritance.

Back to Question 5 (p. 272)

3.13.6.7 Answer 4

C. Base A-intfcMethod

3.13.6.7.1 Explanation 4

Illustrates the use of an interface as a type

The program defines a class named **Base** , and a class named **A** , which extends **Base** , and implements an interface named **X** , as shown below.

NOTE:

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Implementing interfaces

A class may implement none, one, or more interfaces.

The cardinal rule on interfaces

If a class implements one or more interfaces, that class must either be declared abstract, or it must provide concrete definitions of all methods declared in and inherited into all of the interfaces that it implements. If the class is declared abstract, its subclasses must provide concrete definitions of the interface methods.

A concrete definition of an interface method

The interface named **X** in this program declares a method named **intfcMethod** . The class named **A** provides a concrete definition of that method.

(The minimum requirement for a concrete definition is a method that matches the method signature and has an empty body.)

Storing object's reference as an interface type

The interesting part of the program is shown in the following code fragment.

NOTE:

```
    void doIt(){
  Base myVar1 = new Base();
  myVar1.inherMethod();
  X myVar2 = new A();
```

```
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
```

The above fragment instantiates a new object of the class named **A** , and saves a reference to that object in a reference variable of the declared type **X** .

How many ways can you save an object's reference?

Recall that a reference to an object can be held by a reference variable whose type matches any of the following:

- The class from which the object was instantiated.
- Any superclass of the class from which the object was instantiated.
- Any interface implemented by the class from which the object was instantiated.
- Any interface implemented by any superclass of the class from which the object was instantiated.
- Any superinterface of the interfaces mentioned above.

Save object's reference as implemented interface type

In this program, the type of the reference variable matches the interface named **X** , which is implemented by the class named **A** .

What does this allow you to do?

When a reference to an object is held by a reference variable whose type matches an interface implemented by the class from which the object was instantiated, that reference can be used to call any method declared in or inherited into that interface.

(That reference cannot be used to call methods not declared in or not inherited into that interface.)

In this simple case ...

The method named **intfcMethod** is declared in the interface named **X** and implemented in the class named **A** .

Therefore, the method named **intfcMethod** can be called on an object instantiated from the class named **A** when the reference to the object is held in a reference variable of the interface type.

*(The method could also be called if the reference is being held in a reference variable of declared type **A** .)*

The call to the method named **intfcMethod** causes the text **A-intfcMethod** to appear on the screen.

Back to Question 4 (p. 270)

3.13.6.8 Answer 3

C. Base A-intfcMethod

3.13.6.8.1 Explanation 3

What is runtime polymorphic behavior?

One way to describe runtime polymorphic behavior is:

The runtime system selects among two or more methods having the same signature, not on the basis of the type of the reference variable in which an object's reference is stored, but rather on the basis of the class from which the object was originally instantiated.

Illustrates simple class and interface inheritance

The program defines a class named **Base** , and a class named **A** , which extends **Base** , and implements the interface named **X** , as shown in the following fragment.

NOTE:

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Define an interface method

The interface named **X** declares a method named **intfcMethod** . A concrete definition of that method is defined in the class named **A** .

A new object of type Base

The code in the following fragment instantiates a new object of the class **Base** and calls its **inherMethod** . This causes the word **Base** to appear on the output screen. There is nothing special about this. This is a simple example of the use of an object's reference to call one of its instance methods.

NOTE:

```
      void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
```

A new object of type A

The following fragment instantiates a new object of the class **A** and calls its **intfcMethod** . This causes the text **A-intfcMethod** to appear on the output screen. There is also nothing special about this. This is also a simple example of the use of an object's reference to call one of its instance methods.

NOTE:

```
        A myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
```

Not polymorphic behavior

The fact that the class named **A** implements the interface named **X** does not indicate polymorphic behavior in this case. Rather, this program is an example of simple class and interface inheritance.

Interface type is not used

The program makes no use of the interface as a type, and exhibits no polymorphic behavior *(no decision among methods having the same signature is required)* .

The class named **A** inherits an abstract method named **intfcMethod** from the interface and must define it. *(Otherwise, it would be necessary to declare the class named* ***A*** *abstract.)*

The interface is not a particularly important player in this program.

Back to Question 3 (p. 269)

3.13.6.9 Answer 2

A. Compiler Error

3.13.6.9.1 Explanation 2

Simple hierarchical polymorphic behavior

This program is designed to test your knowledge of simple hierarchical polymorphic behavior.

Implement the interface named X

This program defines a class named **A** that extends a class named **Base** , and implements an interface named **X** , as shown in the following code fragment.

NOTE:

```
  class A extends Base implements X{
public void inherMethod(){
  System.out.print(
                  " A-inherMethod ");
}//end inherMethod()

public void intfcMethod(){
  System.out.print("A-intfcMethod ");
}//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Override and define some methods

The class named **A** overrides the method named **inherMethod** , which it inherits from the class named **Base** . It also provides a concrete definition of the method named **intfcMethod** , which is declared in the interface named **X** .

Store object's reference as superclass type

The program instantiates an object of the class named **A** and assigns that object's reference to a reference variable of type **Base** , as shown in the following code fragment.

NOTE:

```
        Base myVar2 = new A();
```

Oops! Cannot call this method

So far, so good. However, the next fragment shows where the program turns sour. It attempts to call the method named **intfcMethod** on the object's reference, which was stored as type **Base** .

NOTE:

```
myVar2.intfcMethod();
```

Polymorphic behavior doesn't apply here

Because the class named **Base** does not define the method named **intfcMethod**, hierarchical polymorphic behavior does not apply. Therefore a reference to the object being stored as type **Base** cannot be used to directly call the method named **intfcMethod**, and the program produces a compiler error.

What is the solution?

Hierarchical polymorphic behavior is possible only when the class defining the type of the reference *(or some superclass of that class)* contains a definition for the method that is called on the reference.

There are a couple of ways that downcasting could be used to solve the problem in this case.

Back to Question 2 (p. 267)

3.13.6.10 Answer 1

A. Compiler Error

3.13.6.10.1 Explanation 1

I put this question in here just to see if you are still awake.

Can store reference as interface type

A reference to an object instantiated from a class can be assigned to any reference variable whose declared type is the name of an interface implemented by the class from which the object was instantiated, or implemented by any superclass of that class.

Define two classes and an interface

This program defines a class named **A** that extends a class named **Base**. The class named **Base** extends **Object** by default.

The program also defines an interface named **X**.

Instantiate an object

The following statement instantiates an object of the class named **A**, and attempts to assign that object's reference to a reference variable whose type is the interface type named **X**.

NOTE:

```
X myVar2 = new A();
```

Interface **X** is defined but not implemented

None of the classes named **A**, **Base**, and **Object** implement the interface named **X**. Therefore, it is not allowable to assign a reference to an object of the class named **A** to a reference variable whose declared type is **X**. Therefore, the program produces the following compiler error under JDK 1.3:

NOTE:

```
    Ap131.java:20: incompatible types
found   : A
required: X
    X myVar2 = new A();
```

Back to Question 1 (p. 266)

-end-

3.14 Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions[18]

3.14.1 Table of Contents

3.14.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 309) to easily find and view the listings while you are reading about them.

3.14.3 Questions

3.14.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 301) ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Joe false
- D. Joe Joe true
- E. None of the above.

[18] This content is available online at <http://cnx.org/content/m45310/1.3/>.

Listing 1: Listing for Question 1.

```
    public class Ap141{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap141

class Worker{
  void doIt(){
    char[] anArray = {'J','o','e'};
    String Str1 = new String(anArray);
    String Str2 = new String(anArray);

    System.out.println(
              Str1 + " " + Str2 + " " +
              (Str1 == Str2));
  }//end doIt()
}// end class Worker
```

3.153

Answer and Explanation (p. 319)

3.14.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 302) ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Joe false
- D. Joe Joe true
- E. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap142{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap142

class Worker{
  void doIt(){
    char[] anArray = {'J','o','e'};
    String Str1 = new String(anArray);
    String Str2 = new String(anArray);

    System.out.println(
              Str1 + " " + Str2 + " " +
              Str1.equals(Str2));
  }//end doIt()
}// end class Worker
```

3.154

Answer and Explanation (p. 318)

3.14.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 303) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 3: Listing for Question 3.

```
    public class Ap143{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap143

class Worker{
  void doIt(){
    java.util.ArrayList ref =
            new java.util.ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                  (String)ref.get(0) +
                  (String)ref.get(1) +
                  (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

3.155

Answer and Explanation (p. 316)

3.14.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 304) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 4: Listing for Question 4.

```
    public class Ap144{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap144

class Worker{
  void doIt(){
    ArrayList ref =
            new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                  (String)ref.get(0) +
                  (String)ref.get(1) +
                  (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

3.156

Answer and Explanation (p. 315)

3.14.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 305) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 5: Listing for Question 5.

```
    import java.util.ArrayList;

public class Ap145{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap145

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref = new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                   (String)ref.get(0) +
                   (String)ref.get(1) +
                   (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

3.157

Answer and Explanation (p. 314)

3.14.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 306) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 6: Listing for Question 6.

```
    import java.util.ArrayList;

public class Ap146{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap146

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                   (String)ref.get(0) +
                   (String)ref.get(1) +
                   (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

3.158

Answer and Explanation (p. 313)

3.14.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 307) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 7: Listing for Question 7.

```
    public class Ap147{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap147

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref = new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");

    System.out.println(
                  (String)ref.get(0) +
                  (String)ref.get(1) +
                  (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

3.159

Answer and Explanation (p. 312)

3.14.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 308) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. None of the above.

Listing 8: Listing for Question 8.

```
    public class Ap148{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap148

class Worker{
  void doIt(){
    System.out.println(1.0/0);
  }//end doIt()
}// end class Worker
```

3.160

Answer and Explanation (p. 312)

3.14.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 308) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. None of the above.

Listing 9: Listing for Question 9.

```
    public class Ap149{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap149

class Worker{
  void doIt(){
    System.out.println(1/0);
  }//end doIt()
}// end class Worker
```

3.161

Answer and Explanation (p. 311)

3.14.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 309) ?

- A. Compiler Error
- B. Runtime Error
- C. AB CD EF
- D. None of the above.

Listing 10: Listing for Question 10.

```
    public class Ap150{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap150

class Worker{
  void doIt(){
    String[] ref = {"AB ","CD ","EF "};
    for(int i = 0; i <= 3; i++){
      System.out.print(ref[i]);
    }//end forloop
    System.out.println("");
  }//end doIt()
}// end class Worker
```

Answer and Explanation (p. 310)

3.14.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

- Listing 10 (p. 309) . Listing for Question 10.

3.14.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions
- File: Ap0130.htm
- Originally published: 2004
- Published at cnx.org: 12/18/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.14.6 Answers

3.14.6.1 Answer 10

Both of the following occur.
C. AB CD EF
B. Runtime Error

3.14.6.1.1 Explanation 10

Another index out of bounds

This is another example of a program that throws an index out of bounds exception. In this case, since the container is an array object, the name of the exception is **ArrayIndexOutOfBoundsException** .

Populate a three-element array object

The code in the following fragment creates and populates a three-element array object containing reference to three **String** objects.

NOTE:

```
  void doIt(){
String[] ref = {"AB ","CD ","EF "};
```

Access an out-of-bounds element

The next fragment attempts to access elements at indices 0 through 3 inclusive.

NOTE:

```
        for(int i = 0; i <= 3; i++){
      System.out.print(ref[i]);
    }//end forloop
```

Since index value 3 is outside the bounds of the array, the program throws the following exception and aborts:

NOTE:

```
    AB CD EF
java.lang.ArrayIndexOutOfBoundsException
 at Worker.doIt(Ap150.java:22)
 at Ap150.main(Ap150.java:14)
```

Note however that the program displays the contents of the three **String** objects referred to by the contents of the first three elements in the array before the problem occurs.

That's the way it often is with runtime errors. Often, a program will partially complete its task before getting into trouble and aborting with a runtime error.

Back to Question 10 (p. 309)

3.14.6.2 Answer 9

B. Runtime Error

3.14.6.2.1 Explanation 9

A setup

If you feel like you've been had, chances are you have been had. The purpose for Question 8 (p. 307) was to set you up for this question.

Division by zero for integer types

This program deals with the process of dividing by zero for **int** types. The code in the following fragment divides the **int** value 1 by the **int** value 0.

NOTE:

```
      void doIt(){
    System.out.println(1/0);
  }//end doIt()
```

Not the same as *double* divide by zero

However, unlike with type **double**, this process doesn't return a very large value and continue running. Rather, for type **int**, attempting to divide by zero will result in a runtime error of type **ArithmeticException** that looks something like the following under JDK 1.3:

NOTE:

```
    java.lang.ArithmeticException: / by zero
 at Worker.doIt(Ap149.java:20)
 at Ap149.main(Ap149.java:14)
```

An exercise for the student

I won't attempt to explain the difference in behavior for essentially the same problem between type **int** and type **double** . As the old saying goes, I'll leave that as an exercise for the student.

Back to Question 9 (p. 308)

3.14.6.3 Answer 8

C. Infinity

3.14.6.3.1 Explanation 8

A double *divide by zero* operation

This program deals with the process of dividing by zero for floating values of type **double** .

The following code fragment attempts to divide the double value 1.0 by the double value 0.

NOTE:

```
    void doIt(){
  System.out.println(1.0/0);
}//end doIt()
```

The program runs successfully, producing the output **Infinity** .

What is Infinity?

Suffice it to say that Infinity is a very large number.

(Any value divided by zero is a very large number.)

At this point, I'm not going to explain it further. If you are interested in learning what you can do with **Infinity** , see the language specifications.

Back to Question 8 (p. 307)

3.14.6.4 Answer 7

B. Runtime Error

3.14.6.4.1 Explanation 7

This program illustrates an **IndexOutOfBounds** exception.

Instantiate and populate an ArrayList object

By now, you will be familiar with the kind of container object that you get when you instantiate the **ArrayList** class.

The code in the following fragment instantiates such a container, having an initial capacity of one element.

Then it adds two elements to the container. Each element is a reference to an object of the class **String** .

NOTE:

```
    void doIt(){
ArrayList ref = null;
ref = new ArrayList(1);
ref.add("ABC ");
ref.add("DEF ");
```

Increase capacity automatically

Because two elements were successfully added to a container having an initial capacity of only one element, the container was forced to increase its capacity automatically.

Following execution of the code in the above fragment, **String** object references were stored at index locations 0 and 1 in the **ArrayList** object.

Get reference at index location 2

The next fragment attempts to use the **get** method to fetch an element from the container at index value 2.

Index values in an **ArrayList** object begin with zero. Therefore, since only two elements were added to the container in the earlier fragment, there is no element at index value 2.

NOTE:

```
System.out.println(
          (String)ref.get(0) +
          (String)ref.get(1) +
          (String)ref.get(2));
```

An IndexOutOfBounds exception

As a result, the program throws an **IndexOutOfBounds** exception. The error produced under JDK 1.3 looks something like the following:

NOTE:

```
    Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index: 2, Size: 2
at java.util.ArrayList.RangeCheck
  (Unknown Source)
at java.util.ArrayList.get
  (Unknown Source)
at Worker.doIt(Ap147.java:27)
at Ap147.main(Ap147.java:16)
```

Attempting to access an element with a negative index value would produce the same result.

An ArrayIndexOutOfBounds exception

A similar result occurs if you attempt to access an element in an ordinary array object outside the bounds of the index values determined by the size of the array. However, in that case, the name of the exception is **ArrayIndexOutOfBounds** .

Back to Question 7 (p. 306)

3.14.6.5 Answer 6

B. Runtime Error

3.14.6.5.1 Explanation 6

The infamous *NullPointerException*

Interestingly, one of the first things that you read when you start reading Java books, is that there are *no pointers in Java* . It is likely that shortly thereafter when you begin writing, compiling, and executing simple Java programs, one of your programs will abort with an error message looking something like that **shown below** :

NOTE:

```
    Exception in thread "main" java.lang.NullPointerException
        at
Worker.doIt(Ap146.java:23)
        at
Ap146.main(Ap146.java:16)
```

What is a NullPointerException?

Stated simply, a **NullPointerException** occurs when you attempt to perform some operation on an object using a reference that doesn't refer to an object.

That is the case in this program

The following code fragment declares a local reference variable and initializes its value to ***null*** .

NOTE:

```
  void doIt(){
ArrayList ref = null;
```

(A reference variable in Java must either refer to a valid object, or specifically refer to no object (null). Unlike a pointer in C and C++, a Java reference variable cannot refer to something arbitrary.)

In this case, null means that the reference variable doesn't refer to a valid object.

No ArrayList object

Note that the code in the above fragment does not instantiate an object of the class **ArrayList** and assign that object's reference to the reference variable.

(The reference variable doesn't contain a reference to an object instantiated from the class named ***ArrayList*** *, or an object instantiated from any class for that matter.)*

Call a method on the reference

However, the code in the next fragment attempts to add a **String** object's reference to a nonexistent **ArrayList** object by calling the **add** method on the reference containing null.

NOTE:

```
    ref.add("ABC ");
```

This results in the **NullPointerException** shown earlier (p. 313) .

What can you do with a null reference?

The only operation that you can perform on a reference variable containing null is to assign an object's reference to the variable. Any other attempted operation will result in a **NullPointerException** .

Back to Question 6 (p. 305)

3.14.6.6 Answer 5

C. ABC DEF GHI

3.14.6.6.1 Explanation 5

The purpose of this program is to

- Continue to illustrate the use of java packages, and
- Illustrate the use of the Java import directive.

Program contains an import directive

This program is the same as the program in Question 4 (p. 303) with a major exception. Specifically, the program contains the *import directive* shown in the following fragment.

NOTE:

```
import java.util.ArrayList;
```

A shortcut

The designers of Java recognized that having to type a fully-qualified name for every reference to a class in a Java program can become burdensome. Therefore, they provided us with a shortcut that can be used, so long as we don't need to refer to two or more class files having the same name.

Import directives

The shortcut is called an import directive.

As can be seen above, the import directive consists of the word *import* followed by the fully-qualified name of a class file that will be used in the program.

A program may have more than one import directive, with each import directive specifying the location of a different class file.

The import directive(s) must appear before any class or interface definitions in the source code.

The alternative wild-card syntax

An alternative form of the import directive replaces the name of the class with an asterisk.

The asterisk behaves as a wild-card character. It tells the compiler to use any class file that it finds in that package that matches a class reference in the source code.

The wild-card form should be used with care, because it can sometimes cause the compiler to use a class file that is different from the one that you intended to use *(if it finds the wrong one first)* .

Class file name collisions

If your source code refers to two different class files having the same name, you must forego the use of the import directive and provide fully-qualified names for those class files.

Back to Question 5 (p. 304)

3.14.6.7 Answer 4

A. Compiler Error

3.14.6.7.1 Explanation 4

The purpose of this program is to continue to illustrate the use of java packages.

No fully-qualified class names

This program is the same as the program in Question 3 (p. 302) with a major exception. Neither of the references to the **ArrayList** class use fully-qualified names in this program. Rather, the references are as shown in the following fragment.

NOTE:

```
ArrayList ref =
    new ArrayList(1);
```

Compiler errors

As a result, the JDK 1.3 compiler produces two error messages similar to the following:

NOTE:

```
    Ap144.java:20: cannot resolve symbol
symbol  : class ArrayList
location: class Worker
    ArrayList ref =
```

Doesn't know how to find the class file

This error message indicates that the compiler didn't know where to look on the disk to find the file named **ArrayList.class**

Back to Question 4 (p. 303)

3.14.6.8 Answer 3

C. ABC DEF GHI

3.14.6.8.1 Explanation 3

Illustrate the use of java packages

Since it was necessary to make use of a class to illustrate packages, this program also previews the use of the **ArrayList** class. We will be very interested in this class later when we study Java data containers.

What is an ArrayList object?

Some of this terminology may not make much sense to you at this point, but I'll go ahead and tell you anyway, just as a preview.

According to Sun, the **ArrayList** class provides a

*"Resizable-array implementation of the **List** interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the **List** interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to **Vector** , except that it is unsynchronized.)"*

Stated more simply ...

Stated more simply, an object of the **ArrayList** class can be used as a replacement for an array object. An **ArrayList** object knows how to increase its capacity on demand, whereas the capacity of a simple array object cannon change once it is instantiated.

An ArrayList object

The following statement instantiates a new object of the **ArrayList** class, with an initial capacity for one element. The initial capacity is determined by the **int** value passed to the constructor when the object is instantiated.

NOTE:

```
java.util.ArrayList ref =
    new java.util.ArrayList(1);
```

Back to the primary purpose ...

Getting back to the primary purpose of this program, what is the meaning of the term ***java.util*** that appears ahead of the name of the class, **ArrayList** ?

Avoiding name conflicts

One of the age-old problems in computer programming has to do with the potential for name conflicts. The advent of OOP and reusable code didn't cause that problem to go away. If anything, it made the problem worse.

For example, you and I may work as programmers for separate companies named X and Y. A company named Z may purchase our two companies and attempt to merge the software that we have written separately. Given that there are only a finite number of meaningful class names, there is a good possibility that you and I may have defined different classes with the same names. Furthermore, it may prove useful to use both of the class definitions in a new program.

Put class files in different directories

Sun's solution to the problem is to cause compiled class files to reside in different directories. Simplifying things somewhat, if your compiled file for a class named **Joe** is placed in a directory named **X** , and my compiled file for a different class named **Joe** is placed in a directory named **Y** , then source code in

the same Java program can refer to those two class files as **X.Joe** and **Y.Joe** . This scheme makes it possible for the Java compiler and the Java virtual machine to distinguish between the two files having the name **Joe.class** .

The java and util directories

Again, simplifying things slightly, the code in the above fragment refers to a file named **ArrayList.class** , which is stored in a directory named **util** , which is a subdirectory of a directory named **java** .

The directory named java is the root of a directory tree containing a very large number of standard Java class files.

(As an aside, there is another directory named javax, which forms the root of another directory tree containing class files considered to be extensions to the standard class library.)

Many directories *(packages)*

Stated simply, a Java package is nothing more or less than a directory containing class files.

The standard and extended Java class libraries are scattered among a fairly large number of directories or packages *(a quick count of the packages in the JDK 1.3 documentation indicates that there are approximately 65 standard and extended packages)* .

A fully-qualified class name

With one exception, whenever you refer to a class in a Java program, you must provide a fully-qualified name for the class, including the path through the directory tree culminating in the name of the class. Thus, the following is the fully-qualified name for the class whose name is **ArrayList** .

java.util.ArrayList

(Later we will see another way to accomplish this that requires less typing effort.)

The exception

The one exception to the rule is the use of classes in the *java.lang* package, *(such as* ***Boolean*** , ***Class*** , *and* ***Double*** *)* . Your source code can refer to classes in the **java.lang** package without the requirement to provide a fully-qualified class name.

An ArrayList object

Now back to the use of the object previously instantiated from the class named **ArrayList** . This is the kind of object that is often referred to as a container.

(A container in this sense is an object that is used to store references to other objects.)

Many methods available

An object of the **ArrayList** class provides a variety of methods that can be used to store object references and to fetch the references that it contains.

The *add* method

One of those methods is the method named **add** .

The following code fragment instantiates three objects of the **String** class, and stores them in the **ArrayList** object instantiated earlier.

(Note that since the initial capacity of the ***ArrayList*** *object was adequate to store only a single reference, the following code causes the object to automatically increase its capacity to at least three.)*

NOTE:

```
        ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");
```

The get() method

The references stored in an object of the **ArrayList** class can be fetched by calling the **get** method on a reference to the object passing a parameter of type **int** .

The code in the following fragment calls the **get** method to fetch the references stored in index locations 0, 1, and 2. These references are passed to the **println** method, where the contents of the **String** objects referred to by those references are concatenated and displayed on the computer screen.

NOTE:

```
System.out.println(
          (String)ref.get(0) +
          (String)ref.get(1) +
          (String)ref.get(2));
```

The output

This results in the following being displayed:

ABC DEF GHI

Summary

The above discussion gave you a preview into the use of containers in general, and the **ArrayList** container in particular.

However, the primary purpose of this program was to help you to understand the use of packages in Java.

The **ArrayList** class was simply used as an example of a class file that is stored in a standard Java package.

Back to Question 3 (p. 302)

3.14.6.9 Answer 2

D. Joe Joe true

3.14.6.9.1 Explanation 2

Two String objects with identical contents

As in Question 1 (p. 300) , the program instantiates two **String** objects containing identical character strings, as shown in the following code fragment.

NOTE:

```
    char[] anArray = {'J','o','e'};
String Str1 = new String(anArray);
String Str2 = new String(anArray);
```

Compare objects for equality

Also, as in Question 1 (p. 300) , this program compares the two objects for equality and displays the result as shown by the call to the **equals** method in the following fragment.

NOTE:

```
System.out.println(
      Str1 + " " + Str2 + " " +
      Str1.equals(Str2));
```

Compare using overridden equals method

The == operator is not used to compare the two objects in this program. Instead, the objects are compared using an overridden version of the **equals** method. In this case, the **equals** method returns true, indicating that the objects are of the same type and contain the same data values.

The equals method

The **equals** method is defined in the **Object** class, and can be overridden in subclasses of **Object** . It is the responsibility of the author of the subclass to override the method so as to implement that author's concept of "equal" insofar as objects of the class are concerned.

The overridden *equals* method

The reason that the **equals** method returned true in this case was that the author of the **String** class provided an overridden version of the **equals** method.

The default *equals* method

If the author of the class does not override the **equals** method, and the default version of the **equals** method inherited from **Object** is called on an object of the class, then according to Sun:

"for any reference values x and y, this method returns true if and only if x and y refer to the same object (x==y has the value true)"

In other words, the default version of the **equals** method inherited from the class **Object** provides the same behavior as the == operator when applied to object references.

Back to Question 2 (p. 301)

3.14.6.10 Answer 1

C. Joe Joe false

3.14.6.10.1 Explanation 1

The identity operator

This program illustrates the behavior of the == operator *(sometimes referred to as the identity operator)* when used to compare references to objects.

Two String objects with identical contents

As shown in the following fragment, this program instantiates two objects of the **String** class containing identical character strings.

NOTE:

```
   class Worker{
 void doIt(){
   char[] anArray = {'J','o','e'};
   String Str1 = new String(anArray);
   String Str2 = new String(anArray);
```

The fact that the two **String** objects contain identical character strings is confirmed by:

- Both objects are instantiated using the same array object of type **char** as input.
- When the **toString** representations of the two objects are displayed later, the display of each object produces Joe on the computer screen.

Compare object references using identity (==)

The references to the two **String** objects are compared using the == operator, and the result of that comparison is displayed. This comparison will produce either true or false. The code to accomplish this comparison is shown in the following fragment.

NOTE:

```
System.out.println(
      Str1 + " " + Str2 + " " +
      (Str1 == Str2));
```

The statement in the above fragment produces the following display:

Joe Joe false

How can this be false?

We know that the two objects are of the same type (**String**) and that they contain the same character strings. Why does the == operator return false?

Doesn't compare the objects

The answer lies in the fact that the above statement doesn't really compare the two objects at all. Rather, it compares the values stored in the reference variables referring to the two objects. That is not the same as comparing the objects.

References are not equal

Even though the objects are of the same type and contain the same character string, they are two different objects, located in different parts of memory. Therefore, the contents of the two reference variables containing references to the two objects are not equal.

The correct answer is *false*

The == operator returns ***false*** as it should. The only way that the == operator could return ***true*** is if both reference variables refer to the same object, *(which is not the case)* .

The bottom line is ...

The == operator cannot be used to compare two objects for equality. However, it can be used to determine if two reference variables refer to the same object.

Back to Question 1 (p. 300)

-end-

3.15 Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references[19]

3.15.1 Table of Contents

3.15.2 Preface

This module is part of a self-assessment test designed to help you determine how much you know about object-oriented programming using Java.

The test consists of a series of questions with answers and explanations of the answers.

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back.

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 329) to easily find and view the listings while you are reading about them.

3.15.3 Questions

3.15.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 321) ?

[19] This content is available online at <http://cnx.org/content/m45302/1.3/>.

- A. Compiler Error
- B. Runtime Error
- C. OK OK
- D. OK
- E. None of the above.

Listing 1: Listing for Question 1.

```
    public class Ap151{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap151

class Worker{
  void doIt(){
    Object refA = new MyClassA();
    Object refB =
              (Object)(new MyClassB());
    System.out.print(refA);
    System.out.print(refB);
    System.out.println("");
  }//end doIt()
}// end class Worker

class MyClassA{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

3.163

Answer and Explanation (p. 339)

3.15.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 322) ?

- A. Compiler Error

- B. Runtime Error
- C. OK OK
- D. OK
- E. None of the above.

Listing 2: Listing for Question 2.

```
    public class Ap152{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap152

class Worker{
  void doIt(){
    Object ref1 = new MyClassA();
    Object ref2 = new MyClassB();
    System.out.print(ref1);

    MyClassB ref3 = (MyClassB)ref1;
    System.out.print(ref3);
    System.out.println("");
  }//end doIt()
}// end class Worker

class MyClassA{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

3.164

Answer and Explanation (p. 338)

3.15.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 323) ?

- A. Compiler Error

- B. Runtime Error
- C. OK
- D. None of the above.

Listing 3: Listing for Question 3.

```
    import java.util.Random;
import java.util.Date;

public class Ap153{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap153

class Worker{
  void doIt(){
    Random ref = new Random(
                 new Date().getTime());
    if(ref.nextBoolean()){
      throw new IllegalStateException();
    }else{
      System.out.println("OK");
    }//end else
  }//end doIt()
}// end class Worker
```

3.165

Answer and Explanation (p. 336)

3.15.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 324) ?

- A. Compiler Error
- B. Runtime Error
- C. 5 10 15
- D. None of the above.

Listing 4: Listing for Question 4.

```
    import java.util.NoSuchElementException;
public class Ap154{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap154

class Worker{
  void doIt(){
    MyContainer ref =
                     new MyContainer();
    ref.put(0,5);
    ref.put(1,10);
    ref.put(2,15);

    System.out.print(ref.get(0)+" ");
    System.out.print(ref.get(1)+" ");
    System.out.print(ref.get(2)+" ");
    System.out.print(ref.get(3)+" ");

  }//end doIt()
}// end class Worker

class MyContainer{
  private int[] array = new int[3];

  public void put(int idx, int data){
    if(idx > (array.length-1)){
      throw new
              NoSuchElementException();
    }else{
      array[idx] = data;
    }//end else
  }//end put()

  public int get(int idx){
    if(idx > (array.length-1)){
      throw new
              NoSuchElementException();
    }else{
      return array[idx];
    }//end else
  }//end put()

}//end class MyContainer
```

3.166

Answer and Explanation (p. 334)

3.15.3.5 Question 5

The source code in Listing 5 (p. 325) is contained in a single file named Ap155.java
What output is produced by the program?

- A. Compiler Error
- B. Runtime Error
- C. OK
- D. None of the above.

Listing 5: Listing for Question 5.

```
    public class Ap155{
  public static void main(
                        String args[]){
    new Ap155a().doIt();
  }//end main()
}//end class Ap155

public class Ap155a{
  void doIt(){
    System.out.println("OK");
  }//end doIt()
}// end class Ap155a
```

Answer and Explanation (p. 334)

3.15.3.6 Question 6

A Java application consists of the two source files shown in Listing 6 (p. 326) and Listing 7 (p. 326) having names of AP156.java and AP156a.java
What output is produced by this program?

- A. Compiler Error
- B. Runtime Error
- C. OK
- D. None of the above.

Listing 6: Listing for Question 6.

```
    public class Ap156{
  public static void main(
                        String args[]){
    new Ap156a().doIt();
  }//end main()
}//end class Ap156
```

3.168

Listing 7: Listing for Question 6.

```
    public class Ap156a{
  void doIt(){
    System.out.println("OK");
  }//end doIt()
}// end class Ap156a
```

3.169

Answer and Explanation (p. 333)

3.15.3.7 Question 7

Explain the purpose of the terms @param and @return in Listing 8 (p. 327) . Also explain any of the other terms that make sense to you.

Listing 8: Listing for Question 7.

```
    public class Ap157{

/**
* Returns the character at the
* specified index. An index ranges from
* <code>0</code> to
* <code>length() - 1</code>.
*
* @param  index  index of desired
* character.
* @return  the desired character.
*/
  public char charAt(int index) {
    //Note, this method is not intended
    // to be operational.  Rather, it
    // ...
    return 'a';//return dummy char
  }//end charAt method
}//end class
```

3.170

Answer and Explanation (p. 332)

3.15.3.8 Question 8

What output is produced by the program shown in Listing 9 ?

- A. Compiler Error
- B. Runtime Error
- C. Tom
- D. None of the above.

Answer and Explanation (p. 331)

3.15.3.9 Question 9

What output is produced by the program shown in Listing 10 (p. 328) ?

- A. Compiler Error
- B. Runtime Error
- C. Tom
- D. None of the above.

Listing 10: Listing for Question 9.

```
    public class Ap159{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap159

class Worker{
  void doIt(){
    char[] ref = null;
    System.out.print(ref);
    System.out.print(" ");
    ref[0] = 'T';
    ref[1] = 'o';
    ref[2] = 'm';
    System.out.println(ref);
  }//end doIt()
}// end class Worker
```

3.171

Answer and Explanation (p. 331)

3.15.3.10 Question 10

What output is produced by the program shown in Listing 11 (p. 329) ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Tom
- D. None of the above.

Listing 11: Listing for Question 10.

```
    public class Ap160{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap160

class Worker{
  void doIt(){
    char[] ref = {'J','o','e'};
    System.out.print(ref);
    System.out.print(" ");
    ref[0] = 'T';
    ref[1] = 'o';
    ref[2] = 'm';
    System.out.println(ref);
  }//end doIt()
}// end class Worker
```

3.172

Answer and Explanation (p. 330)

3.15.4 Listings

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.15.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references
- File: Ap0140.htm
- Originally published: 2004
- Published at cnx.org: 12/18/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.15.6 Answers

3.15.6.1 Answer 10

C. Joe Tom

3.15.6.1.1 Explanation 10

This is an upgrade to the program from Question 9 (p. 327) .

Success at last

The code in the following fragment resolves the compilation problem from Question 8 and the runtime problem from Question 9 (p. 327) .

NOTE:

```
  void doIt(){
char[] ref = {'J','o','e'};
System.out.print(ref);
System.out.print(" ");
ref[0] = 'T';
ref[1] = 'o';
ref[2] = 'm';
System.out.println(ref);
}//end doIt()
```

Simply initializing the local reference variable named **ref** satisfies the compiler, making it possible to compile the program.

Initializing the local reference variable named **ref** with a reference to a valid array object eliminates the NullPointerException that was experienced in Question 9 (p. 327) .

Printing the contents of the array object

The print statement passes the reference variable to the **print** method. The **print** method finds that the reference variable refers to a valid object *(instead of containing null as was the case in Question 9* (p. 327)) and behaves accordingly.

The **print** statement causes the initialized contents of the array object to be displayed. Then those contents are replaced with a new set of characters. The **println** statement causes the new characters to be displayed.

Back to Question 10 (p. 328)

3.15.6.2 Answer 9

B. Runtime Error

3.15.6.2.1 Explanation 9

Purposely initializing a local variable

This is an update to the program from Question 8 (p. 327) . The code in the following fragment solves the compilation problem identified in Question 8 (p. 327) .

NOTE:

```
  void doIt(){
char[] ref = null;
```

In particular, initializing the value of the reference variable named **ref** satisfies the compiler and makes it possible to compile the program.

A *NullPointerException*

However, there is still a problem, and that problem causes a runtime error.

The following statement attempts to use the reference variable named **ref** to print something on the screen. This results, among other things, in an attempt to call the **toString** method on the reference. However, the reference doesn't refer to an object. Rather, it contains the value ***null*** .

NOTE:

```
System.out.print(ref);
```

The result is a runtime error with the following infamous **NullPointerException** message appearing on the screen:

NOTE:

```
   java.lang.NullPointerException
 at java.io.Writer.write(Writer.java:107)
 at java.io.PrintStream.write(PrintStream.java:245)
 at java.io.PrintStream.print(PrintStream.java:396)
 at Worker.doIt(Ap159.java:22)
 at Ap159.main(Ap159.java:15)
```

Back to Question 9 (p. 327)

3.15.6.3 Answer 8

A. Compiler Error

3.15.6.3.1 Explanation 8

Garbage in, garbage out

Earlier programming languages, notably C and C++ allowed you to inadvertently write programs that process the garbage left in memory by previous programs running there. This happens when the C or C++ programmer fails to properly initialize variables, allowing them to contain left-over garbage from memory.

Member variables are automatically initialized to default values

That is not possible in Java. All member variables in a Java object are automatically initialized to a default value if you don't write the code to initialize them to some other value.

Local variables are not automatically initialized

Local variables are not automatically initialized. However, your program will not compile if you write code that attempts to fetch and use a value in a local variable that hasn't been initialized or had a value assigned to it.

Print an uninitialized local variable

The statement in the following code fragment attempts to fetch and print a value using the uninitialized local variable named **ref** .

NOTE:

```
  void doIt(){
char[] ref;
System.out.print(ref);
```

As a result, the program refuses to compile, displaying the following error message under JDK 1.3.

NOTE:

```
Ap158.java:23: variable ref might not have been initialized
System.out.print(ref);
```

Back to Question 8 (p. 327)

3.15.6.4 Answer 7

See explanation below.

3.15.6.4.1 Explanation 7

The javadoc.exe program

When you download the JDK from Oracle, you receive a program named **javadoc.exe** in addition to several other programs.

The purpose of the **javadoc** program is to help you document the Java programs that you write. You create the documentation by running the **javadoc** program and specifying your source file or files as a command-line parameter. For example, you can generate documentation for this program by entering the following at the command line.

NOTE:

```
javadoc Ap157.java
```

Produces HTML files as output

This will produce a large number of related HTML files containing documentation for the class named **Ap157** . The primary HTML file is named **Ap157.html** . A file named **index.html** is also created.

This file can be opened in a browser to provide a viewer for all of the information contained in the many related HTML files.

(As a labor saving device, you can also specify a group of input files to the javadoc program, using wildcard characters as appropriate, to cause the program to produce documentation files for each of the input files in a single run.)

Special documentation comments and directives

If you include comments in your source code that begin with

/**

and end with

*/

they will be picked up by the javadoc program and become part of the documentation.

In addition to comments, you can also enter a variety of special directives to the javadoc program as shown in the following program.

NOTE:

```
    public class Ap157{

/**
* Returns the character at the
* specified index. An index ranges from
* <code>0</code> to
* <code>length() - 1</code>.
*
* @param  index  index of desired
* character.
* @return  the desired character.
*/
  public char charAt(int index) {
    //Note, this method is not intended
    // to be operational.  Rather, it
    // is intended solely to illustrate
    // the generation of javadoc
    // documentation for the parameter
    // and the return value.
    return 'a';//return dummy char
  }//end charAt method
}//end class
```

The @param and @return directives

The **@param** and **@return** directives in the source code shown above are used by the **javadoc** program for documenting information about parameters passed to and information returned from the method named **charAt** . The method definition follows the special javadoc comment.

Back to Question 7 (p. 326)

3.15.6.5 Answer 6

C. OK

3.15.6.5.1 Explanation 6

Public classes in separate files

This program meets the requirement identified in Question 5 (p. 325) . In particular, this program defines two public classes. The source code for each public class is stored in a separate file. Thus, the program compiles and executes successfully, producing the text OK on the screen.

Back to Question 6 (p. 325)

3.15.6.6 Answer 5

A. Compiler Error

3.15.6.6.1 Explanation 5

Public classes in separate files

Java requires that the source code for every public class be contained in a separate file. In this case, the source code for two public classes was contained in a single file. The following compiler error was produced by JDK 1.3:

NOTE:

```
    Ap155.java:18: class Ap155a is public, should be declared in a file
named Ap155a.java
public class Ap155a{
```

Back to Question 5 (p. 325)

3.15.6.7 Answer 4

This program produces both of the following:

- C. 5 10 15
- B. Runtime Error

3.15.6.7.1 Explanation 4

The NoSuchElementException

This program defines, creates, and uses a very simple container object for the purpose of illustrating the **NoSuchElementException** .

The code in the following fragment shows the beginning of a class named **MyContainer** from which the container object is instantiated.

NOTE:

```
    class MyContainer{
  private int[] array = new int[3];

  public void put(int idx, int data){
    if(idx > (array.length-1)){
      throw new
              NoSuchElementException();
    }else{
      array[idx] = data;
    }//end else
  }//end put()
```

A wrapper for an array object

This class is essentially a wrapper for a simple array object of type **int** . An object of the class provides a method named **put** , which can be used to store an **int** value into the array. The **put** method receives two parameters. The first parameter specifies the index of the element where the value of the second parameter is to be stored.

Throw NoSuchElementException on index out of bounds

The **put** method tests to confirm that the specified index is within the positive bounds of the array. If not, it uses the **throw** keyword to throw an exception of the type **NoSuchElementException** . Otherwise, it stores the incoming data value in the specified index position in the array.

(Note that a negative index will cause an ArrayIndexOutOfBoundsException instead of a NoSuchElementException to be thrown.)

The get method

An object of the **MyContainer** class also provides a **get** method that can be used to retrieve the value stored in a specified index.

NOTE:

```
  public int get(int idx){
if(idx > (array.length-1)){
  throw new
          NoSuchElementException();
}else{
  return array[idx];
}//end else
}//end put()
```

The **get** method also tests to confirm that the specified index is within the positive bounds of the array. If not, it throws an exception of the type **NoSuchElementException** . Otherwise, it returns the value stored in the specified index of the array.

(As noted earlier, a negative index will cause an ArrayIndexOutOfBoundsException instead of a NoSuchElementException to be thrown.)

The NoSuchElementException

Thus, this container class illustrates the general intended purpose of the **NoSuchElementException** .

Instantiate and populate a container

The remainder of the program simply exercises the container. The code in the following fragment instantiates a new container, and uses the **put** method to populate each of its three available elements with the values 5, 10, and 15.

NOTE:

```
  void doIt(){
MyContainer ref =
                new MyContainer();
ref.put(0,5);
ref.put(1,10);
ref.put(2,15);
```

Get and display the data in the container

Then the code in the next fragment uses the **get** method to get and display the values in each of the three elements, causing the following text to appear on the screen:

5 10 15

NOTE:

```
        System.out.print(ref.get(0)+" ");
    System.out.print(ref.get(1)+" ");
    System.out.print(ref.get(2)+" ");
```

One step too far

Finally, the code in the next fragment goes one step too far and attempts to get a value from index 3, which is outside the bounds of the container.

NOTE:

```
        System.out.print(ref.get(3)+" ");
```

This causes the **get** method of the container object to throw a **NoSuchElementException** . The program was not designed to handle this exception, so this causes the program to abort with the following text showing on the screen:

NOTE:

```
    5 10 15 java.util.NoSuchElementException
 at MyContainer.get(Ap154.java:49)
 at Worker.doIt(Ap154.java:30)
 at Ap154.main(Ap154.java:15)
```

(Note that the values of 5, 10, and 15 were displayed on the screen before the program aborted and displayed the error message.)

Back to Question 4 (p. 323)

3.15.6.8 Answer 3

This program can produce either of the following depending on the value produced by a random boolean value generator:

- B. Runtime Error
- C. OK

3.15.6.8.1 Explanation 3

Throwing an exception

This program illustrates the use of the **throw** keyword to throw an exception.

*(Note that the **throw** keyword is different from the **throws** keyword.)*

Throw an exception if random boolean value is true

A random **boolean** value is obtained. If the value is true, the program throws an **IllegalStateException** and aborts with the following message on the screen:

NOTE:

```
    java.lang.IllegalStateException
 at Worker.doIt(Ap153.java:29)
 at Ap153.main(Ap153.java:20)
```

If the random **boolean** value is false, the program runs to completion, displaying the text OK on the screen.

Instantiate a Random object

The following code fragment instantiates a new object of the **Random** class and stores the object's reference in a reference variable named **ref** .

NOTE:

```
  void doIt(){
Random ref = new Random(
             new Date().getTime());
```

I'm not going to go into a lot of detail about the **Random** class. Suffice it to say that an object of this class provides methods that will return a pseudo random sequence of values upon successive calls. You might think of this object as a random value generator.

Seeding the random generator

The constructor for the class accepts a **long** integer as the seed for the sequence.

*(Two **Random** objects instantiated using the same seed will produce the same sequence of values.)*

In this case, I obtained the time in milliseconds, relative to January 1, 1970, as a **long** integer, and provided that value as the seed. Thus, if you run the program two times in succession, with a time delay of at least one millisecond in between, the random sequences will be different.

Get a random boolean value

The code in the next fragment calls the **nextBoolean** method on the **Random** object to obtain a random boolean value. *(Think of this as tossing a coin with true on one side and false on the other side.)*

NOTE:

```
  if(ref.nextBoolean()){
throw new IllegalStateException();
```

Throw an exception

If the **boolean** value obtained in the above fragment is true, the code instantiates a new object of the **IllegalStateException class** , and uses the **throw** keyword to throw an exception of this type.

Program aborts

The program was not designed to gracefully handle such an exception. Therefore the program aborts, displaying the error message shown earlier.

Don't throw an exception

The code in the next fragment shows that if the **boolean** value tested above is false, the program will display the text OK and run successfully to completion.

NOTE:

```
  }else{
System.out.println("OK");
}//end else
}//end doIt()
```

You may need to run the program several times to see both possibilities.

Back to Question 3 (p. 322)

3.15.6.9 Answer 2

The answer is both of the following:

- D. OK
- B. Runtime Error

3.15.6.9.1 Explanation 2

One cast is allowable ...

It is allowable, but not necessary, to cast the type of an object's reference toward the root of the inheritance hierarchy.

It is also allowable to cast the type of an object's reference along the inheritance hierarchy toward the actual class from which the object was instantiated.

Another cast is not allowable ...

However, *(excluding interface type casts)* , it is not allowable to cast the type of an object's reference in ways that are not related in a subclass-superclass inheritance sense. For example, you cannot cast the type of an object's reference to the type of a sibling of that object.

Two sibling classes

The code in the following fragment defines two simple classes named **MyClassA** and **MyClassB** . By default, each of these classes extends the class named **Object** . Therefore, neither is a superclass of the other. Rather, they are siblings.

NOTE:

```
    class MyClassA{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

Instantiate one object from each sibling class

The code in the next fragment instantiates one object from each of the above classes, and stores references to those objects in reference variables of type Object.

Then the code causes the overridden **toString** method of one of the objects to be called by passing that object's reference to the **print** method.

NOTE:

```
      void doIt(){
    Object ref1 = new MyClassA();
    Object ref2 = new MyClassB();
    System.out.print(ref1);
```

The code in the above fragment causes the text OK to appear on the screen.

Try to cast to a sibling class type

At this point, the reference variable named **ref1** holds a reference to an object of type **MyClassA** . The reference is being held as type **Object** .

The statement in the next fragment attempts to cast that reference to type **MyClassB** , which is a sibling of the class named **MyClassA** .

NOTE:

```
MyClassB ref3 = (MyClassB)ref1;
```

A ClassCastException

The above statement causes a **ClassCastException** to be thrown, which in turn causes the program to abort. The screen output is shown below:

NOTE:

```
   OK java.lang.ClassCastException:MyClassA
 at Worker.doIt(Ap152.java:24)
 at Ap152.main(Ap152.java:14)
```

(Note that the text OK appeared on the screen before the program aborted and displayed diagnostic information on the screen.)

Back to Question 2 (p. 321)

3.15.6.10 Answer 1

C. OK OK

3.15.6.10.1 Explanation 1

Type conversion

This program illustrates type conversion up the inheritance hierarchy, both with and without a cast.

Store object's reference as type Object

The following fragment instantiates a new object of the class named **MyClassA** , and stores that object's reference in a reference variable of type **Object** . This demonstrates that you can store an object's reference in a reference variable whose type is a superclass of the class from which the object was instantiated, with no cast required.

NOTE:

```
   class Worker{
 void doIt(){
   Object refA = new MyClassA();
```

Cast object's reference to type Object

The code in the next fragment instantiates an object of the class named **MyClassB** , and stores the object's reference in a reference variable of type **Object** , after first casting the reference to type **Object** . This, and the previous fragment demonstrate that while it is allowable to cast a reference to the superclass type before storing it in a superclass reference variable, such a cast is not required.

NOTE:

```
Object refB =
      (Object)(new MyClassB());
```

Type conversion and assignment compatibility

This is part of a larger overall topic commonly referred to as type conversion. It also touches the fringes of something that is commonly referred to as assignment-compatibility.

Automatic type conversions

Some kinds of type conversions happen automatically. For example, you can assign a value of type **byte** to a variable of type **int** and the type conversion will take place automatically.

Cast is required for narrowing conversions

However, if you attempt to assign a value of type **int** to a variable of type **byte** , the assignment will not take place automatically. Rather, the compiler requires you to provide a cast to confirm that you accept responsibility for the conversion, which in the case of **int** to byte could result in the corruption of data.

Automatic conversions up the inheritance hierarchy

When working with objects, type conversion takes place automatically for conversions toward the root of the inheritance hierarchy. Therefore, conversion from any class type to type **Object** happen automatically. However, conversions in the direction away from the root require a cast.

(Conversion from any class type to any superclass of that class also happens automatically.)

Polymorphic behavior

The code in the next fragment uses polymorphic behavior to display the contents of the two String objects.

NOTE:

```
        System.out.print(refA);
    System.out.print(refB);
```

No cast required

This works without the use of a cast because the **print** method calls the **toString** method on any object's reference that it receives as an incoming parameter. The **toString** method is defined in the **Object** class, and overridden in the **String** class. Polymorphic behavior dictates that in such a situation, the version of the method belonging to the object will be called regardless of the type of the reference variable holding the reference to the object.

When would a cast be required?

Had the program attempted to call a method on the reference that is not defined in the **Object** , class, it would have been necessary to cast the reference down the inheritance hierarchy in order to successfully call the method.

Back to Question 1 (p. 320)

-end-

Chapter 4

Programming Fundamentals

4.1 Jb0103 Preface to Programming Fundamentals[1]

4.1.1 Table of Contents

4.1.2 Welcome

Welcome to Programming Fundamentals.

This group of modules under the heading *Programming Fundamentals* is not part of a formal course of study at Austin Community College [2] where I teach. Instead, it is a compilation of material that I have published over the years for the benefit of those students who desire to enroll in ITSE 2321 Object Oriented Programming [3] but who don't have the required prerequisite knowledge for that course.

If you fall in that category, or if you just want to get a good introduction to computer programming, you may find this material useful.

Even if you have completed a programming fundamentals course in another language, or you have considerable programming experience in another language, you may find this material useful as an introduction to the Java programming language and its syntax.

Most of the topics are divided into two modules – a primary module and a review module. The review modules contain review questions and answers keyed to the material in the primary modules.

As you work your way through the modules in this group, you should prepare yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [4]
- Java OOP: Objects and Encapsulation [5]

[1] This content is available online at <http://cnx.org/content/m45179/1.6/>.
[2] http://www.austincc.edu/
[3] http://cnx.org/content/m45222
[4] http://cnx.org/content/m44148
[5] http://cnx.org/content/m44153

4.1.3 The DrJava IDE and the Java Development K it

In order to work with the material in this group of *Programming Fundamentals* modules, you will need access to Oracle's Java Development Kit [6] *(JDK)* . You will also need access to a text editor, preferably one that is tailored to the creation of Java programs. One such freely available text editor is named DrJava [7] .

However, DrJava is more than just a text editor. It is an Integrated Development Environment *((IDE)* that is designed for use by students learning how to program in the Java programming language. I recommend it for use with this group of *Programming Fundamentals* modules.

See A Quick Start Guide to DrJava [8] for instructions on downloading and installing both the DrJava IDE and Oracle's Java Development Kit *(JDK)* .

The Quick Start Guide also provides instructions for using the DrJava IDE.

4.1.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0103 Preface to Programming Fundamentals
- File: Jb0103.htm
- Published: 11/22/12
- Revised: 05/28/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.2 Jb0105: Java OOP: Similarities and Differences between Java and C++ [9]

4.2.1 Table of Contents

[6] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[7] http://www.drjava.org/
[8] http://www.drjava.org/docs/quickstart/index.html
[9] This content is available online at <http://cnx.org/content/m45142/1.2/>.

4.2.2 Preface

This module, which presents some of the similarities and differences between Java and C++, is provided solely for the benefit of those students who are already familiar with C++ and are making the transition from C++ into Java.

If you have some familiarity with C++, you may find the material in this module helpful. If not, simply skip this module and move on to the next module in the collection.

In general, students in Prof. Baldwin's Java/OOP courses are not expected to have any specific knowledge of C++.

This module is intended to be general in nature. Therefore, although a few update notes were added prior to publication at cnx.org, no significant effort has been made to keep it up to date relative to any particular version of the Java JDK or any particular version of C++. Changes have occurred in both Java and C++ since the first publication of this document in 1997. Those changes may not be reflected in this module.

4.2.3 Similarities and differences

This list of similarities and differences is based heavily on The Java Language Environment, A White Paper [10] by James Gosling and Henry McGilton and *Thinking in Java* by Bruce Eckel, which was freely available on the web when this document was first published.

Java does not support **typedefs**, **defines**, or a **preprocessor**. Without a preprocessor, there are no provisions for including header files.

Since Java does not have a preprocessor there is no concept of **#define** macros or *manifest constants*. However, the declaration of named constants is supported in Java through use of the **final** keyword.

Java does not support **enums** but, as mentioned above, does support *named constants*. *(Note: the enum type [11] was introduced into Java sometime between the first publication of this document and Java version 7.)*

Java supports *classes*, but does not support *structures* or *unions*.

All stand-alone C++ programs require a function named **main** and can have numerous other functions, including both stand-alone functions and functions that are members of a class. There are no stand-alone functions in Java. Instead, there are only functions that are members of a class, usually called methods. However, a Java application *(not a Java applet)* does require a class definition containing a **main** method.

Global functions and global data are not allowed in Java. However, variables that are declared **static** are shared among all objects instantiated from the class in which the **static** variables are declared. *(Generally, static has a somewhat different meaning in C++ and Java. For example, the concept of a static local variable does not exist in Java as it does in C++.)*

All classes in Java ultimately inherit from the class named **Object**. This is significantly different from C++ where it is possible to create inheritance trees that are completely unrelated to one another. All Java objects contain the eleven methods that are inherited from the **Object** class.

All function or method definitions in Java are contained within a class definition. To a C++ programmer, they may look like inline function definitions, but they aren't. Java doesn't allow the programmer to request that a function be made inline, at least not directly.

Both C++ and Java support class *(static)* methods or functions that can be called without the requirement to instantiate an object of the class.

The **interface** keyword in Java is used to create the equivalence of an abstract base class containing only method declarations and constants. No variable data members or method definitions are allowed in a Java interface definition. *(True abstract base classes can also be created in Java.)* The interface concept is not supported by C++ but can probably be emulated.

Java does not support multiple class inheritance. To some extent, the **interface** feature provides the desirable features of multiple class inheritance to a Java program without some of the underlying problems.

[10] http://net.uom.gr/Books/Manuals/langenviron-a4.pdf
[11] http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

While Java does not support multiple class inheritance, single inheritance in Java is similar to C++, but the manner in which you implement inheritance differs significantly, especially with respect to the use of constructors in the inheritance chain.

In addition to the access modifiers applied to individual members of a class, C++ allows you to provide an additional access modifier when inheriting from a class. This latter concept is not supported by Java.

Java does not support the **goto** statement *(but goto is a reserved word)* . However, it does support labeled **break** and **continue** statements, a feature not supported by C++. In certain restricted situations, labeled **break** and **continue** statements can be used where a **goto** statement might otherwise be used.

Java does not support **operator overloading** .

Java does not support automatic type conversions *(except where guaranteed safe)* .

Unlike C++, Java has a **String** type, and objects of this type are immutable *(cannot be modified)* . *(Note, although I'm not certain, I believe that the equivalent of a Java String type was introduced into C++ sometime after the original publication of this document.)*

Quoted strings are automatically converted into **String** objects in Java. Java also has a **StringBuffer** type. Objects of this type can be modified, and a variety of string manipulation methods are provided.

Unlike C++, Java provides true arrays as first-class objects. There is a length member, which tells you how big the array is. An exception is thrown if you attempt to access an array out of bounds. All arrays are instantiated in dynamic memory and assignment of one array to another is allowed. However, when you make such an assignment, you simply have two references to the same array. Changing the value of an element in the array using one of the references changes the value insofar as both references are concerned.

Unlike C++, having two "pointers" or references to the same object in dynamic memory is not necessarily a problem *(but it can result in somewhat confusing results)* . In Java, dynamic memory is reclaimed automatically, but is not reclaimed until all references to that memory become NULL or cease to exist. Therefore, unlike in C++, the allocated dynamic memory cannot become invalid for as long as it is being referenced by any reference variable.

Java does not support **pointers** *(at least it does not allow you to modify the address contained in a pointer or to perform pointer arithmetic)* . Much of the need for pointers was eliminated by providing types for arrays and strings. For example, the oft-used C++ declaration **char* ptr** needed to point to the first character in a C++ null-terminated "string" is not required in Java, because a string is a true object in Java.

A class definition in Java looks similar to a class definition in C++, but there is no closing semicolon. Also *forward reference declarations* that are sometimes required in C++ are not required in Java.

The scope resolution operator (::) required in C++ is not used in Java. The dot is used to construct all fully-qualified references. Also, since there are no pointers, the pointer operator (->) used in C++ is not required in Java.

In C++, static data members and functions are called using the name of the class and the name of the static member connected by the scope resolution operator. In Java, the dot is used for this purpose.

Like C++, Java has primitive types such as **int** , **float** , etc. Unlike C++, the size of each primitive type is the same regardless of the platform. There is no unsigned integer type in Java. Type checking and type requirements are much tighter in Java than in C++.

Unlike C++, Java provides a true **boolean** type. *(Note, the C++ equivalent of the Java boolean type may have been introduced into C++ subsequent to the original publication of this document.)*

Conditional expressions in Java must evaluate to **boolean** rather than to integer, as is the case in C++. Statements such as

if(x+y)...

are not allowed in Java because the conditional expression doesn't evaluate to a **boolean** .

The **char** type in C++ is an 8-bit type that maps to the ASCII *(or extended ASCII)* character set. The **char** type in Java is a 16-bit type and uses the Unicode character set *(the Unicode values from 0 through 127 match the ASCII character set)* . For information on the Unicode character set see http://www.unicode.org/ [12] .

[12] http://www.unicode.org/

Unlike C++, the ≫ operator in Java is a "signed" right bit shift, inserting the sign bit into the vacated bit position. Java adds an operator that inserts zeros into the vacated bit positions.

C++ allows the instantiation of variables or objects of all types either at compile time in static memory or at run time using dynamic memory. However, Java requires all variables of primitive types to be instantiated at compile time, and requires all objects to be instantiated in dynamic memory at runtime. Wrapper classes are provided for all primitive types to allow them to be instantiated as objects in dynamic memory at runtime if needed.

C++ requires that classes and functions be declared before they are used. This is not necessary in Java.

The "namespace" issues prevalent in C++ are handled in Java by including everything in a class, and collecting classes into packages.

C++ requires that you re-declare static data members outside the class. This is not required in Java.

In C++, unless you specifically initialize variables of primitive types, they will contain garbage. Although local variables of primitive types can be initialized in the declaration, primitive data members of a class cannot be initialized in the class definition in C++.

In Java, you can initialize primitive data members in the class definition. You can also initialize them in the constructor. If you fail to initialize them, they will be initialized to zero *(or equivalent)* automatically.

Like C++, Java supports constructors that may be overloaded. As in C++, if you fail to provide a constructor, a default constructor will be provided for you. If you provide a constructor, the default constructor is not provided automatically.

All objects in Java are passed by reference, eliminating the need for the copy constructor used in C++.

(In reality, all parameters are passed by value in Java. However, passing a copy of a reference variable makes it possible for code in the receiving method to access the object referred to by the variable, and possibly to modify the contents of that object. However, code in the receiving method cannot cause the original reference variable to refer to a different object.)

There are no destructors in Java. Unused memory is returned to the operating system by way of a *garbage collector* , which runs in a different thread from the main program. This leads to a whole host of subtle and extremely important differences between Java and C++.

Like C++, Java allows you to overload functions *(methods)* . However, default arguments are not supported by Java.

Unlike C++, Java does not support templates. Thus, there are no generic functions or classes. *(Note, generics similar to C++ templates were introduced into Java in version 5 subsequent to the original publication of this document.)*

Unlike C++, several "data structure" classes are contained in the "standard" version of Java. *(Note, the Standard Template Library was introduced into the C++ world subsequent to the original publication of this document.)*

More specifically, several "data structure" classes are contained in the standard class library that is distributed with the Java Development Kit (JDK). For example, the standard version of Java provides the containers **Vector** and **Hashtable** that can be used to contain any object through recognition that any object is an object of type **Object** . However, to use these containers, you must perform the appropriate upcasting and downcasting, which may lead to efficiency problems. *(Note, the upcasting and downcasting requirements were eliminated in conjunction with the introduction of "generics" into Java mentioned earlier.)*

Multithreading is a standard feature of the Java language.

Although Java uses the same keywords as C++ for access control: **private** , **public** , and **protected** , the interpretation of these keywords is significantly different between Java and C++.

There is no **virtual** keyword in Java. All non-static methods use dynamic binding, so the virtual keyword isn't needed for the same purpose that it is used in C++.

Java provides the **final** keyword that can be used to specify that a method cannot be overridden and that it can be statically bound. *(The compiler may elect to make it inline in this case.)*

The detailed implementation of the exception handling system in Java is significantly different from that in C++.

Unlike C++, Java does not support operator overloading. However, the (+) and (+=) operators are automatically overloaded to concatenate strings, and to convert other types to string in the process.

As in C++, Java applications can call functions written in another language. This is commonly referred to as *native methods* . However, applets cannot call native methods.

Unlike C++, Java has built-in support for program documentation. Specially written comments can be automatically stripped out using a separate program named **javadoc** to produce program documentation.

Generally Java is more robust than C++ due to the following:

- Object handles *(references)* are automatically initialized to null.
- Handles are checked before accessing, and exceptions are thrown in the event of problems.
- You cannot access an array out of bounds.
- The potential for memory leaks is prevented *(or at least greatly reduced)* by automatic garbage collection.

4.2.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0105: Java OOP: Similarities and Differences between Java and C++
- File: Jb0105.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.3 Jb0110: Java OOP: Programming Fundamentals, Getting Started[13]

4.3.1 Table of Contents

[13] This content is available online at <http://cnx.org/content/m45137/1.3/>.

4.3.2 Preface

4.3.2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.

This module explains how to get started programming using the Java programming language.

4.3.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [14] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [15] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.3.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.3.2.3.1 Listings

[14] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[15] http://download.oracle.com/javase/7/docs/api/

4.3.3 Writing, compiling, and running Java programs

4.3.3.1 Writing Java code

Writing Java code is straightforward. You can write Java code using any plain text editor. You simply need to cause the output file to have an extension of .java.

There are a number of high-level *Integrated Development Environments (IDEs)* available, such as Eclipse and NetBeans, but they tend to be overkill for the relatively simple Java programs described in these modules.

There are also some low-level IDEs available, such as JCreator and DrJava, which are very useful. I normally use a free version of JCreator, mainly because it contains a color-coded editor.

So, just find an editor that you are happy with and use it to write your Java code.

4.3.3.2 Preparing to compile and run Java code

Perhaps the most complicated thing is to get your computer set up for compiling and running Java code in the first place.

4.3.3.2.1 Downloading the java development kit *(JDK)*

You will need to download and install the free Java JDK from the Oracle/Sun website. As of November, 2012, you will find that website at http://www.oracle.com/technetwork/java/javase/downloads/index.html [16]

There is a 64-bit version of the JDK, but I haven't tried it yet because my home computer won't can't support it. I am still using the 32-bit version. However, I suspect that the 64-bit version will work just fine if you have a computer that supports it.

Whether you elect to use the 32-bit or 64-bit version is strictly up to you. Either of them should do the job very nicely.

4.3.3.2.2 Installing the JDK

As of November 2012, you will find installation instructions at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html [17] .

I strongly recommend that you read the instructions and pay particular attention to the information having to do with setting the **path** environment variable.

A word of caution

If you happen to be running Windows Vista or Windows 7, you may need to use something like the following when updating the PATH Environment Variable

```
... ;C:\Program Files (x86)\Java\jdk1.6.0_26\bin
```

in place of

```
... ;C:\Program Files\Java\jdk1.7.0\bin
```

as shown in the installation instructions.

I don't have any experience with any Linux version. Therefore, I don't have any hints to offer there.

[16] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[17] http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html

4.3.3.2.3 The JDK documentation

It is very difficult to program in Java without access to the documentation for the JDK.

Several different types of Java documentation are available online at http://www.oracle.com/technetwork/java/javase/documentation/index.html [18] .

Specific documentation for classes, methods, etc., is available online at http://download.oracle.com/javase/7/docs/api/ [19] .

It is also possible to download the documentation and install it locally if you have room on your disk. The download links for JDK 6 and JDK 7 documentation are also shown on the page at http://www.oracle.com/technetwork/java/javase/downloads/index.html [20] .

4.3.3.3 Compiling and running Java code

There are a variety of ways to compile and run Java code. The way that I will describe here is the most basic and, in my opinion, the most reliable. These instructions apply to a Windows operating system. If you are using a different operating system, you will need to translate the instructions to your operating system.

4.3.3.3.1 Write your Java program

Begin by using your text editor to write your Java program into one or more text files, each with an extension of .java. *(Files of this type are often referred to as source code files.)* Save the source code files in an empty folder somewhere on your disk. Make sure that the name of the **class** containing the **main** method *(which you will learn about in a future module)* matches the name of the file in which that class is contained *(except for the extension of .java on the file name, which does not appear in the class name)* .

4.3.3.3.2 Create a batch file

Use your text editor to create a batch file *(or whatever the equivalent is for your operating system)* containing the text shown in Listing 1 (p. 349) *(with the modifications discussed below)* and store it in the same folder as your Java source code files..

Then execute the batch file, which in turn will execute the program if there are no compilation errors.

Listing 1: Windows batch file.

```
echo off
cls

del *.class

javac -cp .; hello.java
java -cp .; hello

pause
```

4.1

---

[18] http://www.oracle.com/technetwork/java/javase/documentation/index.html
[19] http://download.oracle.com/javase/7/docs/api/
[20] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Comments regarding the batch file

The commands in the batch file of Listing 1 (p. 349) will

- Open a command-line screen for the folder containing the batch file.
- Delete all of the compiled class files from the folder. *(If the folder doesn't contain any class files, this will be indicated on the command-line screen.)*
- Attempt to compile the program in the file named **hello.java.**
- Attempt to run the compiled program using a compiled Java file named **hello.class** .
- Pause and wait for you to dismiss the command-line screen by pressing a key on the keyboard.

If errors occur, they will be reported on the command-line screen and the program won't be executed.

If your program is named something other than **hello** , *(which it typically would be)* substitute the new name for the word **hello** where it appears twice in the batch file.

Don't delete the pause command

The **pause** command causes the command-line window to stay on the screen until you dismiss it by pressing a key on the keyboard. You will need to examine the contents of the window if there are errors when you attempt to compile and run your program, so don't delete the pause command.

Translate to other operating systems

The format of the batch file in Listing 1 (p. 349) is a Windows format. If you are using a different operating system, you will need to translate the information in Listing 1 (p. 349) into the correct format for your operating system.

4.3.3.3.3 A test program

The test program in Listing 2 (p. 350) can be used to confirm that Java is properly installed on your computer and that you can successfully compile and execute Java programs.

Listing 2: A test program.

```
    class hello {
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//end class
```

4.2

Instructions

Copy the code shown in Listing 2 (p. 350) into a text file named **hello.java** and store in an empty folder somewhere on your disk.

Create a batch file named **hello.bat** containing the text shown in Listing 1 (p. 349) and store that file in the same folder as the file named **hello.java** .

Execute the batch file.

If everything is working, a command-line screen should open and display the following text:

```
    Hello World
Press any key to continue . . .
```

Congratulations

If that happens, you have just written, compiled and executed your first Java program.

Oops

If that doesn't happen, you need to go back to the installation instructions and see if you can determine why the JDK isn't properly installed.

If you get an error message similar to the following, that probably means that you didn't set the **path** environment variable correctly.

```
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

Beyond that, I can't provide much advice in the way of troubleshooting hints.

4.3.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0110: Java OOP: Programming Fundamentals, Getting Started
- File: Jb0110.htm
- Published: 11/16/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.4 Jb0110r Review[21]

4.4.1 Table of Contents

4.4.2 Preface

This module contains review questions and answers keyed to the module titled Jb0110: Java OOP: Programming Fundamentals, Getting Started [22] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.4.3 Questions

4.4.3.1 Question 1 .

True or false? You need a special IDE to write Java code.

Answer 1 (p. 355)

4.4.3.2 Question 2

True or false? All of the software that you need to create, compile, and run Java programs is free.

Answer 2 (p. 355)

4.4.3.3 Question 3

True or false? Installing the Java JDK can be a little difficult.

Answer 3 (p. 355)

4.4.3.4 Question 4

True or false? Java is so easy that you don't need documentation to program using Java.

Answer 4 (p. 355)

4.4.3.5 Question 5

True or false? The most fundamental way to compile and run Java applications is from the command line.

Answer 5 (p. 355)

4.4.3.6 Question 6

Write a simple test program that can be used to confirm that the JDK is properly installed on your system.

Answer 6 (p. 354)

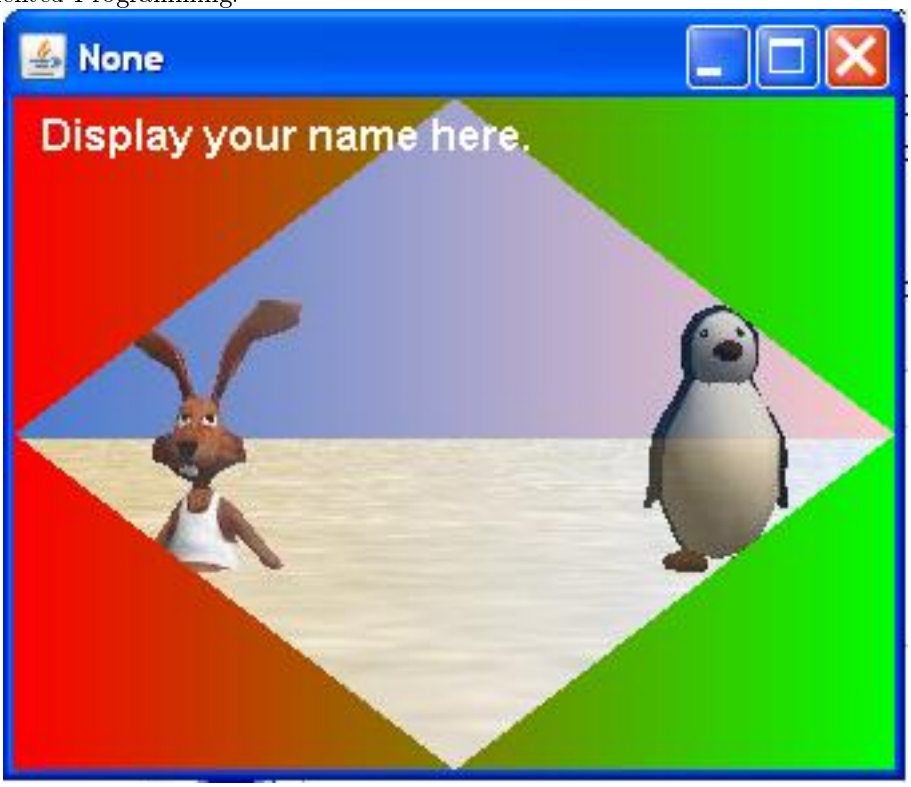
[21] This content is available online at <http://cnx.org/content/m45162/1.5/>.
[22] http://cnx.org/content/m45137

4.4.4 Listings

- Listing 1 (p. 354) . A Java test program.

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



Here is another image that was inserted for the same purpose – to insert space between the questions and the answers.

4.4.5 Answers

4.4.5.1 Answer 6

If you can compile and run the program code shown in Listing 1 (p. 354) , the JDK is probably installed properly on your computer.

Listing 1: A Java test program.

```
    class hello {
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//end class
```

4.3

Back to Question 6 (p. 352)

4.4.5.2 Answer 5

True. Although a variety of IDEs are available that can be used to compile and run Java applications, the most fundamental way is to compile and run the programs from the command line. A batch file in Windows, or the equivalent in other operating systems, can be of some help in reducing the amount of typing required to compile and run a Java application from the command line.

Back to Question 5 (p. 352)

4.4.5.3 Answer 4

False. Java uses huge class libraries, which few if any of us can memorize. Therefore, it is very difficult to program in Java without access to the documentation for the JDK.

As of November 2012, several different types of Java documentation are available online at http://www.oracle.com/technetwork/java/javase/documentation/index.html [23] .

Back to Question 4 (p. 352)

4.4.5.4 Answer 3

True. Installing the Java JDK can be a little difficult depending on your experience and knowledge. As of November 2012, you will find installation instructions at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html [24] .

Back to Question 3 (p. 352)

4.4.5.5 Answer 2

True. You will need to download and install the **free** Java JDK from the Oracle/Sun website. As of November, 2012, you will find that website at http://www.oracle.com/technetwork/java/javase/downloads/index.html [25]

Back to Question 2 (p. 352)

4.4.5.6 Answer 1

False. You can write Java code using any plain text editor. You simply need to cause the output file to have an extension of .java.

Back to Question 1 (p. 352)

4.4.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0110r Review for Programming Fundamentals, Getting Started
- File: Jb0110r.htm
- Published: 11/20/12
- Revised: 01/02/13

[23] http://www.oracle.com/technetwork/java/javase/documentation/index.html
[24] http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html
[25] http://www.oracle.com/technetwork/java/javase/downloads/index.html

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.5 Jb0115: Java OOP: First Program[26]

4.5.1 Table of Contents

4.5.2 Preface

The purpose of this module is to present the first complete Java program of the collection that previews the most common forms of the three pillars of procedural programming:

- sequence
- selection
- loop

The program also illustrates

- calling a method,
- passing a parameter to the method, and
- receiving a returned value from the method.

As mentioned above, this is simply a preview. Detailed discussions of these topics will be presented in future modules.

[26] This content is available online at <http://cnx.org/content/m45220/1.2/>.

4.5.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

4.5.2.1.1 Images

- Image 1 (p. 358) . Program output.

4.5.2.1.2 Listings

- Listing 1 (p. 360) . Source code for FirstProgram.

4.5.3 Discussion

4.5.3.1 Instructions for compiling and running the program

Assuming that the Java Development Kit (JDK) is properly installed on your computer *(see Jb0110: Java OOP: Programming Fundamentals, Getting Started* [27] *)*, do the following to compile and run this program.

1. Copy the text from Listing 1 (p. 360) into a text file named **FirstProgram.java** and store the file in a folder on your disk.
2. Open a command-line window in the folder containing the file.
3. Type the following command at the prompt to compile the program:
   javac FirstProgram.java
4. Type the following command at the prompt to run the program:
   java FirstProgram

4.5.3.2 Comments

Any text in the program code that begins with // is a comment. The compiler will ignore everything from the // to the end of the line.

Comments were inserted into the program code to explain the code.

The compiler also ignores blank lines.

Note that this program was designed to illustrate the concepts while being as non-cryptic as possible.

4.5.3.3 Program output

The program should display the text shown in Image 1 (p. 358) on the screen except that the time will be different each time you run the program.

[27] http://cnx.org/content/m45137

Image 1: Program output.

```
value in = 5
Odd time = 1353849164875
countA = 0
countA = 1
countA = 2
countB = 0
countB = 1
countB = 2
value out = 10
```

4.4

4.5.4 Run the program

I encourage you to copy the code from Listing 1 (p. 360) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.5.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0115: Java OOP: First Program
- File: Jb0115.htm
- Published: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

4.5.6 Complete program listing

A complete listing of the program follows.

Listing 1: Source code for FirstProgram.

```
/* Begin block comment
This is the beginning of a block comment in Java.
Everything in this block comment is for human consumption
and will be ignored by the Java compiler.

File: FirstProgram.java
Copyright 2012, R.G. Baldwin

This program is designed to illustrate the most common
forms of the three pillars of procedural programming in
Java code:

sequence
selection
loop

The program also illustrates calling a method, passing
a parameter to the method, and receiving a returned
value from the method.

Assuming that the Java Development Kit (JDK) is properly
installed on your computer, do the following to compile
and run this program.

1. Copy this program into a file named FirstProgram.java
and store the file in a folder on your disk.
2. Open a command-line window in the folder containing
the file.
3. Type the following command to compile the program:
javac FirstProgram.java
4.4. Type the following command to run the program:
java FirstProgram

Any text that begins with // in the following program
code is a comment. The compiler will ignore everything
from the // to the end of the line.

The compiler also ignores blank lines.

Note that this program was designed to illustrate the
concepts while being as non-cryptic as possible.

The program should display the following text on the
screen except that the time will be different each time
that you run the program.

value in = 5
Odd time = 1353849164875
countA = 0
countA = 1
countA = 2
countB = 0
countB = 1
countB = 2
```

-end-

4.6 Jb0120: Java OOP: A Gentle Introduction to Java Programming[28]

4.6.1 Table of Contents

4.6.2 Preface

4.6.2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.

It provides a gentle introduction to Java programming.

4.6.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [29] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [30] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.6.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

[28] This content is available online at <http://cnx.org/content/m45138/1.2/>.
[29] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[30] http://download.oracle.com/javase/7/docs/api/

4.6.2.3.1 Images

- Image 1 (p. 364) . A checkout counter algorithm.

4.6.2.3.2 Listings

- Listing 1 (p. 365) . Program named Memory01.
- Listing 2 (p. 366) . Batch file for Memory01.

4.6.3 Discussion and sample code

4.6.3.1 Introduction

All data is stored in a computer in numeric form. Computer programs do what they do by executing a series of calculations on numeric data. It is the order and the pattern of those calculations that distinguishes one computer program from another.

Avoiding the detailed work

Fortunately, when we program using a high-level programming language such as Java, much of the detailed work is done for us behind the scenes.

Musicians or conductors

As programmers, we are more like conductors than musicians. The various parts of the computer represent the musicians. We tell them what to play, and when to play it, and if we do our job well, we produce a solution to a problem.

4.6.3.2 Compartments

As the computer program performs its calculations in the correct order, it is often necessary for it to store intermediate results someplace, and then come back and get them to use them in subsequent calculations later. The intermediate results are stored in memory, often referred to as RAM or *Random Access Memory* .

A mechanical analogy

We can think of random access memory as being analogous to a metal rack containing a large number of compartments. The compartments are all the same size and are arranged in a column. Each compartment has a numeric address printed above it. No two compartments have the same numeric address. Each compartment also has a little slot into which you can insert a name or a label for the compartment. No two compartments can have the same name.

Joe, the computer program

Think of yourself as a computer program. You have the ability to write values on little slips of paper and to put them into the compartments. You also have the ability to read the values written on the little slips of paper and to use those values for some purpose. However, there are two rules that you must observe:

- You may not remove a slip of paper from a compartment without replacing it by another slip of paper on which you have written a value.
- You may not put a slip of paper in a compartment without removing the one already there.

4.6.3.3 Checkout counter example

In understanding how you might behave as a human computer program, consider yourself to have a job working at the checkout counter of a small grocery store in the 1930s.

You have two tools to work with:

- A mechanical adding machine

- The rack of compartments described above

Initialization

Each morning, the owner of the grocery store tells you to insert a name in the slot above each compartment and to place a little slip of paper with a number written on it inside each compartment. *(In programming jargon, we would refer to this as initialization.)*

Each of the names on the compartments represents a type of grocery such as

- Beans
- Apples
- Pears

No two compartments can have the same name.

No compartment is allowed to have more than one slip of paper inside it.

The price of a can of beans

When you place a new slip of paper in a compartment, you must be careful to remove and destroy the one that was already there.

Each slip of paper that you insert into a compartment contains the price for the type of grocery identified by the label on the compartment.

For example, the slip of paper in the compartment labeled **Beans** may contain the value 15, meaning that each can of beans costs 15 cents.

The checkout procedure

As each customer comes to your checkout counter during the remainder of the day, you execute the following procedure:

- Examine each grocery item to determine its type.
- Read the price stored in the compartment corresponding to that type of grocery.
- Add that price to that customer's bill using your mechanical adding machine.

In programming jargon, we would say that as you process each grocery item for the same customer, you are *looping* . We would also say that you are executing a procedure or an *algorithm* .

When you have processed all of the grocery items for a particular customer, you would

- Press the TOTAL key on the adding machine,
- Turn the crank, and
- Present the customer with the bill.

A schematic representation of the procedure

We might represent the procedure in schematic form as shown in Image 1 (p. 364) .

Image 1: A checkout counter algorithm.

```
For each customer, do the following:

  For each item, do the following:
    a. Identify the type of grocery item
    b. Get the price from the compartment
    c. Add the price to accumulated total
  End loop on grocery items

  Present customer with a bill

End loop on a specific customer
```

4.6

Common programming activities

This procedure illustrates the three activities commonly believed to be the fundamental activities of any computer program:

- sequence
- selection
- loop

Sequence

A sequence of operations is illustrated by the three items labeled a, b, and c in Image 1 (p. 364) because they are executed in sequential order.

Selection

The process of identifying the type of grocery item is often referred to as *selection* . A selection operation is the process of selecting among two or more choices.

Loop

The process of repetitively examining each grocery item and processing it is commonly referred to as a *loop* . In the early days of programming, for a programming language named FORTRAN, this was referred to as a ***do loop*** .

An algorithm

The entire procedure is often referred to as an *algorithm* .

Modifying stored data

Sometimes during the day, the owner of the grocery store may come to you and say that he is going to increase the price of a can of Beans from 15 cents to 25 cents and asks you to take care of the change in price.

You write 25 on a slip of paper and put it in the compartment labeled Beans, being careful to remove and destroy the slip of paper that was previously in that compartment. For the rest of the day, the new price for Beans will be used in your calculations unless the owner asks you to change it again.

Not a bad analogy

This is a pretty good analogy to how random access memory is actually used by a computer program.

Names versus addresses

As a programmer using a high-level language such as Java, you usually don't have to be concerned about the numeric addresses of the compartments.

You are able to think about them and refer to them in terms of their names. *(Names are easier to remember than numeric addresses).* However, deep inside the computer, these names are cross-referenced to addresses and at the lowest level, the program works with memory addresses instead of names.

Execute an algorithm

A computer program always executes some sort of procedure, which is often called an *algorithm* . The algorithm may be very simple as described in the checkout counter example described above, or it may be very complex as would be the case for a spreadsheet program. As the program executes its algorithm, it uses the random access memory to store and retrieve the data that is needed to execute the algorithm.

Why is it called RAM?

The reason this kind of memory is called *RAM* or *random access memory* is that it can be accessed in any order.

Sequential memory

Some types of memory, such as a magnetic tape, must be accessed in sequential order. This means that to get a piece of data *(the price of beans, for example)* from deep inside the memory, it is necessary to start at the beginning and examine every piece of data until the correct one is found.

Combination random/sequential

Other types of memory, such as disks, provide a combination of sequential and random access. For example, the data on a disk is stored in tracks that form concentric circles on the disk. The tracks can be accessed in random order, but the data within a track must be accessed sequentially starting at a specific point on the track.

Sequential memory isn't very good for use by most computer programs because access to each particular piece of data is quite slow.

4.6.3.4 Sample program

Listing 1 (p. 365) shows a sample Java program that illustrates the use of memory for the storage and retrieval of data.

Listing 1: Program named Memory01.

```
//File Memory01.java
class Memory01 {
  public static void main(String[] args){
    int beans;
    beans = 25;
    System.out.println(beans);
  }//end main
}//End Memory01 class
```

4.7

Listing 2 (p. 366) shows a batch file that you can use to compile and run this program.

Listing 2: Batch file for Memory01.

```
echo off
cls

del *.class

javac -cp .; Memory01.java
java -cp .; Memory01

pause
```

4.8

Using the procedure that you learned in the *Getting Started* [31] module, you should be able to compile and execute this program. When you do, the program should display 25 on your computer screen.

Variables

You will learn in a future lesson that the term *variable* is synonymous with the term *compartment* that I have used for illustration purposes in this lesson.

The important lines of code

The use of memory is illustrated by the three lines of code in Listing 1 (p. 365) that begin with **int** , **beans** , and **System** . We will ignore the other lines in the program in this module and learn about them in future modules.

Declaring a variable

A memory compartment *(or variable)* is set aside and given the name **beans** by the line that begins with **int** in Listing 1 (p. 365) .

In programmer jargon, this is referred to as *declaring a variable* . The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given a name.

That name can be used later to refer to the value stored in that chunk of memory or variable.

This declaration in Listing 1 (p. 365) specifies that any value stored in the variable must be of type **int** . Basically, this means that the value must be an integer. Beyond that, don't worry about what the *type* means at this point. I will explain the concept of type in detail in a future module.

Storing a value in the variable

A value of 25 is stored in the variable named **beans** by the line in Listing 1 (p. 365) that begins with the word **beans** .

In programmer jargon, this is referred to as *assigning a value to a variable* .

From this point forward, when the code in the program refers to this variable by its name, **beans** , the reference to the variable will be interpreted to mean the value stored there.

Retrieving a value from the variable

The line in Listing 1 (p. 365) that begins with the word **System** reads the value stored in the variable named **beans** by referring to the variable by its name.

[31] http://cnx.org/content/m45137/latest/

This line also causes that value to be displayed on your computer screen. However, at this point, you needn't worry about what causes it to be displayed. You will learn those details in a future module. Just remember that the reference to the variable by its name, **beans** , reads the value stored in the variable.

The remaining details

Don't be concerned at this point about the other details in the program. They are there to make it possible for you to compile and execute the program. You will learn about them in future modules.

4.6.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.6.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0120: Java OOP: A Gentle Introduction to Java Programming
- File: Jb0120.htm
- Published: 11/16/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.7 Jb0120r Review[32]

4.7.1 Table of Contents

4.7.2 Preface

This module contains review questions and answers keyed to the module titled Jb0120: Java OOP: A Gentle Introduction to Java Programming [33] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.7.3 Questions

4.7.3.1 Question 1 .

True or false? All data is stored in a computer in numeric form. Computer programs do what they do by executing a series of calculations on numeric data. It is the order and the pattern of those calculations that distinguishes one computer program from another.

Answer 1 (p. 372)

4.7.3.2 Question 2

True or false? When we program using Java, we must perform most of the detailed work.

Answer 2 (p. 372)

4.7.3.3 Question 3

True or false? As the computer program performs its calculations in the correct order, it is often necessary for it to store intermediate results someplace, and then come back and get them to use them in subsequent calculations later.

Answer 3 (p. 371)

4.7.3.4 Question 4

True or false? The structured solution to a computer programming problem is often called an algorithm.

Answer 4 (p. 371)

4.7.3.5 Question 5

Which, if any of the following activities is not commonly believed to be fundamental activities of any computer program:

- A. sequence

[32] This content is available online at <http://cnx.org/content/m45164/1.4/>.
[33] http://cnx.org/content/m45138

- B. selection
- C. loop

Answer 5 (p. 371)

4.7.3.6 Question 6

True or false? As a programmer using a high-level language such as Java, you usually don't have to be concerned about the numeric memory addresses of variables.

Answer 6 (p. 371)

4.7.3.7 Question 7

Why is modern computer memory often referred to as RAM?

Answer 7 (p. 371)

4.7.3.8 Question 8

True or false? The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given an address.

Answer 8 (p. 371)

4.7.3.9 Question 9

True or false? A value of the type **int** must be an integer.

Answer 9 (p. 371)

4.7.3.10 Question 10

True or false? In programmer jargon, storing a value in a variable is also referred to as assigning a value to a variable.

Answer 10 (p. 371)

4.7.3.11 Question 11

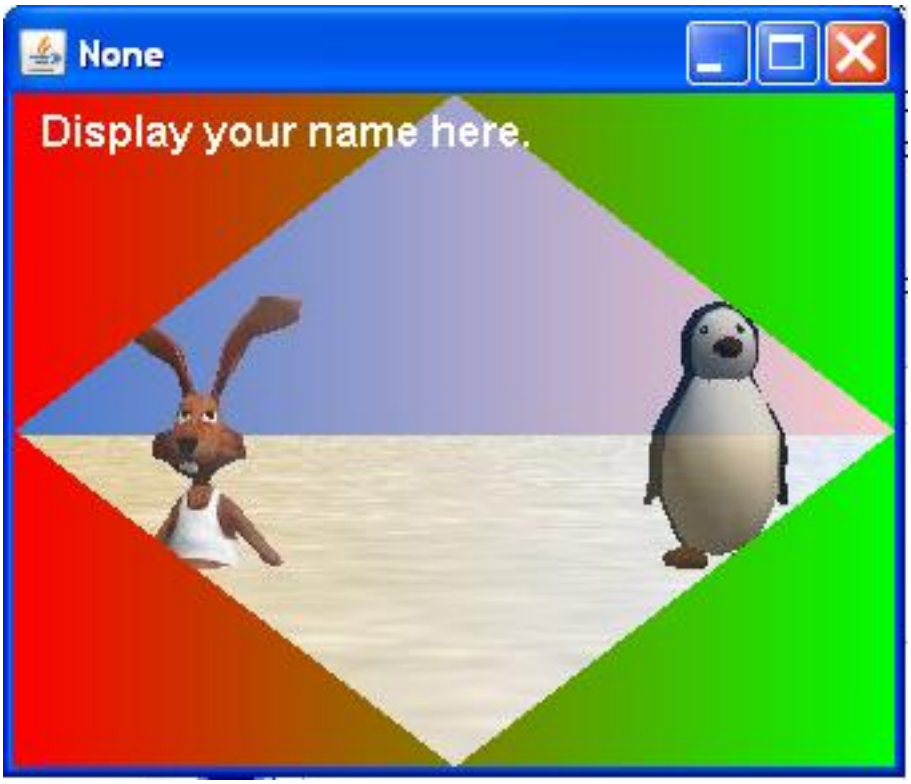
True or false? A reference to a variable name in Java code returns the value stored in the variable.

Answer 11 (p. 371)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.7.4 Answers

4.7.4.1 Answer 11

True.
Back to Question 11 (p. 369)

4.7.4.2 Answer 10

True.
Back to Question 10 (p. 369)

4.7.4.3 Answer 9

True.
Back to Question 9 (p. 369)

4.7.4.4 Answer 8

False. The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given a **name** .

Back to Question 8 (p. 369)

4.7.4.5 Answer 7

Modern computer memory is often called *RAM* or *random access memory* because it can be accessed in any order.
Back to Question 7 (p. 369)

4.7.4.6 Answer 6

True. You are able to think about variables and refer to them in terms of their names. *(Names are easier to remember than numeric addresses).* However, deep inside the computer, these names are cross-referenced to addresses and at the lowest level, the program works with memory addresses instead of names.
Back to Question 6 (p. 369)

4.7.4.7 Answer 5

None. All three are commonly believed to be the fundamental activities of any computer program.
Back to Question 5 (p. 368)

4.7.4.8 Answer 4

True.
Back to Question 4 (p. 368)

4.7.4.9 Answer 3

True.
Back to Question 3 (p. 368)

4.7.4.10 Answer 2

False. Fortunately, when we program using a high-level programming language such as Java, much of the detailed work is done for us behind the scenes.

Back to Question 2 (p. 368)

4.7.4.11 Answer 1

True.

Back to Question 1 (p. 368)

4.7.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0120r Review for A Gentle Introduction to Java Programming.
- File: Jb0120r.htm
- Published: 12/20/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.8 Jb0130: Java OOP: A Gentle Introduction to Methods in Java[34]

4.8.1 Table of Contents

[34] This content is available online at <http://cnx.org/content/m45139/1.2/>.

4.8.2 Preface

4.8.2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.

It provides a gentle introduction to Java programming methods.

4.8.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [35] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [36] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.8.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.8.2.3.1 Listings

[35] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[36] http://download.oracle.com/javase/7/docs/api/

4.8.3 Discussion and sample code

4.8.3.1 Introduction

Methods have been used in computer programming since the early days of programming. Methods are often called functions, procedures, subroutines, and various other names.

Calculate the square root

Suppose that your program needs to calculate the square root of a number. Referring back to your high-school algebra book, you could refresh your memory on how to calculate a square root. Then you could construct the algorithm describing that process.

Having the algorithm available, you could write the code to calculate the square root and insert it into your program code. Then you could compile, and run your program. If you did it all correctly, your program should calculate the square root. *(For reasons that will become apparent later, I will refer to the code that you inserted as in-line code.)*

Oops, need to do it all over again

Suppose that further on in your program you discover that you need to calculate the square root of another number. And later, you discover that you need to calculate the square root of still another number. Obviously, with a few changes, you could copy your original code and insert it as *in-line code* at each location in your program where you need to calculate the square root of a number.

Is there a better way?

However, after doing this a few times, you might start asking if there is a better way. The answer is *"yes, there is a better way."*

A method provides a better way

The better way is to create a separate program module that has the ability to calculate the square root and make that module available for use as a helper to your main program each time your main program needs to calculate a square root. In Java, this separate program module is called a **method** .

4.8.3.2 Standard methods

The Java programming language contains a large number of methods *(in the class libraries)* that are already available for your use. *(Later, I will illustrate the use of a standard method for calculating the square root of a number.)*

In addition to the standard methods that are already available, if you need a method to perform some function and there is no standard method already available to perform that function, you can write your own method.

4.8.3.3 Passing parameters

Make the method general

Normally, when designing and writing a method such as one that can calculate the square root of a number, it is desirable to write it in such a way that it can calculate the square root of any number *(as opposed to only one specific number)* . This is accomplished through the use of something called *parameters* .

The process of causing a method to be executed is commonly referred to as *calling the method* .

Pass me the number please

When your program calls the square-root method, it will need to tell the method the value for which the square root is needed.

In general, many methods will require that you provide certain kinds of information when you *call* them. The code in the method needs this information to be able to accomplish its purpose.

Passing parameters

This process of providing information to a method when you call it is commonly referred to as *passing parameters* to the method. For the square-root method, you need to pass a parameter whose value is the value of the number for which you need the square root.

4.8.3.4 Returning values

A method will usually

- perform an action
- send back an answer. or
- some combination of the two

Performing an action

An example of a method that performs an action is the method named **println** . We used the **println** method in an earlier module to cause information to be displayed on the computer screen. This method does not need to send back an answer, because that is not the objective of the method. The objective is simply to display some information.

Sending back an answer

On the other hand, a method that is designed to calculate the square root of a number needs to be able to send the square-root value back to the program that called the method. After all, it wouldn't be very useful if the method calculated the square root and then kept it a secret. The process of sending back an answer is commonly referred to as *returning a value* .

Returned values can be ignored

Methods can be designed in such a way that they either will or will not return a value. When a method does return a value, the program that called the method can either pay attention to that value and use it for some purpose, or ignore it entirely.

For example, in some cases where a method performs an action and also returns a value, the calling program may elect to ignore the returned value. On the other hand, if the sole purpose of a method is to return a value, it wouldn't make much sense for a program to call that method and then ignore the value that is returned *(although that would be technically possible)* .

4.8.3.5 Writing your own methods

As mentioned earlier, you can write your own methods in Java. I mention this here so you will know that it is possible. I will have more to say about writing your own methods in future modules.

4.8.3.6 Sample program

A complete listing of a sample program named **SqRt01.java** is provided in Listing 5 (p. 379) near the end of the lesson. A batch file that you can use to compile and run the program is provided in Listing 6 (p. 379) .

When you compile and run the program, the following output should appear on your computer screen:

5.049752469181039

6.0

As you will see shortly, these are the square root values respectively for 25.5 and 36.

4.8.3.6.1 Interesting code fragments

I will explain portions of this program in fragments. I will explain only those portions of the program that are germane to this module. Don't worry about the other details of the program. You will learn about those details in future modules.

You may find it useful to open this lesson in another browser window so that you can easily scroll back and forth among the fragments while reading the discussion.

The first code fragment that I will explain is shown in Listing 1 (p. 376) .

Listing 1: The price of beans.

```
double beans;
beans = 25.5;
```

4.9

What is the price of beans?

The code fragment shown in Listing 1 (p. 376) declares a *variable* named **beans** and assigns a value of 25.5 to the variable. *(I briefly discussed the declaration of variables in a previous module. I will discuss them in more detail in a future module.)*

What is that double thing?

In an earlier module, I declared a variable with a type named **int** . At that time, I explained that only integer values could be stored in that variable.

The variable named **beans** in Listing 1 (p. 376) is declared to be of the type **double** . I will explain the concept of data types in detail in a future module. Briefly, **double** means that you can store any numeric value in this variable, with or without a decimal part. In other words, you can store a value of 3 or a value of 3.33 in this variable, whereas a variable with a declared type of **int** won't accept a value of 3.33.

Every method has a name

Every method, every variable, and some other things as well have names. The names are *case sensitive* . By case sensitive, I mean that the method named **amethod** is not the same as the method named **aMethod** .

A few words about names in Java

There are several rules that define the format of allowable names in Java. You can dig into this in more detail on the web if you like, but if you follow these two rules, you will be okay:

- Use only letters and numbers in Java names.
- Always make the first character a letter.

A standard method named sqrt

Java provides a **Math** library that contains many standard methods. Included in those methods is a method named **sqrt** that will calculate and return the square root of a number that is passed as a parameter when the method is called.

The **sqrt** method is called on the right-hand side of the equal sign (=) in the code fragment in Listing 2 (p. 376) .

Listing 2: Compute the square root of the price of beans.

```
double sqRtBns = Math.sqrt(beans);
```

4.10

Calling the sqrt method

I'm not sure why you would want to do this, but the code fragment in Listing 2 (p. 376)

- calls the **sqrt** method and
- passes a copy of the value stored in the **beans** variable as a parameter.

The **sqrt** method calculates and returns the square root of the number that it receives as its incoming parameter. In this case, it returns the square root of the price of a can of beans.

A place to save the square root

I needed some place to save the square root value until I could display it on the computer screen later in the program. I declared another variable named **sqRtBns** in the code fragment in Listing 2 (p. 376) . I also caused the value returned from the **sqrt** method to be stored in, or assigned to, this new variable named **sqRtBns** .

How should we interpret this code fragment?

You can think of the process implemented by the code fragment in Listing 2 (p. 376) as follows.

First note that there is an equal sign (=) near the center of the line of code. *(Later we will learn that this is called the assignment operator.)*

The code on the left-hand side of the assignment operator causes a new chunk of memory to be set aside and named **sqRtBns** . *(We call this chunk of code a variable.)*

The code on the right-hand side of the assignment operator calls the **sqrt** method, passing a copy of the value stored in the **beans** variable to the method.

When the **sqrt** method returns the value that is the square root of its incoming parameter, the assignment operator causes that value to be stored and saved in the variable named **sqRtBns** .

Now display the square root value

The code in the fragment in Listing 3 (p. 377) causes the value now stored in **sqRtBns** to be displayed on the computer screen.

Listing 3: Display the square root value.

```
System.out.println(sqRtBns);
```

4.11

Another method is called here

The display of the square root value is accomplished by

- calling another standard method named **println** and
- passing a copy of the value stored in **sqRtBns** as a parameter to the method.

The **println** method performs an action *(displaying something on the computer screen)* and doesn't return a value.

A method exhibits behavior

We say that a method exhibits behavior. The behavior of the **sqrt** method is to calculate and return the square root of the value passed to it as a parameter.

The behavior of the **println** method is to cause its incoming parameter to be displayed on the computer screen.

What do we mean by syntax?

Syntax is a word that is often used in computer programming. The thesaurus in the editor that I am using to type this document says that a synonym for syntax is grammar.

I also like to think of syntax as meaning something very similar to format.

Syntax for passing parameters

Note the syntax in Listing 2 (p. 376) and Listing 3 (p. 377) for passing a parameter to the method. The syntax consists of following the name of the method with a pair of matching parentheses that contain the parameter. If more than one parameter is being passed, they are all included within the parentheses and separated by commas. Usually, the order of the parameters is important if more than one parameter is being passed.

Reusing the methods

The purpose of the code fragment in Listing 4 (p. 378) is to illustrate the reusable nature of methods.

Listing 4: Calling the same methods again.

```
double peas;
peas = 36.;
double sqRtPeas = Math.sqrt(peas);
System.out.println(sqRtPeas);
```

4.12

The code in this fragment calls the same **sqrt** method that was called before. In this case, the method is called to calculate the square root of the value stored in the variable named **peas** instead of the value stored in the variable named **beans** .

This fragment saves the value returned from the **sqrt** method in a new variable named **sqRtPeas** . Then the fragment calls the same **println** method as before to display the value now stored in the variable named **sqRtPeas** .

Write once and use over and over

Methods make it possible to write some code once and then use that code many times in the same program. This is the opposite of *in-line code* , which requires you to write essentially the same code multiple times in order to accomplish the same purpose more than once in a program.

4.8.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.8.5 Complete program listings

Listing 5 (p. 379) is a complete listing of the program named **SqRt01** .

Listing 5: The program named SqRt01.

```
//File SqRt01.java
class SqRt01 {
  public static void main(String[] args){
    double beans;
    beans = 25.5;
    double sqRtBns = Math.sqrt(beans);
    System.out.println(sqRtBns);
    double peas;
    peas = 36.;
    double sqRtPeas = Math.sqrt(peas);
    System.out.println(sqRtPeas);
  }//end main
}//End SqRt01 class
```

4.13

Listing 6 (p. 379) contains the commands for a batch file that can be used to compile and run the program named **SqRt01** .

Listing 6: A batch file for compiling and running the program named SqRt01.

```
echo off
cls

del *.class

javac -cp .; SqRt01.java
java -cp .; SqRt01

pause
```

4.14

4.8.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0130: Java OOP: A Gentle Introduction to Methods in Java
- File: Jb0130.htm
- Published: 12/16/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.9 Jb0130r Review[37]

4.9.1 Table of Contents

4.9.2 Preface

This module contains review questions and answers keyed to the module titled Jb0130: Java OOP: A Gentle Introduction to Methods in Java [38] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.9.3 Questions

4.9.3.1 Question 1 .

True or false? Methods are often called functions, procedures, subroutines, and various other names.

Answer 1 (p. 386)

4.9.3.2 Question 2

True or false? A Java method can be thought of as a separate program module that has the ability to do something useful. Having written the method, you can make it available for use as a helper to your main program each time your main program needs to have that useful thing done.

Answer 2 (p. 386)

4.9.3.3 Question 3

True or false? In Java, you must write all of the methods that you need.

Answer 3 (p. 385)

4.9.3.4 Question 4

True or false? In the following statement, **sqRtPeas** is the name of a method.

System.out.println(sqRtPeas);

Answer 4 (p. 385)

4.9.3.5 Question 5

True or false? Java only allows you to use the pre-written methods in the class libraries.

Answer 5 (p. 385)

[37] This content is available online at <http://cnx.org/content/m45165/1.4/>.
[38] http://cnx.org/content/m45139

4.9.3.6 Question 6

Normally, when designing and writing a method such as one that can calculate the square root of a number, it is desirable to write it in such a way that it can calculate the square root of any number *(as opposed to only one specific number)* . How is that accomplished?

Answer 6 (p. 385)

4.9.3.7 Question 7

True or false? According to common programming jargon, the process of causing a method to be executed is commonly referred to as *setting* the method.

Answer 7 (p. 385)

4.9.3.8 Question 8

True or false? This process of providing information to a method when you call it is commonly referred to as *sending a message* to the method.

Answer 8 (p. 385)

4.9.3.9 Question 9

True or false? When called, a method will usually

- perform an action
- send back an answer. or
- some combination of the two

Answer 9 (p. 385)

4.9.3.10 Question 10

True or false? A value of type **double** can be *(almost)* any numeric value, positive or negative, with or without a decimal part.

Answer 10 (p. 385)

4.9.3.11 Question 11

True or false? Java is not a case-sensitive programming language.

Answer 11 (p. 385)

4.9.3.12 Question 12

True or false? The following two rules will generally suffice to keep you out of trouble when defining variable and method names in Java:

- Use only letters and numbers in Java names.
- Always make the first character a letter.

Answer 12 (p. 384)

4.9.3.13 Question 13

True or false? In Java, the assignment operator is the % character.

Answer 13 (p. 384)

4.9.3.14 Question 14

True or false? The behavior of the **sqrt** method is to calculate and display the square root of the value passed to it as a parameter.

Answer 14 (p. 384)

4.9.3.15 Question 15

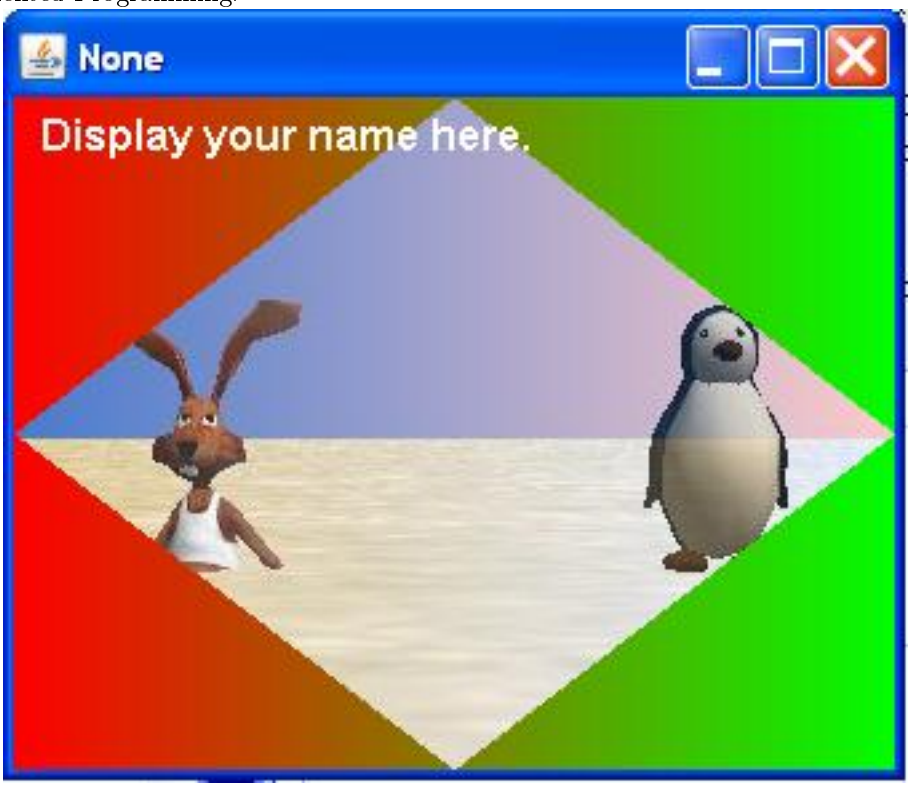
True or false? The syntax for passing parameters to a method consists of following the name of the method with a pair of matching parentheses that contain the parameter or parameters. If more than one parameter is being passed, they are all included within the parentheses and separated by commas. The order of the parameters is not important.

Answer 15 (p. 384)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.9.4 Answers

4.9.4.1 Answer 15

False. Normally the order in which parameters are passed to a method is very important.
Back to Question 15 (p. 383)

4.9.4.2 Answer 14

False. The behavior of the **sqrt** method is to calculate and ***return*** the square root of the value passed to it as a parameter.
Back to Question 14 (p. 383)

4.9.4.3 Answer 13

False. In Java, the assignment operator is the = character.
Back to Question 13 (p. 382)

4.9.4.4 Answer 12

True.
Back to Question 12 (p. 382)

4.9.4.5 Answer 11

False. Just like C, C++, and C#, Java is very much a case-sensitive programming language.
Back to Question 11 (p. 382)

4.9.4.6 Answer 10

True.
Back to Question 10 (p. 382)

4.9.4.7 Answer 9

True.
Back to Question 9 (p. 382)

4.9.4.8 Answer 8

False. If you continue in this field of study, you will learn that we *send messages* to objects by calling methods that belong to the objects. The process of providing information to a method when you call it is commonly referred to as *passing parameters* to the method.
Back to Question 8 (p. 382)

4.9.4.9 Answer 7

False. The process of causing a method to be executed is commonly referred to as ***calling*** or possibly ***invoking*** the method.
Back to Question 7 (p. 382)

4.9.4.10 Answer 6

That is accomplished through the use of something called *method parameters* .
Back to Question 6 (p. 382)

4.9.4.11 Answer 5

False. In addition to the standard methods that are already available, if you need a method to perform some function and there is no standard method already available to perform that function, you can write your own method.
Back to Question 5 (p. 381)

4.9.4.12 Answer 4

False. In the following statement, **println** is the name of a method. **sqRtPeas** is the name of a variable whose contents are being passed as a parameter to the **println** method.

```
System.out.println(sqRtPeas);
```

Back to Question 4 (p. 381)

4.9.4.13 Answer 3

False. The Java programming environment contains a large number of methods *(in the class libraries)* that are already available for you to use when you need them.
Back to Question 3 (p. 381)

4.9.4.14 Answer 2

True.
Back to Question 2 (p. 381)

4.9.4.15 Answer 1

True.
Back to Question 1 (p. 381)

4.9.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0130r Review: A Gentle Introduction to Methods in Java
- File: Jb0130r.htm
- Published: 12/20/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.10 Jb0140: Java OOP: Java comments[39]

4.10.1 Table of Contents

[39] This content is available online at <http://cnx.org/content/m45140/1.3/>.

4.10.2 Preface

4.10.2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.

It explains Java comments.

4.10.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [40] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [41] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.10.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

4.10.2.3.1 Images

4.10.2.3.2 Listings

[40] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[41] http://download.oracle.com/javase/7/docs/api/

4.10.3 Discussion and sample code

4.10.3.1 Comments

Producing and using a Java program consists of the following steps:

1. Write the source code.
2. Compile the source code.
3. Execute the program.

The source code consists of a set of instructions that will later be presented to a special program called a compiler for the purpose of producing a program that can be executed. In other words, when you write the source code, you are writing instructions that the compiler will use to produce the executable program.

Some things should be ignored

Sometimes, when you are writing source code, you would like to include information that may be useful to you, but should be ignored by the compiler. Information of that sort is called a **comment** .

Three styles of comments

Java supports the three styles of comments shown in Image 1 (p. 388) .

Image 1: Three styles of comments.

```
/** special documentation comment
used by the javadoc tool */

/* This is a
multi-line comment */

//Single-line comment
program code // Another single-line comment
```

4.15

The javadoc tool

The javadoc tool mentioned in Image 1 (p. 388) is a special program that is used to produce documentation for Java program. Comments of this style begin with /** and end with */ as shown in Image 1 (p. 388) .

The compiler ignores everything in the source code that begins and ends with this pattern of characters. Documentation produced using the javadoc program is very useful for on-line or on-screen documentation.

Multi-line comments

Multi-line comments begin with /* and end with */ as shown in Image 1 (p. 388) . As you have probably already guessed, the compiler also ignores everything in the source code that matches this format. *(A javadoc comment is simply a multi-line comment insofar as the compiler knows. Only the special program named javadoc.exe cares about javadoc comments.)*

The multi-line comment style is particularly useful for creating large blocks of information that should be ignored by the compiler. This style can be used to produce a comment consisting of a single line of text as well. However, the single-line comment style discussed in the next section requires less typing.

Single-line comments

Single-line comments begin with // and end at the end of the line. The compiler ignores the // and everything following the slash characters to the end of the line.

This style is particularly useful for inserting short comments throughout the source code. In this case, the // can appear at the beginning of the line as shown in Image 1 (p. 388) , or can appear anywhere in the line, including at the end of some valid source code *(also shown in Image 1* (p. 388) *)* .

4.10.3.2 Sample program

The purpose of the program named **Comments01** , which is shown in Listing 3 (p. 390) near the end of the module, is to illustrate the use of single and multi-line comments. The program does not contain any javadoc comments.

The commands for a batch file that you can use to compile and run this program are provided in Listing 4 (p. 391) .

When you compile and run the program, the following text should appear on your command-line screen:

Hello World

4.10.3.2.1 Interesting code fragments

I will explain this program in fragments, and will explain only those portions of the program that are germane to this module. Don't worry about the other details of the program at this time. You will learn about those details in future modules.

A multi-line comment

Listing 1 (p. 389) , shows a multi-line comment, which consists of three lines of text.

As required, this multi-line comment begins with /* and ends with */. The extra stars on the third line are simply part of the comment.

You will often see formats similar to this being used to provide a visual separation between multi-line comments and the other parts of a program.

Listing 1: A multi-line comment.

```
/*File Comments01.java
This is a multi-line comment.
*****************************************/
```

4.16

Single-line comments

Listing 2 (p. 390) shows three single-line comments. Can you spot them? Remember, single-line comments begin with //.

Listing 2: Three single-line comments.

```
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

4.17

One of the comments in Listing 2 (p. 390) starts at the beginning of the line. The other two comments follow some program code.

4.10.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.10.5 Complete program listings

Listing 3 (p. 390) contains a complete listing of the program named **Comments01** .

Listing 3: The program named Comments01.

```
/*File Comments01.java
This is a multi-line comment.
**************************************/
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

4.18

Listing 4 (p. 391) contains the commands for a batch file that can be used to compile and run the program named **Comments01** .

Listing 4: Batch file to compile and run the program named Comments01.

```
echo off
cls

del *.class

javac -cp .; Comments01.java
java -cp .; Comments01

pause
```

4.19

4.10.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: JJb0140: Java OOP: Java comments
- File: Jb0140.htm
- Published: 11/16/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.11 Jb0140r Review[42]

4.11.1 Table of Contents

4.11.2 Preface

This module contains review questions and answers keyed to the module titled Jb0140: Java OOP: Java comments [43] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.11.3 Questions

4.11.3.1 Question 1 .

True or false? Comments in your source code are ignored by the compiler.

Answer 1 (p. 395)

4.11.3.2 Question 2

True or false? Java supports the four styles of comments.

Answer 2 (p. 395)

4.11.3.3 Question 3

True or false? The **javadoc** tool is a special program that is used to compile Java programs.

Answer 3 (p. 395)

4.11.3.4 Question 4

True or false? Comments recognized by the **javadoc** tool begin with /** and end with */

Answer 4 (p. 395)

4.11.3.5 Question 5

True or false? Multi-line comments begin with /# and end with #/

Answer 5 (p. 394)

4.11.3.6 Question 6

True or false? The multi-line comment style is particularly useful for creating large blocks of information that should be ignored by the compiler.

Answer 6 (p. 394)

[42] This content is available online at <http://cnx.org/content/m45169/1.4/>.
[43] http://cnx.org/content/m45140

4.11.3.7 Question 7

True or false? The multi-line comment style cannot be used to produce a comment consisting of a single line of text.

Answer 7 (p. 394)

4.11.3.8 Question 8

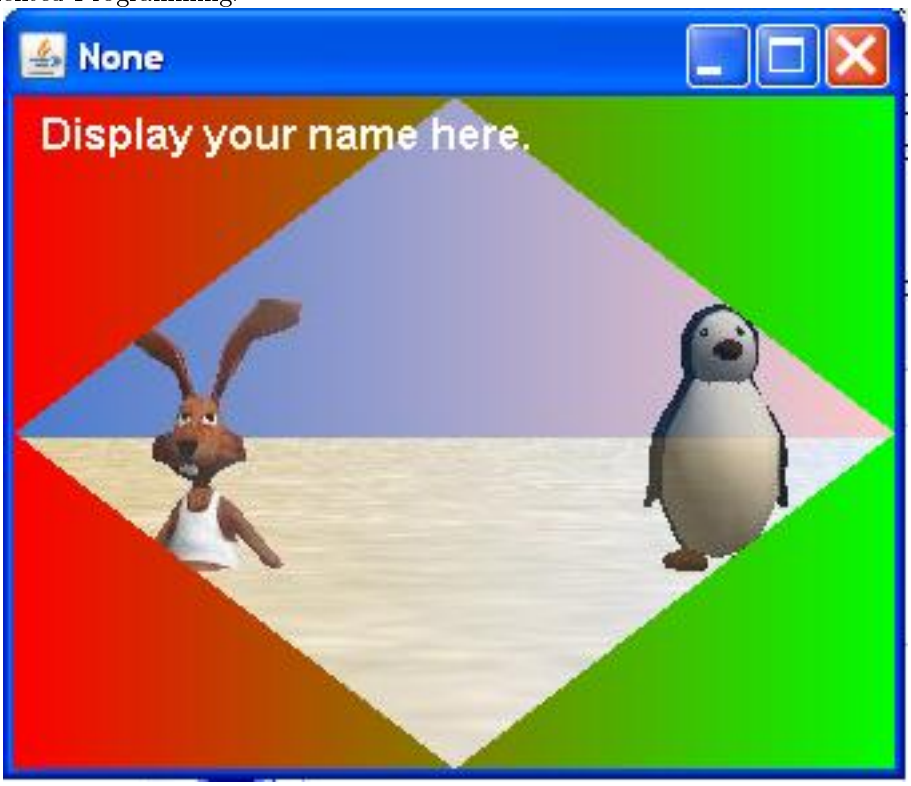
True or false? Single-line comments begin with // and end at the end of the line.

Answer 8 (p. 394)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.11.4 Answers

4.11.4.1 Answer 8

True.

Back to Question 8 (p. 393)

4.11.4.2 Answer 7

False. The multi-line comment style can be used to produce a comment consisting of none, one, or more lines of text.

Back to Question 7 (p. 393)

4.11.4.3 Answer 6

True.

Back to Question 6 (p. 392)

4.11.4.4 Answer 5

False. Multi-line comments begin with /* and end with */

Back to Question 5 (p. 392)

4.11.4.5 Answer 4

True.
Back to Question 4 (p. 392)

4.11.4.6 Answer 3

False. The **javadoc** tool is a special program that is used to produce documentation for Java program.
Back to Question 3 (p. 392)

4.11.4.7 Answer 2

False. Java supports the three styles of comments.
Back to Question 2 (p. 392)

4.11.4.8 Answer 1

True.
Back to Question 1 (p. 392)

4.11.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0140r Review: Java comments
- File: Jb0140r.htm
- Published: 11/21/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.12 Jb0150: Java OOP: A Gentle Introduction to Java Data Types[44]

4.12.1 Table of Contents

4.12.2 Preface

4.12.2.1 General

This module is part of a sub-collection of modules designed to help you learn to program computers.
It introduces Java data types.

4.12.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.*[45] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/*[46] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.12.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images while you are reading about them.

[44] This content is available online at <http://cnx.org/content/m45141/1.2/>.
[45] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[46] http://download.oracle.com/javase/7/docs/api/

4.12.2.3.1 Images

- Image 1 (p. 401) . Range of values for whole-number types.
- Image 2 (p. ??) . Definition of floating point.
- Image 3 (p. 402) . Different ways to represent 623.57185.
- Image 4 (p. 403) . Relationships between multiplicative factors and exponentiation.
- Image 5 (p. 404) . Other ways to represent the same information.
- Image 6 (p. 404) . Still other ways to represent 623.57185.
- Image 7 (p. 406) . Range of values for floating-point types.
- Image 8 (p. 407) . Example of the use of the boolean type.

4.12.3 Discussion

4.12.3.1 Introduction

Type-sensitive languages

Java and some other modern programming languages make heavy use of a concept that we refer to as *type* , or *data type* .

We refer to those languages as *type-sensitive languages* . Not all languages are type-sensitive languages. In particular, some languages hide the concept of type from the programmer and automatically deal with type issues behind the scenes.

So, what do we mean by type?

One analogy that comes to my mind is international currency. For example, many years ago, I spent a little time in Japan and quite a long time on an island named Okinawa *(Okinawa is now part of Japan)* .

Types of currency

At that time, as now, the type of currency used in the United States was the dollar. The type of currency used in Japan was the yen, and the type of currency used on the island of Okinawa was also the yen. However, even though two of the currencies had the same name, they were different types of currency, as determined by the value relationships among them.

The exchange rate

As I recall, at that time, the exchange rate between the Japanese yen and the U.S. dollar was 360 yen for each dollar. The exchange rate between the Okinawan yen and the U.S. dollar was 120 yen for each dollar. This suggests that the exchange rate between the Japanese yen and the Okinawan yen would have been 3 Japanese yen for each Okinawan yen.

Analogous to different types of data

So, why am I telling you this? I am telling you this to illustrate the concept that different types of currency are roughly analogous to different data types in programming.

Purchasing transactions were type sensitive

In particular, because there were three different types of currency involved, the differences in the types had to be taken into account in any purchasing transaction to determine the price in that particular currency. In other words, the purchasing process was sensitive to the type of currency being used for the purchase *(type sensitive)* .

Different types of data

Type-sensitive programming languages deal with different types of data. Some data types such at type **int** involve whole numbers only *(no fractional parts are allowed)* .

Other data types such as **double** involve numbers with fractional parts.

Some data types conceptually have nothing to do with numeric values, but deal only with the concept of true or false *(**boolean**)* or with the concept of the letters of the alphabet and the punctuation characters *(**char**)* .

Type specification

For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type *(we will learn some other names for instance later)* ?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

What do I mean by an instance of a type?

Think of the type specification as being analogous to the plan or blueprint for a model airplane. Assume that you build three model airplanes from the same set of plans. You will have created three instances of the plans.

We might say that an instance is the physical manifestation of a plan or a type.

Using mixed types

Somewhat secondary to the specifications for the different types, but also extremely important, is a set of rules that define what happens when you perform an operation involving mixed types *(such as making a purchase using some yen currency in combination with some dollar currency)* .

The short data type

For example, in addition to the integer type **int** , there is a data type in Java known as **short** . The **short** type is also an integer type.

If you have an instance of the **short** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -32,768 to +32,767.

This constitutes a set of 65,536 different values, including the value zero. No other value can be stored in an instance of the type **short** . For example, you cannot store the value 35,000 in an instance of the type **short** in Java. If you need to store that value, you will need to use some type other than **short** .

Kind of like an odometer

This is somewhat analogous to the odometer in your car *(the thing that records how many miles the car has been driven)* . For example, depending on the make and model of car, there is a specified set of values that can appear in the odometer. The value that appears in the odometer depends on how many miles your car has been driven.

It is fairly common for an odometer to be able to store and to display the set of all positive values ranging from zero to 99999. If your odometer is designed to store that set of values and if you drive your car more than 99999 miles, it is likely that the odometer will roll over and start back at zero after you pass the 99999-mile mark. In other words, that particular odometer does not have the ability to store a value of 100,000 miles. Once you pass the 99999-mark, the data stored in the odometer is corrupt.

Now let's return to the Java type named short

Assume that you have two instances of the type **short** in a Java program. What are the operations that you can perform on those instances? For example:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.
- You can divide one by the other.
- You can compare one with the other to determine which is algebraically larger.

There are some other operations that are allowed as well. In fact, there is a well-defined set of operations that you are allowed to perform on those instances. That set of operations is defined in the specification for the type **short** .

What if you want to do something different?

However, if you want to perform an operation that is not allowed by the type specification, then you will have to find another way to accomplish that purpose.

For example, some programming languages allow you to raise whole-number types to a power *(examples: four squared, six cubed, nine to the fourth power, etc.)* . However, that operation is not allowed by the Java specification for the type **short** . If you need to do that operation with a data value of the Java **short** type, you must find another way to do it.

Two major categories of type

Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

These categories are discussed in more detail in the following sections.

4.12.3.2 Primitive types

Java is an extensible programming language

What this means is that there is a core component to the language that is always available. Beyond this, individual programmers can extend the language to provide new capabilities. The primitive types discussed in this section are the types that are part of the core language. A later section will discuss user-defined types that become available when a programmer extends the language.

More subdivision

It seems that when teaching programming, I constantly find myself subdividing topics into sub-topics. I am going to subdivide the topic of Primitive Types into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Hopefully this categorization will make it possible for me to explain these types in a way that is easier for you to understand.

4.12.3.2.1 Whole-number types

The whole-number types, often called *integer* types, are relatively easy to understand. These are types that can be used to represent data without fractional parts.

Applesauce and hamburger

For example, consider purchasing applesauce and hamburger. At the grocery store where I shop, I am allowed to purchase cans of applesauce only in whole-number or integer quantities.

Can purchase integer quantities only

For example, the grocer is happy to sell me one can of applesauce and is even happier to sell me 36 cans of applesauce. However, she would be very unhappy if I were to open a can of applesauce in the store and attempt to purchase 6.3 cans of applesauce.

Counting doesn't require fractional parts

A count of the number of cans of applesauce that I purchase is somewhat analogous to the concept of whole-number data types in Java. Applesauce is not available in fractional parts of cans *(at my grocery store)* .

Fractional pounds of hamburger are available

On the other hand, the grocer is perfectly willing to sell me 6.3 pounds of hamburger. This is somewhat analogous to *floating-point data types* in Java.

Accommodating applesauce and hamburger in a program

Therefore, if I were writing a program dealing with quantities of applesauce and hamburger, I might elect to use a whole number type to represent cans of applesauce and to use a floating-point type to represent pounds of hamburger.

Different whole-number types

In Java, there are four different whole-number types:

- byte

- short
- int
- long

(The char type is also a whole number type, but since it is not intended to be used for arithmetic, I discuss it later as a character type.)

The four types differ primarily in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Differences in operations?

Although there are some subtle differences among the four whole-number types in terms of the operations that you can perform on them, I will defer a discussion of those differences until a more advanced module. *(For example some operations require instances of the* ***byte*** *and* ***short*** *types to be converted to type* ***int*** *before the operation takes place.)*

Algebraically signed values

All four of these types can be used to represent algebraically signed values ranging from a specific negative value to a specific positive value.

Range of the byte type

For example, the **byte** type can be used to represent the set of whole numbers ranging from -128 to +127 inclusive. *(This constitutes a set of 256 different values, including the value zero.)*

The **byte** type cannot be used to represent any value outside this range. For example, the **byte** type cannot be used to represent either -129 or +128.

No fractional parts allowed by the byte type

Also, the **byte** type cannot be used to represent fractional values within the allowable range. For example, the byte type cannot be used to represent the value of 63.5 or any other value that has a fractional part.

Like a strange odometer

To form a crude analogy, the byte type is sort of like a strange odometer in a new *(and unusual)* car that shows a mileage value of -128 when you first purchase the car. As you drive the car, the negative values shown on the odometer increment toward zero and then pass zero. Beyond that point they increment up toward the value of +127.

Oops, numeric overflow!

When the value passes *(or attempts to pass)* +127 miles, something bad happens. From that point forward, the value shown on the odometer is not a reliable indicator of the number of miles that the car has been driven.

Ranges for each of the whole-number types

Image 1 (p. 401) shows the range of values that can be accommodated by each of the four whole-number types supported by the Java programming language:

Image 1: Range of values for whole-number types.

```
byte
-128 to +127

short
-32768 to +32767

int
-2147483648 to +2147483647

long
-9223372036854775808 to +9223372036854775807
```

4.20

Can represent some fairly large values

As you can see, the **int** and **long** types can represent some fairly large values. However, if your task involves calculations such as distances in interstellar space, these ranges probably won't accommodate your needs. This will lead you to consider using the *floating-point* types discussed in the upcoming sections. I will discuss the operations that can be performed on whole-number types more fully in future modules.

4.12.3.2.2 Floating-point types

Floating-point types are a little more complicated than whole-number types. I found the definition of floating-point shown in Image 2 (p. **??**) in the *Free On-Line Dictionary of Computing* at this URL [47] .

A number representation consisting of a mantissa, M, an exponent, E, and an (assumed) radix *(or "base")* . The number

4.21

[47] http://foldoc.org/floating+point

So what does this really mean?

Assuming a base or radix of 10, I will attempt to explain it using an example.

Consider the following value:

623.57185

I can represent this value in any of the ways shown in Image 3 (p. 402) *(where * indicates multiplication).*

Image 3: Different ways to represent 623.57185.

```
.62357185*1000
6.2357185*100
62.357185*10
623.57185*1
6235.7185*0.1
62357.185*0.01
623571.85*0.001
6235718.5*0.0001
62357185.*0.00001
```

4.22

In other words, I can represent the value as a mantissa (62357185) multiplied by a factor where the purpose of the factor is to represent a left or right shift in the position of the decimal point.

Now consider the factor

Each of the factors shown in Image 3 (p. 402) represents the value of ten raised to some specific power, such as ten squared, ten cubed, ten raised to the fourth power, etc.

Exponentiation

If we allow the following symbol (^) to represent exponentiation *(raising to a power)* and allow the following symbol (/) to represent division, then we can write the values for the above factors in the ways shown in Image 4 (p. 403) .

Note in particular the characters following the first equal character (=) on each line, which I will refer to later as the exponents.

Image 4: Relationships between multiplicative factors and exponentiation.

```
    1000 = 10^+3 = 1*10*10*10
100 = 10^+2 = 1*10*10
10 = 10^+1 = 1*10
1 = 10^+0 = 1
0.1 = 10^-1 = 1/10
0.01 = 10^-2 = 1/(10*10)
0.001 = 10^-3 = 1/(10*10*10)
0.0001 = 10^-4 = 1/(10*10*10*10)
0.00001 = 10^-5 = 1/(10*10*10*10*10)
```

4.23

In the above notation, the term 10^+3 means 10 raised to the third power.

The zeroth power

By definition, the value of any value raised to the zeroth power is 1. *(Check this out in your high-school algebra book.)*

The exponent and the factor

Hopefully, at this point you will understand the relationship between the exponent and the factor introduced earlier in Image 3 (p. 402) .

Different ways to represent the same value

Having reached this point, by using substitution, I can rewrite the original set of representations (p. 402) of the value 623.57185 in the ways shown in Image 5 (p. 404) .

(It is very important to for you to understand that these are simply different ways to represent the same value.)

Image 5: Other ways to represent the same information.

```
    .62357185*10^+3
6.2357185*10^+2
62.357185*10^+1
623.57185*10^+0
6235.7185*10^-1
62357.185*10^-2
623571.85*10^-3
6235718.5*10^-4
62357185.*10^-5
```

4.24

A simple change in notation

Finally, by making a simplifying change in notation where I replace (*10^) by (E) I can rewrite the different representations of the value of 623.57185 in the ways shown in Image 6 (p. 404) .

Image 6: Still other ways to represent 623.57185.

```
    .62357185E+3
6.2357185E+2
62.357185E+1
623.57185E+0
6235.7185E-1
62357.185E-2
623571.85E-3
6235718.5E-4
62357185.E-5
```

4.25

Getting the true value

Floating point types represent values as a mantissa containing a decimal point along with an exponent

value which tells how many places to shift the decimal point to the left or to the right in order to determine the true value.

Positive exponent values mean that the decimal point should be shifted to the right. Negative exponent values mean that the decimal point should be shifted to the left.

Maintaining fractional parts

One advantage of floating-point types is that they can be used to maintain fractional parts in data values, such as 6.3 pounds of hamburger.

Accommodating a very large range of values

Another advantage is that a very large range of values can be represented using a reasonably small amount of computer memory for storage of the values.

Another example

For example *(assuming that I counted the number of digits correctly)* the following very large value
62357185000000000000000000000000000000.0
can be represented as
6.2357185E+37
Similarly, again assuming that I counted the digits correctly, the following very small value
0.00000000000000000000000000000062357185
can be represented as
6.2357185E-30

When would you use floating-point?

If you happen to be working in an area where you

- need to keep track of fractional parts *(such as the amount of hamburger in a package)* ,
- have to work with extremely large numbers *(distances between galaxies)* , or
- have to work with extremely small values *(the size of atomic particles)* ,

then you will need to use the floating-point types.

Don't use floating-point in financial transactions

You probably don't want to use floating-point in financial calculations, however, because there is a lot of rounding that takes place in floating-point calculations. In other words, floating point calculations provide answers that are very close to the truth but the answers are often not exact.

Two floating-point types

Java supports two different floating point types:

- float
- double

These two types differ primarily in terms of the range of values that they can support.

Range of values for floating point types

The table in **Image 7** (p. 406) shows the smallest and largest values that can be accommodated by each of the floating-point types. Values of either type can be either positive or negative.

Image 7: Range of values for floating-point types.

```
    float
1.4E-45 to 3.4028235E38

double
4.9E-324 to 1.7976931348623157E308
```

4.26

I will discuss the operations that can be performed on floating-point types in a future module.

4.12.3.2.3 The character type

Computers deal only in numeric values. They don't know how to deal directly with the letters of the alphabet and punctuation characters. This gives rise to a type named **char** .

Purpose of the char type

The purpose of the character type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character, much as you may have done to create secret codes when you were a child.

A single character type

Java supports a single character type named **char** . The char type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Why so many characters?

The reason for the large number of possible characters is to make it possible to represent the characters making up the alphabets of many different countries and many different spoken languages.

What are the numeric values representing characters?

As long as the characters that you use in your program appear on your keyboard, you usually don't have a need to know the numeric value associated with the different characters. If you are curious, however, the upper-case A is represented by the value 65 in the Unicode character set.

Representing a character symbolically

In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:

'A'.

The Java programming tools know how to cross reference that specific character symbol against the Unicode table to obtain the corresponding numeric value. *(A discussion of the use of the* ***char*** *type to represent characters that don't appear on your keyboard is beyond the scope of this module.)*

I will discuss the operations that can be performed on the **char** type in a future module.

4.12.3.2.4 The boolean type

The boolean type is the simplest type supported by Java. It can have only two values:

- true
- false

Generally speaking, about the only operations that can be directly applied to an instance of the **boolean** type are to change it from **true** to **false** , and vice versa. However, the **boolean** type can be included in a large number of somewhat higher-level operations.

The **boolean** type is commonly used in some sort of a test to determine what to do next, such as that shown in Image 8 (p. 407) .

Image 8: Example of the use of the boolean type.

```
Perform a test that returns a value of type boolean.
if that value is true,
  do one thing
otherwise (meaning that value is false)
  do a different thing
```

4.27

I will discuss the operations that can be performed on the boolean type in more detail in a future module.

4.12.3.3 User-defined or reference types

Extending the language

Java is an *extensible* programming language. By this, I mean that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Creating new types

One of the ways that individual programmers can extend the language is to create new types. When creating a new type, the programmer must define the set of values that can be stored in an instance of the type as well as the operations that can be performed on instances of the type.

No magic involved

While this might initially seem like magic, once you get to the heart of the matter, it is really pretty straightforward. New types are created by combining instances of primitive types along with instances of other user-defined types. In other words, the process begins with the primitive types explained earlier and builds upward from there.

An example

For example, a **String** type, which can be used to represent a person's last name, is just a grouping of a bunch of instances of the primitive **char** or character type.

A user-defined **Person** type, which could be used to represent both a person's first name and their last name, might simply be a grouping of two instances of the user-defined **String** type.

Differences

The biggest conceptual difference between the **String** type and the **Person** type is that the **String** type is contained in the standard Java library while the **Person** type isn't in that library. However, you could put it in a library of your own design if you choose to do so.

Removing types

You could easily remove the **String** type from your copy of the standard Java library if you choose to do so, although that would probably be a bad idea. However, you cannot remove the primitive **double** type from the core language without making major modifications to the language.

The company telephone book

A programmer responsible for producing the company telephone book might create a new type that can be used to store the first and last names along with the telephone number of an individual. That programmer might choose to give the new type the name **Employee** .

The programmer could create an instance of the **Employee** type to represent each employee in the company, populating each such instance with the name and telephone number for an individual employee. *(At this point, let me sneak a little jargon in and tell you that we will be referring to such instances as objects.)*

A comparison operation

The programmer might define one of the allowable operations for the new **Employee** type to be a comparison between two objects of the new type to determine which is greater in an alphabetical sorting sense. This operation could be used to sort the set of objects representing all of the employees into alphabetical order. The set of sorted objects could then be used to print a new telephone book.

A name-change operation

Another allowable operation that the programmer might define would be the ability to change the name stored in an object representing a particular employee. For example when Suzie Smith marries Tom Jones, she might elect to thereafter be known as

- Suzie Smith
- Suzie Jones,
- Suzie Smith-Jones,
- Suzie Jones-Smith, or
- something entirely different.

In this case, there would be a need to modify the object that represents Suzie in order to reflect her newly-elected surname. *(Or perhaps Tom Jones might elect to thereafter be known as Tom Jones-Smith, in which case it would be necessary to modify the object that represents him.)*

An updated telephone book

The person charged with maintaining the database could

- use the name-changing operation to modify the object and change the name,
- make use of the sorting operation to re-sort the set of objects, and
- print and distribute an updated version of the telephone book.

Many user-defined types already exist

Unlike the primitive types which are predefined in the core language, I am unable to give you much in the way of specific information about user-defined types, simply because they don't exist until a user defines them.

I can tell you, however, that when you obtain the Java programming tools from Sun, you not only receive the core language containing the primitive types, you also receive a large library containing several thousand user-defined types that have already been defined. A large documentation package is available from Sun to help you determine the individual characteristics of these user-defined types.

The most important thing

At this stage in your development as a Java programmer, the most important thing for you to know about user-defined types is that they are possible.

You can define new types. Unlike earlier procedural programming languages such as C and Pascal, you are no longer forced to adapt your problem to the available tools. Rather, you now have the opportunity to extend the tools to make them better suited to solve your problem.

The class definition

The specific Java mechanism that makes it possible for you to define new types is a mechanism known as the *class definition* .

In Java, whenever you define a new class, you are at the same time defining a new type. Your new type can be as simple, or as complex and powerful as you want it to be.

An object *(instance)* of your new type can contain a very small amount of data, or it can contain a very large amount of data. The operations that you allow to be performed on an object of your new type can be rudimentary, or they can be very powerful.

It is all up to you

Whenever you define a new class *(type)* you not only have the opportunity to define the data definition and the operations, you also have a responsibility to do so.

Much to learn and much to do

But, you still have much to learn and much to do before you will need to define new types.

There are a lot of fundamental programming concepts that we will need to cover before we seriously embark on a study involving the definition of new types.

For the present then, simply remember that such a capability is available, and if you work to expand your knowledge of Java programming one small step at a time, when we reach the point of defining new types, you will be ready and eager to do so.

4.12.3.4 Sample program

I'm not going to provide a sample program in this module. Instead, I will be using what you have learned about Java data types in the sample programs in future modules.

4.12.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0150: Java OOP: A Gentle Introduction to Java Data Types
- File: Jb0150.htm
- Published: 11/17/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.13 Jb0150r Review[48]

4.13.1 Table of Contents

4.13.2 Preface

This module contains review questions and answers keyed to the module titled Jb0150: Java OOP: A Gentle Introduction to Java Data Types [49] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.13.3 Questions

4.13.3.1 Question 1 .

True or false? Java is a type-sensitive language.

Answer 1 (p. 417)

4.13.3.2 Question 2

True or false? Data type **double** involves whole numbers only *(no fractional parts are allowed)* .

Answer 2 (p. 417)

4.13.3.3 Question 3

True or false? Type **double** involves numbers with fractional parts.

Answer 3 (p. 417)

4.13.3.4 Question 4

True or false? All Java data types conceptually have something to do with numeric values.

Answer 4 (p. 417)

4.13.3.5 Question 5

True or false? The Java **char** type deals conceptually with the letters of the alphabet, the numeric characters, and the punctuation characters.

Answer 5 (p. 417)

[48] This content is available online at <http://cnx.org/content/m45168/1.4/>.
[49] http://cnx.org/content/m45141

4.13.3.6 Question 6

True or false? For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

Answer 6 (p. 417)

4.13.3.7 Question 7

True or false? If you have an instance of the **byte** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -256 to +255.

Answer 7 (p. 417)

4.13.3.8 Question 8

Name or describe four of the operations that you can perform with data of type **short** .

Answer 8 (p. 416)

4.13.3.9 Question 9

True or false? Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

Answer 9 (p. 416)

4.13.3.10 Question 10

True or false? The primitive types are not part of the core language.

Answer 10 (p. 416)

4.13.3.11 Question 11

True or false? For purposes of discussion, primitive types can be subdivided into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Answer 11 (p. 416)

4.13.3.12 Question 12

True or false? In Java, there are three different whole-number types:

- byte
- short
- int

Answer 12 (p. 416)

4.13.3.13 Question 13

True or false? The whole-number types differ in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Answer 13 (p. 416)

4.13.3.14 Question 14

True or false? Java provides an unsigned version of all of the primitive whole-number types.

Answer 14 (p. 416)

4.13.3.15 Question 15

True or false? Floating point types represent values as a mantissa containing a decimal point along with an exponent value that tells how many places to shift the decimal point to the left or to the right in order to determine the true value.

Answer 15 (p. 416)

4.13.3.16 Question 16

True or false? With a floating point type, positive exponent values mean that the decimal point should be shifted to the left. Negative exponent values mean that the decimal point should be shifted to the right.

Answer 16 (p. 416)

4.13.3.17 Question 17

True or false? Java supports two different floating point types:

- float
- double

Answer 17 (p. 415)

4.13.3.18 Question 18

True or false? The purpose of the **char** type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character.

Answer 18 (p. 415)

4.13.3.19 Question 19

True or false? The **char** type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Answer 19 (p. 415)

4.13.3.20 Question 20

True or false? In Java, you usually represent a character in your program by surrounding it with quotation marks as shown below:

"A".

Answer 20 (p. 415)

4.13.3.21 Question 21

True or false? The boolean type can have three values:

- true
- false
- maybe

Answer 21 (p. 415)

4.13.3.22 Question 22

True or false? Java is an *extensible* programming language, meaning that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Answer 22 (p. 415)

4.13.3.23 Question 23

True or false? As is the case in C++, one of the ways that individual programmers can extend the Java language is to create overloaded operators for the primitive types.

Answer 23 (p. 415)

4.13.3.24 Question 24

True or false? One of the ways that individual programmers can extend the Java language is to create new types.

Answer 24 (p. 415)

4.13.3.25 Question 25

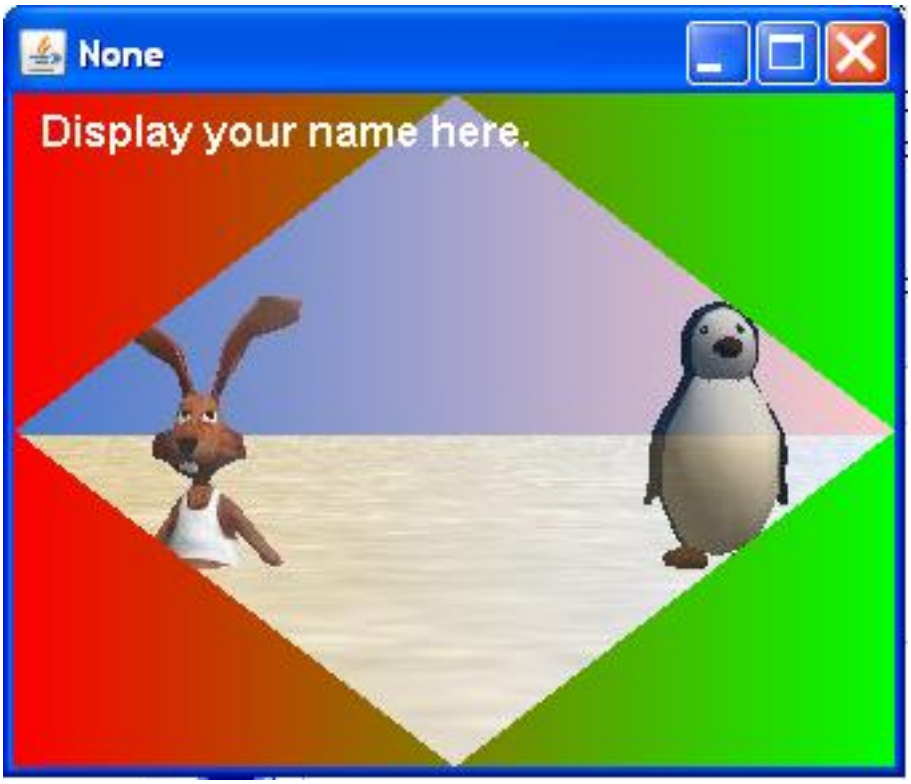
True or false? The specific Java mechanism that makes it possible for programmers to define new types is a mechanism known as the *class definition* .

Answer 25 (p. 415)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.13.4 Answers

4.13.4.1 Answer 25

True.
Back to Question 25 (p. 413)

4.13.4.2 Answer 24

True.
Back to Question 24 (p. 413)

4.13.4.3 Answer 23

False. Java does not allow programmers to create overloaded operators for the primitive types.
Back to Question 23 (p. 413)

4.13.4.4 Answer 22

True.
Back to Question 22 (p. 413)

4.13.4.5 Answer 21

False. The boolean type can have only two values:

- true
- false

Back to Question 21 (p. 413)

4.13.4.6 Answer 20

False. In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:
'A'.
Back to Question 20 (p. 412)

4.13.4.7 Answer 19

True.
Back to Question 19 (p. 412)

4.13.4.8 Answer 18

True.
Back to Question 18 (p. 412)

4.13.4.9 Answer 17

True.
Back to Question 17 (p. 412)

4.13.4.10 Answer 16

False. With a floating point type, positive exponent values mean that the decimal point should be shifted to the ***right*** . Negative exponent values mean that the decimal point should be shifted to the ***left*** .
Back to Question 16 (p. 412)

4.13.4.11 Answer 15

True.
Back to Question 15 (p. 412)

4.13.4.12 Answer 14

False. Other than type **char** , there are no unsigned whole-number primitive types in Java.
Back to Question 14 (p. 412)

4.13.4.13 Answer 13

True.
Back to Question 13 (p. 412)

4.13.4.14 Answer 12

False. In Java, there are five different whole-number types:

- byte
- short
- int
- long
- char

Back to Question 12 (p. 411)

4.13.4.15 Answer 11

True.
Back to Question 11 (p. 411)

4.13.4.16 Answer 10

False. The primitive types are part of the core language.
Back to Question 10 (p. 411)

4.13.4.17 Answer 9

True.
Back to Question 9 (p. 411)

4.13.4.18 Answer 8

Four of the possible operations are:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.

- You can divide one by the other.

Back to Question 8 (p. 411)

4.13.4.19 Answer 7

False. If you have an instance of the **byte** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -128 to +127.

Back to Question 7 (p. 411)

4.13.4.20 Answer 6

True.

Back to Question 6 (p. 411)

4.13.4.21 Answer 5

True.

Back to Question 5 (p. 410)

4.13.4.22 Answer 4

False. In Java, data type ***boolean*** conceptually has nothing to do with numeric values, but deals only with the concept of ***true*** or ***false*** .

Back to Question 4 (p. 410)

4.13.4.23 Answer 3

True.

Back to Question 3 (p. 410)

4.13.4.24 Answer 2

False. Some data types such at type **int** involve whole numbers only *(no fractional parts are allowed)* .

Back to Question 2 (p. 410)

4.13.4.25 Answer 1

True.

Back to Question 1 (p. 410)

4.13.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0150r Review: A Gentle Introduction to Java Data Types
- File: Jb0150r.htm
- Published: 11/21/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.14 Jb0160: Java OOP: Hello World[50]

4.14.1 Table of Contents

4.14.2 Preface

It is traditional in introductory programming courses to write and explain a simple program that prints the text ***"Hello World"*** on the computer screen.

This module continues that tradition.

4.14.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view images and listings while you are reading about them.

4.14.2.1.1 Images

[50] This content is available online at <http://cnx.org/content/m45143/1.2/>.

4.14.2.1.2 Listings

4.14.3 Introduction

This module introduces you to Java programming by presenting and discussing a traditional *Hello World* program.

Two approaches

Java programs can be written and executed in several different ways, including the following:

- Stand-alone application from the command line.
- Applet that runs under control of a Java-capable browser.

It is also possible in many cases to write applets, which can be run in a stand-alone mode from the command line, or can be run under control of a Java-capable browser. An example of such an applet will be presented in a future module.

Applets vs. applications

Programming an *"application"* in Java is significantly different from programming an *"applet."* Applets are designed to be downloaded and executed on-line under control of a browser.

Restrictions on applets

Their functionality of an applet is usually restricted in an attempt to prevent downloaded applets from damaging your computer or your data. No such restrictions apply to the functionality of a Java application.

Class definitions

All Java programs consist of one or more **class** definitions. In this course, I will often refer to the *primary* class definition for a Java *application* as the *controlling class* .

The main method

A stand-alone Java application requires a method named **main** in its *controlling class* .

An **Applet** does not require a **main** method. The reason that a Java **Applet** does not require a **main** method will be explained in a future module.

Getting started

Image 1 (p. **??**) shows the steps for compiling and running a Java application.

1. Download and install the JDK from Oracle. Also consider downloading and installing the documentation, which is a separate download.

2. Using any editor that can produce a plain text file *(such as Notepad),* create a source code file with the extension on the file name being .java This file contains your actual Java instructions. *(You can copy some sample programs from the early lessons in this collection to get started.)*

3. Open a command-line window and change directory to the directory containing the source file. It doesn't really matter which directory the source file is in, but I normally put my Java files in a directory all their own.

4. Assume that the name of the file is ***joe.java*** , just to have something definitive to refer to.

5. To compile the file, enter the following command at the prompt:

javac joe.java

6. Correct any compiler errors that show up. Once you have corrected all compiler errors, the **javac** program will execute and return immediately to the prompt with no output. At that point, the directory should also contain a file named ***joe.class*** and possibly some other files with a .class extension as well. These are the compiled Java files.

7. To run the program, enter the following command:

java joe

8. If your program produces the correct output, congratulations. You have written, compiled, and executed a Java application. If not, you will need to determine why not.

1. Download and install the JDK from Oracle. Also consider downloading and installing the documentation, which is a separate download.
2. Using any editor that can produce a plain text file *(such as Notepad),* create a source code file with the extension on the file name being .java This file contains your actual Java instructions. *(You can copy some sample programs from the early lessons in this collection to get started.)*
3. Open a command-line window and change directory to the directory containing the source file. It doesn't really matter which directory the source file is in, but I normally put my Java files in a directory all their own.
4. Assume that the name of the file is ***joe.java*** , just to have something definitive to refer to.
5. To compile the file, enter the following command at the prompt:

javac joe.java

6. Correct any compiler errors that show up. Once you have corrected all compiler errors, the **javac** program will execute and return immediately to the prompt with no output. At that point, the directory should also contain a file named ***joe.class*** and possibly some other files with a .class extension as well. These are the compiled Java files.
7. To run the program, enter the following command:

java joe

8. If your program produces the correct output, congratulations. You have written, compiled, and executed a Java application. If not, you will need to determine why not.

Here are the steps for compiling and running a Java application, based on the assumption that you are running under Wi

4.28

4.14.4 The Java version of Hello World

The class file

Compiled Java programs are stored in *"bytecode"* form in a file with an extension of **class** where the name of the file is the same as the name of the *controlling class (or other class)* in the program.

The main method is static

The **main** method in the controlling class of an application must be *static* , which results in **main** being a *class* method.

Class methods can be called without a requirement to instantiate an object of the class.

When a Java application is started, the *Java Virtual Machine or JVM (an executable file named java.exe)* finds and calls the **main** method in the class whose name matches the name of the class file specified on the command line.

Running an application

For example, to start the JVM and run a Java application named **hello1** , a command such as the following must be executed at the operating system prompt:

java hello1

This command instructs the operating system to start the JVM, and then instructs the JVM to find and execute the java application stored in the file named **hello1.class** . *(Note that the .class extension is not included in the command (p. 420) .)*

This sample program is a Java application named **hello1.java** .

When compiled, it produces a class file named **hello1.class** .

When the program is run, the JVM calls the **main** method defined in the *controlling class* .

The **main** method is a *class* method.

Class methods can be called without a requirement to instantiate an object of the class.

The program displays the following words on the screen:

Hello World

4.14.5 Interesting code fragments

I will explain this program code in fragments. A complete listing of the program is provided in Listing 5 (p. 424) .

The code fragment in Listing 1 (p. 421) shows the first line of the class definition for the controlling class named **hello1** . *(I will discuss class definitions in detail in a future module.)*

Listing 1: Beginning of the class named hello1.

```
class hello1 { //define the controlling class
```

4.29

The code fragment in Listing 2 (p. 421) begins the definition of the **main** method. I will also discuss method definitions in detail in a future module.

Listing 2: Beginning of the main method.

```
public static void main(String[] args){
```

4.30

The fragment in Listing 3 (p. 422) causes the string **Hello World** to be displayed on the command-line screen.

The statement in Listing 3 (p. 422) is an extremely powerful statement from an object-oriented programming viewpoint. When you understand how it works, you will be well on your way to understanding the Java version of Object-Oriented Programming (OOP).

I will discuss this statement in more detail later in a future module.

Listing 3: Display the string Hello World.

```
System.out.println("Hello World");
```

4.31

Listing 4 (p. 422) ends the **main** method and also ends the class definition for the class named **hello1** .

Listing 4: End of the class named hello1.

```
  }//end main
}//End hello1 class
```

4.32

The complete program listing

As mentioned earlier, a complete listing of the program is provided in Listing 5 (p. 424) near the end of the module.

4.14.6 General information

This program illustrates several general aspects of Java programming.

Overall skeleton of java program

The overall skeleton of any Java program consists of one or more class definitions.

All methods and variables must be defined inside a **class** definition. There can be no freestanding methods or global variables.

File names and extensions

The name of the *controlling class* should be the same as the name of the source file that contains it.

Files containing source code in Java have an extension of *java* .

The main method

The controlling class definition for an application must contain the **main** method.

The primary class file

The file produced by compiling the file containing the controlling class has the same name as the controlling class, and has an extension of **class** .

Many class files may be produced

The java compiler produces a separate file for every class definition contained in an application or applet, even if two or more class definitions are contained in the same source file.

Thus, the compilation of a large application can produce many different *class* files.

What are jar files?

A feature known as a *jar* file can be used to consolidate those class files into a single file for more compact storage, distribution, and transmission. Such a file has an extension of **jar** .

The main method is static

The controlling class for a Java application must contain a ***static*** method named **main** .

When you run the application using the JVM, you specify the name of the *class* file that you want to run.

The JVM then calls the **main** method defined in the *class* file having that name. This is possible because a *class method* can be called without a requirement to instantiate an object of the class.

The **main** method defined in that class definition controls the flow of the program.

4.14.7 Run the program

I encourage you to copy the code from Listing 5 (p. 424) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.14.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0160: Java OOP: Hello World
- File: Jb0160.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

4.14.9 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 5 (p. 424) .

Listing 5: Complete program listing.

```
/*File hello1.java Copyright 1997, R.G.Baldwin
This is a Java application program .

When compiled, this program produces the class named:

hello1.class

When the Java interpreter is called upon the application's
controlling class using the following statement at the
command line:

java hello1

the interpreter starts the program by calling the main
method defined in the controlling class. The main method is
a class method which can be called without the requirement
to instantiate an object of the class.

The program displays the following words on the screen:

Hello World

**********************************************************/
class hello1 { //define the controlling class
  //define main method
  public static void main(String[] args){
    //display text string
    System.out.println("Hello World");
  }//end main
}//End hello1 class.
```

4.33

-end-

4.15 Jb0160r Review[51]

4.15.1 Table of Contents

4.15.2 Preface

This module contains review questions and answers keyed to the module titled Jb0160: Java OOP: Hello World [52] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.15.3 Questions

4.15.3.1 Question 1 .

True or false? Applications are designed to be downloaded and executed on-line under control of a web browser, while applets are designed to be executed in a stand-alone mode from the command line.

Answer 1 (p. 431)

4.15.3.2 Question 2

True or false? All applications and applets written in Java require a **main** method.

Answer 2 (p. 430)

4.15.3.3 Question 3

Explain the relationship between the name of the class file for a Java application and the location of the **main** method in the application.

Answer 3 (p. 430)

4.15.3.4 Question 4

Explain how you cause a method to be a *class* method in Java.

Answer 4 (p. 430)

4.15.3.5 Question 5

True or false? *Class* methods can be called without the requirement to instantiate an object of the class:

Answer 5 (p. 430)

[51] This content is available online at <http://cnx.org/content/m45159/1.7/>.
[52] http://cnx.org/content/m45143

4.15.3.6 Question 6

Write the source code for a Java application that will display your name and address on the standard output device. Show the command-line statement that would be required to execute a compiled version of your application.

Answer 6 (p. 430)

4.15.3.7 Question 7

Show the three styles of comment indicators that are supported by Java.

Answer 7 (p. 429)

4.15.3.8 Question 8

True or false? Java allows free-standing methods outside of a class definition?

Answer 8 (p. 429)

4.15.3.9 Question 9

What is the relationship between the name of the *controlling class* in an application and the names of the files that comprise that application.

Answer 9 (p. 429)

4.15.3.10 Question 10

What is the relationship between the number of classes in an application and the number of separate files with the ***class*** extension that go to make up that application? How does this change when all the classes are defined in the same source file?

Answer 10 (p. 429)

4.15.3.11 Question 11

True or false? ***Class*** methods in Java can only be called relative to a specific object.

Answer 11 (p. 429)

4.15.3.12 Question 12

Write the signature line for the main method in a Java application.

Answer 12 (p. 429)

4.15.3.13 Question 13

Write a Java application that will display your name on the screen.

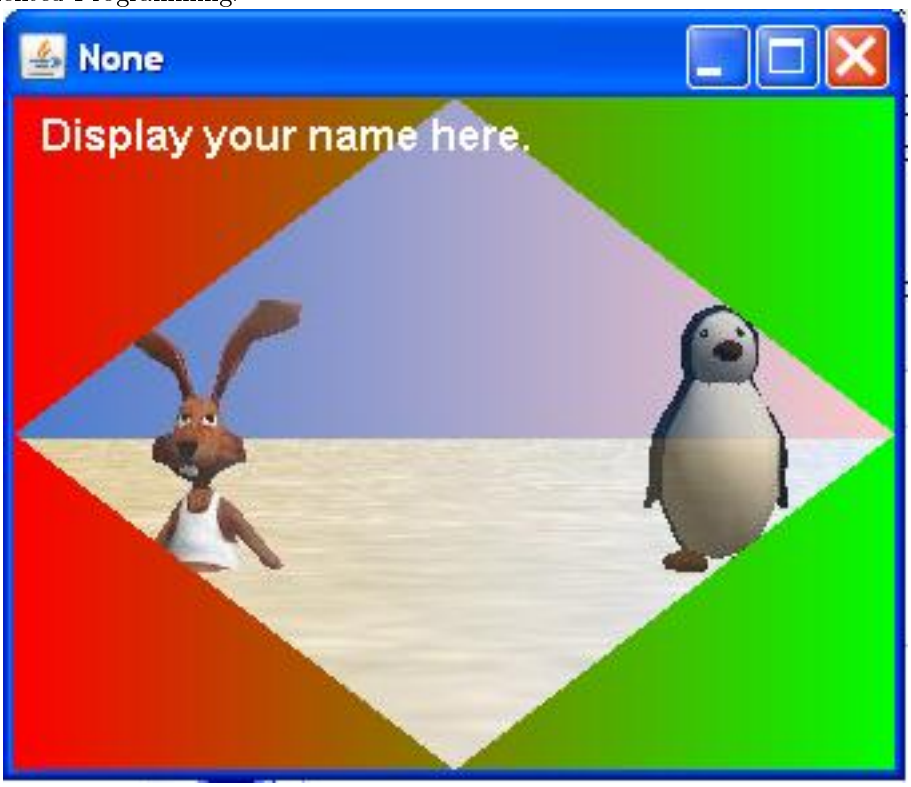
Answer 13 (p. 428)

4.15.4 Listings

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.15.5 Answers

4.15.5.1 Answer 13

Listing 1: Listing for Answer 13.

```
/*File SampProg02.java from lesson 10
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write an
application that will display your name on the screen.
**********************************************************/
class SampProg02 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("Dick Baldwin");
  }//end main
}//End SampProg02 class.
```

4.34

Back to Question 13 (p. 426)

4.15.5.2 Answer 12

NOTE:

```
public static void main(String[] args)
```

Back to Question 12 (p. 426)

4.15.5.3 Answer 11

False. ***Class*** methods can be called by joining the name of the class with the name of the method using a period.

Back to Question 11 (p. 426)

4.15.5.4 Answer 10

Each class definition results in a separate ***class*** file regardless of whether or not the classes are defined in separate source files.

Back to Question 10 (p. 426)

4.15.5.5 Answer 9

One of the files must have the same name as the *controlling class* with an extension of ***class*** .

Back to Question 9 (p. 426)

4.15.5.6 Answer 8

False.

Back to Question 8 (p. 426)

4.15.5.7 Answer 7

Listing 2: Listing for Answer 7.

```
/** special documentation comment used by the JDK javadoc tool */
/* C/C++ style multi-line comment */
// C/C++// C/C++ style single-line comment
```

4.35

Back to Question 7 (p. 426)

4.15.5.8 Answer 6

Listing 3: Listing for Answer 6.

```
/*File Name01.java
This is a Java application that will display a
name on the standard output device.

The command required at the command line to execute this
program is:

java Name01

**********************************************************/

class Name01 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println(
     "Dick Baldwin\nAustin Community College\nAustin, TX");
  }//end main
}//End Name01 class.
```

4.36

Note that the **\n** characters in Listing 3 (p. 430) cause the output display to advance to the next line.
Back to Question 6 (p. 426)

4.15.5.9 Answer 5

True.
Back to Question 5 (p. 425)

4.15.5.10 Answer 4

Preface or precede the name of the method with the **static** keyword.
Back to Question 4 (p. 425)

4.15.5.11 Answer 3

The name of the class file must be the same as the name of the class that contains the **main** method *(sometimes called the controlling class)* .
Back to Question 3 (p. 425)

4.15.5.12 Answer 2

False. Applets do not require a **main** method while applications do require a **main** method.
Back to Question 2 (p. 425)

4.15.5.13 Answer 1

False. Applications are for stand-alone use while applets are for browsers.
Back to Question 1 (p. 425)

4.15.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0160r Review: Hello World
- File: Jb0160r.htm
- Published: 11/18/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.16 Jb0170: Java OOP: A little more information about classes.[53]

4.16.1 Table of Contents

4.16.2 Preface

This module is part of a sub-collection of modules designed to help you learn to program computers.
This module sheds a little more light on the Java construct called a *class* .

4.16.3 Listings

[53] This content is available online at <http://cnx.org/content/m45144/1.2/>.

4.16.4 Introduction

New types

Java makes extensive use of classes. When a class is defined in Java, a new *type* comes into being. The new type definition can then be used to instantiate *(create instances of)* one or more objects of that new type.

A blueprint

The class definition provides a *blueprint* that describes the *data* contained within, and the *behavior* of objects instantiated according to the new type.

The data

The data is contained in variables defined within the class *(often called member variables, data members, attributes, fields, properties, etc.*).

The behavior

The behavior is controlled by methods defined within the class.

State and behavior

An object is said to have *state* and *behavior* . At any instant in time, the *state* of an object is determined by the values stored in its *variables* and its behavior is determined by its *methods* .

Class vs. instance

It is possible to define:

- instance variables and instance methods
- static or *class* variables and static or *class* methods.

Instance variables and instance methods can only be accessed through an object instantiated from the class. They belong to the individual objects, *(which is why they are called instance variables and instance methods)* .

Class variables and *class* methods can be accessed without first instantiating an object. They are shared among all of the objects instantiated from the class and are even accessible in the total absence of an object of the class.

The class name alone is sufficient for accessing *class* variables and *class* methods by joining the name of the class to the name of the variable or method using a period.

4.16.5 Defining a class in Java

The general syntax for defining a class in Java is shown in Listing 1 (p. 432) .

Listing 1: General syntax for defining a Java class.

```
class MyClassName{
  . . .
} //End of class definition.
```

4.37

This syntax defines a class and creates a new type named **MyClassName** .

The definitions of variables, methods, constructors, and a variety of other members are inserted between the opening and closing curly brackets.

4.16.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0170: Java OOP: A little more information about classes.
- File: Jb0170.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.17 Jb0170r: Review[54]

4.17.1 Table of Contents

4.17.2 Preface

This module contains review questions and answers keyed to the module titled Jb0170: Java OOP: A little more information about classes [55] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.17.3 Questions

4.17.3.1 Question 1 .

List two of the many names commonly used for variables defined within a class in Java.

Answer 1 (p. 437)

4.17.3.2 Question 2

List two of the many names commonly used for the functions defined within a class in Java.

Answer 2 (p. 437)

4.17.3.3 Question 3

An object is said to have *state* and *behavior* . At any instant in time, the *state* of an object is determined by the values stored in its (a)_____ and its behavior is determined by its (b)_____.

Answer 3 (p. 437)

4.17.3.4 Question 4

What keyword is used to cause a variable or method to become a *class* variable or *class* method in Java?

Answer 4 (p. 436)

4.17.3.5 Question 5

True or false? *Instance* variables and *instance* methods can only be accessed through an object of the class in Java.

Answer 5 (p. 436)

4.17.3.6 Question 6

True or false? In Java, the class name alone is sufficient for accessing *class* variables and *class* methods by joining the name of the class with the name of the variable or method using a colon.

Answer 6 (p. 436)

[54] This content is available online at <http://cnx.org/content/m45177/1.4/>.
[55] http://cnx.org/content/m45144

4.17.3.7 Question 7

True or false? Show the general syntax of an empty class definition in Java.

Answer 7 (p. 436)

4.17.3.8 Question 8

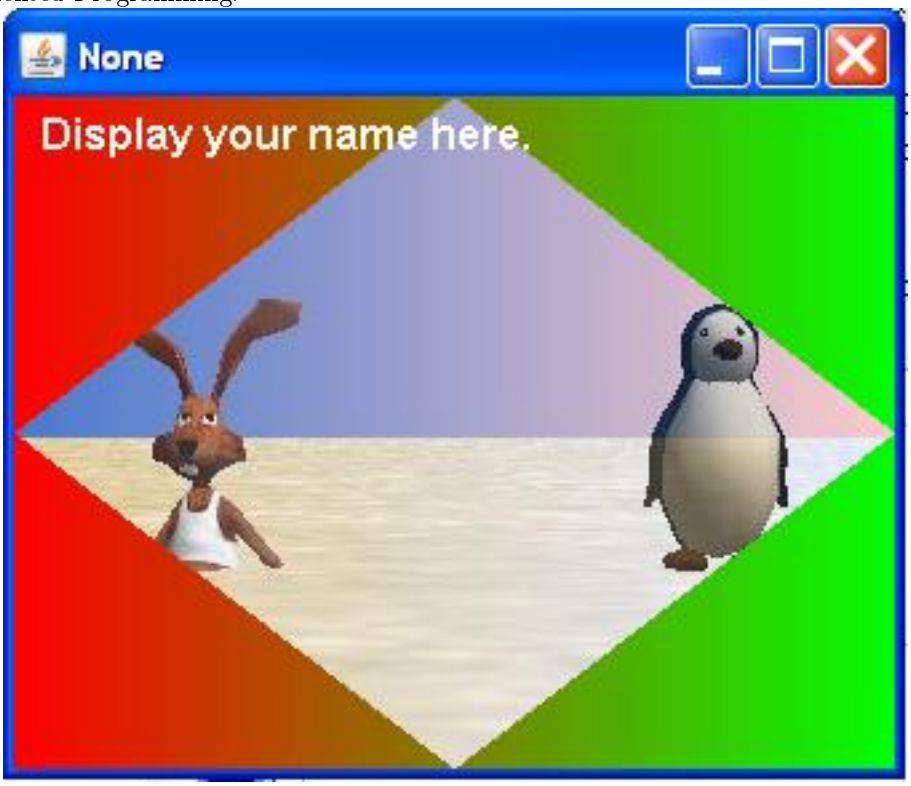
True or false? The syntax for a class definition in Java requires a semicolon following the closing curly bracket.

Answer 8 (p. 436)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.17.4 Answers

4.17.4.1 Answer 8

False. Java does not require the use of a semicolon following the closing curly bracket in a class definition.
Back to Question 8 (p. 435)

4.17.4.2 Answer 7

class NameOfClass{}
Back to Question 7 (p. 435)

4.17.4.3 Answer 6

False. A colon is not used in Java. Instead, a period is used in Java.
Back to Question 6 (p. 434)

4.17.4.4 Answer 5

True.
Back to Question 5 (p. 434)

4.17.4.5 Answer 4

static
Back to Question 4 (p. 434)

4.17.4.6 Answer 3

- (a) instance variables
- (b) methods

Back to Question 3 (p. 434)

4.17.4.7 Answer 2

Member functions and instance methods.
Back to Question 2 (p. 434)

4.17.4.8 Answer 1

Instance variables and attributes.
Back to Question 1 (p. 434)

4.17.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0170r: Review: A little more information about classes
- File: Jb0170r.htm
- Published: 11/21/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.18 Jb0180: Java OOP: The main method.[56]

4.18.1 Table of Contents

[56] This content is available online at <http://cnx.org/content/m45145/1.2/>.

4.18.2 Preface

This module is part of a sub-collection of modules designed to help you learn to program computers.

Every Java application requires a class containing a method named **main** . This module provides information on the **main** method.

4.18.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images while you are reading about them.

4.18.2.1.1 Images

4.18.3 The main method in Java

There must be a **main** method in the controlling class in every Java application.

The method signature

The Java literature frequently refers to the signature of a method, or the *method signature* .

Exploring Java by Patrick Niemeyer and Joshua Peck (O'Reilly) provides the definition of a method signature shown in Image 1 (p. ??) .

"A method signature is a collection of information about the method, as in a C prototype or a forward function declaratic

4.38

Type

Apparently in this definition, the authors are referring to the *type* of the method as distinguishing between static and non-static. *(Other literature refers to the type of a function or method as being the return type which according to the above definition is a separate part of the signature.)*

Visibility

Apparently also the use of the word visibility in the above definition refers to the use of **public** , **private** , etc.

According to Oracle...

Oracle's Java Tutorials [57] , on the other hand, describe the method signature as in Image 2 (p. ??) .

[57] http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

Definition: Two of the components of a method declaration comprise the method signature–the method's name and the pa

4.39

As you can see, the Oracle definition is more restrictive than the Niemeyer and Peck definition.

Bottom line on method signature

The method signature can probably be thought of as providing information about the programming interface to the method. In other words, it provides the information that you need to be able to call the method in your program code.

Signature of main method

The controlling class of every Java application must contain a **main** method having one of the signatures shown in Image 3 (p. 439) .

Image 3: Allowable signatures for the main method.

```
public static void main(String[] args)

public static void main(String args[])
```

4.40

(I prefer the first signature in Image 3 (p. 439) as being the most descriptive of an array of ***String*** *references which is what is passed in as an argument.)*

public

The keyword **public** indicates that the method can be called by any object. A future module will discuss the keywords **public** , **private** , and **protected** in more detail.

static

The keyword **static** indicates that the method is a *class* method, which can be called without the requirement to instantiate an object of the class. This is used by the JVM to launch the program by calling the **main** method of the class identified in the command to start the program.

void

The keyword **void** indicates that the method doesn't return any value.

args

The formal parameter **args** is a reference to an array object of type **String** . The array elements contain references to **String** objects that encapsulate **String** representations of the arguments, if any, entered at the command line.

Note that the **args** parameter must be specified whether or not the user is required to enter command-line arguments and whether or not the code in the program actually makes use of the argument. Also note that the name can be any legal Java identifier. It doesn't have to be **args** . It could be joe or sue, for example.

The length property

The parameter named **args** is a reference to an array object. Java array objects have a property named **length** , which specifies the number of elements in the array.

The runtime system monitors for the entry of command-line arguments by the user and constructs the **String** array containing those arguments.

Processing command-line arguments

The **args.length** property can be used by the code in the program to determine the number of arguments actually entered by the user.

If the **length** property is not equal to zero, the first string in the array corresponds to the first argument entered on the command line.

Command-line arguments along with strings and **String** arrays will be discussed in more detail in a future module.

4.18.4 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0180: Java OOP: The main method.
- File: Jb0180.htm
- Originally published: 1997
- Published at cnx.org: 11/17/12
- Revised: 01/02/13

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.19 Jb0180r Review[58]

4.19.1 Table of Contents

4.19.2 Preface

This module contains review questions and answers keyed to the module titled Jb0180: Java OOP: The main method [59] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.19.3 Questions

4.19.3.1 Question 1 .

Write the method signature for the **main** method in a Java application.

Answer 1 (p. 444)

4.19.3.2 Question 2

Briefly explain the reason that the **main** method in a Java application is declared **public** .

Answer 2 (p. 444)

4.19.3.3 Question 3

Explain the reason that the **main** method in a Java application must be declared **static** .

Answer 3 (p. 444)

4.19.3.4 Question 4

Describe the purpose of the keyword **void** when used as the return type for the **main** method.

Answer 4 (p. 444)

4.19.3.5 Question 5

True or false? If the Java application is not designed to use command-line arguments, it is not necessary to include a formal parameter for the **main** method.

Answer 5 (p. 443)

4.19.3.6 Question 6

True or false? When using command-line arguments in Java, if the name of the string array is **args** , the **args.length** variable can be used by the code in the program to determine the number of arguments actually entered.

Answer 6 (p. 443)

[58] This content is available online at <http://cnx.org/content/m45171/1.4/>.
[59] http://cnx.org/content/m45145

4.19.3.7 Question 7

True or false? The first string in the array of command-line arguments contains the name of the Java application

Answer 7 (p. 443)

4.19.3.8 Question 8

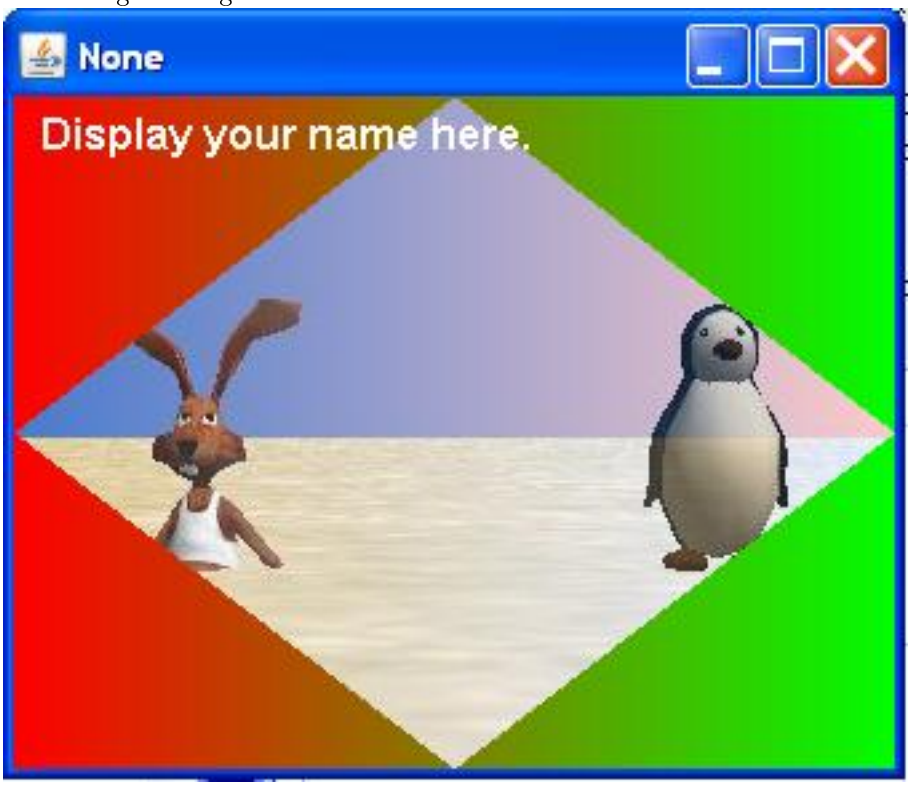
The *controlling class* of every Java application must contain a **main** method. Can other classes in the same application also have a **main** method? If not, why not? If so, why might you want to do that?

Answer 8 (p. 443)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.19.4 Answers

4.19.4.1 Answer 8

Any and all classes in a Java application can have a **main** method. Only the one in the *controlling class* for the program being executed is actually called.

It is often desirable to provide a **main** method for a class that will not ultimately be the *controlling class* to allow the class to be tested in a stand-alone mode, independent of other classes.

Back to Question 8 (p. 442)

4.19.4.2 Answer 7

False. Unlike C++, the first string in the array of command-line arguments in a Java application does **not** contain the name of the application.

Back to Question 7 (p. 442)

4.19.4.3 Answer 6

True.

Back to Question 6 (p. 441)

4.19.4.4 Answer 5

False. The **main** method in a Java program must always provide the formal argument list regardless of whether it is actually used in the program.

Back to Question 5 (p. 441)

4.19.4.5 Answer 4

The **void** keyword when used as the return type for any Java method indicates that the method does not **return** anything.

Back to Question 4 (p. 441)

4.19.4.6 Answer 3

The keyword **static** indicates that the method is a *class* method which can be called without the requirement to instantiate an object of the class. This is used by the Java virtual machine to launch the program by calling the **main** method of the class identified in the command to start the program.

Back to Question 3 (p. 441)

4.19.4.7 Answer 2

The keyword **public** indicates that the method can be called by any object.

Back to Question 2 (p. 441)

4.19.4.8 Answer 1

NOTE:

```
public static void main(String[] args)
```

Back to Question 1 (p. 441)

4.19.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0180r Review: The main method
- File: Jb0180r.htm
- Published: 11/21/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.20 Jb0190: Java OOP: Using the System and PrintStream Classes[60]

4.20.1 Table of Contents

4.20.2 Preface

This module takes a preliminary look at the complexity of OOP by examining some aspects of the **System** and **PrintStream** classes.

4.20.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.20.2.1.1 Listings

4.20.3 Introduction

This lesson introduces you to the use of the **System** and **PrintStream** classes in Java. This is our first introduction to the complexity that can accompany the **OOP** paradigm. It gets a little complicated, so you might need to pay special attention to the discussion.

4.20.4 Discussion

What does the main method do?

The **main** method in the controlling class of a Java application controls the flow of the program.

The **main** method can also access other classes along with the variables and methods of those classes and of objects instantiated from those classes.

The hello1 Application

Listing 1 (p. 446) shows a simple Java application named **hello1** .

(By convention, class definitions should begin with an upper-case character. However, the original version of this module was written and published in 1997, before that convention was firmly established.)

[60] This content is available online at <http://cnx.org/content/m45148/1.2/>.

Listing 1: The program named hello1.

```
/*File hello1.java Copyright 1997, R.G.Baldwin
************************************************************/
class hello1 { //define the controlling class
  //define main method
  public static void main(String[] args){
    //display text string
    System.out.println("Hello World");
  }//end main
}//End hello1 class.  No semicolon at end of Java class.
```

4.41

Does this program Instantiate objects?

This is a simple example that does not instantiate objects of any other class.

Does this program access another class?

However, it does access another class. It accesses the **System** class that is provided with the Java development kit. *(The **System** class will be discussed in more detail in a future module.)*

The variable named *out*

The variable named **out**, referred to in Listing 1 (p. 446) as **System.out**, is a *class variable* of the **System** class. *(A class variable is a variable that is declared to be static.)*

Recall that a class variable can be accessed without a requirement to instantiate an object of the class. As is the case with all variables, the class variable must be of some specific type.

Primitive variables vs. reference variables

A class variable may be a *primitive variable*, which contains a primitive value, or it may be a *reference variable*, which contains a reference to an object.

(I'll have more to say about the difference between primitive variables and reference variables in a future module.)

The variable named **out** in this case is a *reference variable*, which refers to an object of another type.

Accessing class variables

You access class variables or class methods in Java by joining the name of the class to the name of the variable or method with a period.

NOTE: System.out

accesses the class variable named **out** in the Java class named **System**.

The PrintStream class

Another class that is provided with the Java development kit is the **PrintStream** class. The **PrintStream** class is in a package of classes that are used to provide stream input/output capability for Java.

What does the out variable refer to?

The **out** variable in the **System** class refers to *(points to)* an instance of the **PrintStream** class *(a **PrintStream** object)*, which is automatically instantiated when the **System** class is loaded into the application.

We will be discussing the **PrintStream** class along with a number of other classes in detail in a future module on input/output streams, so this is not intended to be an exhaustive discussion.

The println method

The **PrintStream** class has an *instance method* named **println** , which causes its argument to be displayed on the standard output device when it is called.

(Typically, the standard output device is the command-line window. However, it is possible to redirect it to some other device.)

Accessing an instance method

The method named **println** can be accessed by joining a **PrintStream** object's reference to the name of the method using a period.

Thus, *(assuming that the standard output device has not been redirected)* , the statement shown in Listing 2 (p. 447) causes the string "Hello World" *(without the quotation marks)* to be displayed in the command-line window.

Listing 2: Display the string "Hello World".

```
System.out.println("Hello World");
```

4.42

This statement calls the **println** method of an object instantiated from the **PrintStream** class, which is referred to *(pointed to)* by the variable named **out** , which is a *class variable* of the **System** class.

Read the previous paragraph very carefully. As I indicated when I started this module, this is our first introduction to the complexity that can result from use of the OOP paradigm. *(It can get even more complicated.)* If this is not clear to you, go back over it and think about it until it becomes clear.

4.20.5 A word about class variables

How many instances of a class variable exist?

The runtime system allocates a class variable only once no matter how many instances *(objects)* of the class are instantiated.

All objects of the class share the same physical memory space for the class variable.

If a method in one object changes the value stored in the class variable, it is changed insofar as all of the objects are concerned. *(This is about as close to a global variable as you can get in Java.)*

Accessing a class variable

You can use the name of the class to access class variables by joining the name of the class to the name of the variable using a period.

You can also access a class variable by joining the name of a reference variable containing an object's reference to the name of the variable using a period as the joining operator.

Referencing object methods via class variables

Class variables are either primitive variables or reference variables. *(Primitive variables contain primitive values and reference variables contain references to objects.)*

A referenced object may provide methods to control the behavior of the object. In Listing 2 (p. 447) , we accessed the **println** method of an object of the **PrintStream** class referred to by the class variable named **out** .

Instance variables and methods

As a side note, in addition to class variables, Java provides *instance variables* and *instance methods* . Every instance of a class has its own set of instance variables. You can only access instance variables and instance methods through an object of the class.

4.20.6 Run the program

I encourage you to copy the code from Listing 1 (p. 446) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.20.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0190: Java OOP: Using the System and PrintStream Classes
- File: Jb0190.htm
- Originally published: 1997
- Published at cnx.org: 11/18/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.21 Jb0190r: Review[61]

4.21.1 Table of Contents

4.21.2 Preface

This module contains review questions and answers keyed to the module titled Jb0190: Java OOP: Using the System and PrintStream Classes [62] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.21.3 Questions

4.21.3.1 Question 1 .

True or false? The **main** method in the controlling class of a Java application controls the flow of the program.

Answer 1 (p. 455)

4.21.3.2 Question 2

True or false? The **main** method cannot access the variables and methods of objects instantiated from other classes.

Answer 2 (p. 454)

4.21.3.3 Question 3

True or false? The **main** method must instantiate objects of other classes in order for the program to execute.

Answer 3 (p. 454)

4.21.3.4 Question 4

True or false? In order to be useful, the **System** class must be used to instantiate objects in a Java application.

Answer 4 (p. 454)

4.21.3.5 Question 5

True or false? *Class* variables such as the **out** variable of the **System** class must be of some specific type.

Answer 5 (p. 454)

[61] This content is available online at <http://cnx.org/content/m45175/1.4/>.
[62] http://cnx.org/content/m45148

4.21.3.6 Question 6

True or false? *Class* variables must be of a primitive type such as **int** or **float** .
Answer 6 (p. 454)

4.21.3.7 Question 7

True or false? The **out** variable in the **System** class is of a primitive type.
Answer 7 (p. 454)

4.21.3.8 Question 8

True or false? What does the following code fragment access?

NOTE:

```
System.out
```

Answer 8 (p. 454)

4.21.3.9 Question 9

True or false? An object of type **PrintStream** is automatically instantiated when the **System** class is loaded into an application.
Answer 9 (p. 454)

4.21.3.10 Question 10

True or false? The **out** variable in the **System** class refers to an instance of what class?
Answer 10 (p. 454)

4.21.3.11 Question 11

True or false? The **println** method is an instance method of what class?
Answer 11 (p. 453)

4.21.3.12 Question 12

What is the primary behavior of the **println** method?
Answer 12 (p. 453)

4.21.3.13 Question 13

How can the **println** method be accessed?
Answer 13 (p. 453)

4.21.3.14 Question 14

Assuming that the standard output device has not been redirected, write a code fragment that will cause your name to be displayed on the screen.
Answer 14 (p. 453)

4.21.3.15 Question 15

Explain how your code fragment in Answer 14 (p. 453) produces the desired result.

Answer 15 (p. 453)

4.21.3.16 Question 16

If you have a class named **MyClass** that has a class variable named **myClassVariable** that requires four bytes of memory and you instantiate ten objects of type **MyClass**, how much total memory will be allocated to contain the allocated variables *(assume that the class definition contains no other class, instance, or local variables)*.

Answer 16 (p. 453)

4.21.3.17 Question 17

How many actual instances of the variable named **out** are allocated in memory by the following code fragment?

NOTE:

```
System.out.println("Dick Baldwin");
```

Answer 17 (p. 453)

4.21.3.18 Question 18

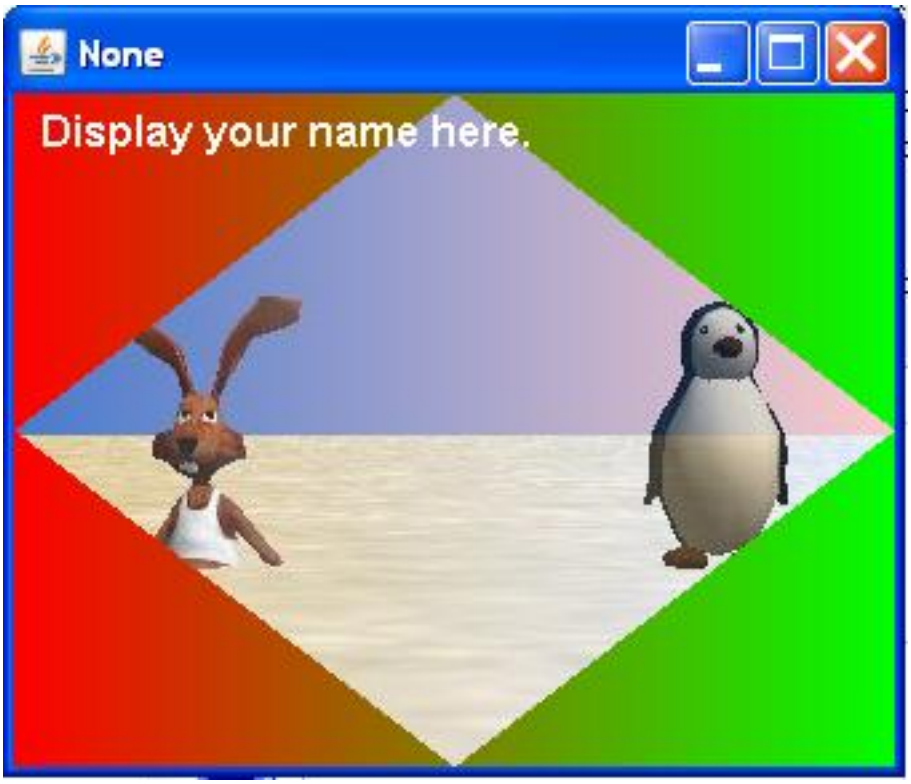
If you have a class named **MyClass** that has an instance variable named **myInstanceVariable** that requires four bytes of memory and you instantiate ten objects of type **MyClass**, how much total memory will be allocated to contain the allocated variables *(assume that the class definition contains no other class, instance, or local variables)*.

Answer 18 (p. 453)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.21.4 Answers

4.21.4.1 Answer 18

Every instance of a class has its own set of instance variables. You can only access *instance variables* and *instance methods* through an object of the class. In this case, forty bytes of memory would be required to contain the instance variables of the ten objects.

Back to Question 18 (p. 451)

4.21.4.2 Answer 17

Only one, because **out** is a class variable of the **System** class.

Back to Question 17 (p. 451)

4.21.4.3 Answer 16

The runtime system allocates a *class variable* only once no matter how many instances of the class are instantiated. Thus, all objects of the class share the same physical memory space for the class variable, and in this case, only four bytes of memory will be allocated to contain the allocated variables.

Back to Question 16 (p. 451)

4.21.4.4 Answer 15

The statement in Answer 14 (p. 453) calls the **println** method belonging to an object of the **PrintStream** class, which is referenced *(pointed to)* by the **out** variable, which is a *class* variable of the **System** class.

Back to Question 15 (p. 451)

4.21.4.5 Answer 14

NOTE:

```
System.out.println("Dick Baldwin");
```

Back to Question 14 (p. 450)

4.21.4.6 Answer 13

The **println** method can be accessed by joining the name of a variable that references a **PrintStream** object to the name of the **println** method using a period.

Back to Question 13 (p. 450)

4.21.4.7 Answer 12

The **println** method causes its argument to be displayed on the standard output device. *(The standard output device is the screen by default, but can be redirected by the user at the operating system level.)*

Back to Question 12 (p. 450)

4.21.4.8 Answer 11

The **println** method is an instance method of the **PrintStream** class.

Back to Question 11 (p. 450)

4.21.4.9 Answer 10

The **out** variable in the **System** class refers to an instance of the **PrintStream** class *(a **PrintStream** object)*, which is automatically instantiated when the **System** class is loaded into the application.

Back to Question 10 (p. 450)

4.21.4.10 Answer 9

True.

Back to Question 9 (p. 450)

4.21.4.11 Answer 8

The code fragment accesses the contents of the *class* variable named **out** in the class named **System** .

Back to Question 8 (p. 450)

4.21.4.12 Answer 7

False. the variable named **out** defined in the **System** class is a reference variable that points to an object of another type.

Back to Question 7 (p. 450)

4.21.4.13 Answer 6

False. A *class* variable can be a primitive type, or it can be a reference variable that points to another object.

Back to Question 6 (p. 450)

4.21.4.14 Answer 5

True.

Back to Question 5 (p. 449)

4.21.4.15 Answer 4

False. The **System** class has several *class* variables *(including **out** and **in**)* that are useful without the requirement to instantiate an object of the **System** class.

Back to Question 4 (p. 449)

4.21.4.16 Answer 3

False. While it is probably true that the **main** method must instantiate objects of other classes in order to accomplish much that is of value, this is not a requirement. The **main** method in the "Hello World" program of this module [63] does not instantiate objects of any class at all.

Back to Question 3 (p. 449)

4.21.4.17 Answer 2

False. The **main** method can access the variables and methods of objects instantiated from other classes. Otherwise, the flow of the program would be stuck within the **main** method itself and wouldn't be very useful.

Back to Question 2 (p. 449)

[63] http://cnx.org/content/m45148/latest/

4.21.4.18 Answer 1

True.

Back to Question 1 (p. 449)

4.21.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0190r: Review: Using the System and PrintStream Classes
- File: Jb0190r.htm
- Published: 11/22/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.22 Jb0200: Java OOP: Variables[64]

4.22.1 Table of Contents

[64] This content is available online at <http://cnx.org/content/m45150/1.2/>.

4.22.2 Preface

Earlier modules have touched briefly on the topic of variables. This module discusses Java variables in depth.

4.22.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

4.22.2.1.1 Images

4.22.2.1.2 Listings

4.22.3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as

- variables
- types
- expressions
- flow-of-control, etc.

This and several future modules concentrate on that foundation.

A sample program

The module begins with a sample Java program named **simple1** . The user is asked to enter some text and to terminate with the # character.

(This program contains a lot of code that you are not yet prepared to understand. For the time being, just concentrate on the use of variables in the program. You will learn about the other aspects of the program in future modules.)

The program loops, saving individual characters until encountering the # character. When it encounters the # character, it terminates and displays the character entered immediately prior to the # character.

4.22.4 Sample program named simple1

A complete listing of the program named **simple1** is provided in Listing 1 (p. 458) . Discussions of selected portions of the program are presented later in the module.

Listing 1: Source code for the program named simple1.

```
/*File simple1.java Copyright 1997, R.G.Baldwin
This Java application reads bytes from the keyboard until
encountering the integer representation of '#'.  At
the end of each iteration, it saves the byte received and
goes back to get the next byte.

When the '#' is entered, the program terminates input and
displays the character which was entered before the #.
**********************************************************/

class simple1 { //define the controlling class

  //It is necessary to declare that this method
  // can throw the exception shown below (or catch it).
  public static void main(String[] args) //define main
                       throws java.io.IOException {

    //It is necessary to initialize ch2 to avoid a compiler
    // error (possibly unitialized variable) at the
    // statement which displays ch2.
    int ch1, ch2 = '0';

    System.out.println(
                      "Enter some text, terminate with #");

    //Get and save individual bytes
    while( (ch1 = System.in.read() ) != '#')
      ch2 = ch1;

    //Display the character immediately before the #
    System.out.println(
          "The char before the # was " + (char)ch2);
  }//end main
}//End simple1 class.
```

4.43

Program output

The output produced by compiling and running this program is shown in Image 1 (p. 459) . The second line of text in Image 1 (p. 459) ending with the # character was typed by the user.

Image 1: Screen output from the program named simple1.

```
Enter some text, terminate with #
abcde#
The char before the # was e
```

4.44

4.22.4.1 Discussion of the simple1 program

Purpose

I will use the program shown in Listing 1 (p. 458) to discuss several important aspects of the structure of a Java program. I will also provide two additional sample programs that illustrate specific points not illustrated in the above program later in this module.

4.22.5 Variables

NOTE: **What is a variable?** Variables are used in a Java program to contain data that changes during the execution of the program.

Declaring a variable

To use a variable, you must first notify the compiler of the *name* and the *type* of the variable. This is known as *declaring a variable* .

The syntax for declaring a variable is to precede the *name* of the variable with the *name of the type* of the variable as shown in Listing 2 (p. 459) . It is also possible *(but not required)* to initialize a variable in Java when it is declared as shown in Listing 2 (p. 459) .

Listing 2: Declaring and initializing two variables named ch1 and ch2.

```
int ch1, ch2 = '0';
```

4.45

The statement in Listing 2 (p. 459) declares two variables of type **int** , initializing the second variable *(ch2)* to the value of the zero character (0). *(Note that I didn't say initialized to the value zero.)*

NOTE: **Difference between zero and '0' - Unicode characters** The value of the zero character is not the same as the numeric value of zero, but hopefully you already knew that.

As an aside, characters in Java are 16-bit entities called Unicode characters instead of 8-bit entities as is the case with many programming languages. The purpose is to provide many more possible characters including characters used in alphabets other than the one used in the United States.

Initialization of the variable

Initialization of the variable named **ch2** in this case was necessary to prevent a compiler error. Without initialization of this variable, the compiler would recognize and balk at the possibility that an attempt might be made to execute the statement shown in Listing 3 (p. 460) with a variable named **ch2** that had not been initialized

---

Listing 3: Display the character.

```
System.out.println("The char before the # was "
                                    + (char)ch2);
```

4.46

Error checking by the compiler

The strong error-checking capability of the Java compiler would refuse to compile this program until that possibility was eliminated by initializing the variable.

Using the cast operator

You should also note that the contents of the variable **ch2** is being *cast* as type **char** in Listing 3 (p. 460) .

(A cast is used to change the type of something to a different type.)

Recall that **ch2** is a variable of type **int** , containing the numeric value that represents a character.

We want to display the character that the numeric value represents and not the numeric value itself. Therefore, we must cast it *(purposely change its type for the evaluation of the expression)* . Otherwise, we would not see the character on the screen. Rather, we would see the numeric value that represents that character.

NOTE: **Initialization of instance variables and local variables:** As another aside, *member variables* in Java are automatically initialized to zero or the equivalent of zero. However, *local variables* , of which ch2 is an example, are not automatically initialized.

Why declare the variables as type int?

It was necessary to declare these variables as type **int** becausethe statement in Listing 4 (p. 461) *(more specifically, the call to the System.in.read method)* returns a value of type **int** .

Listing 4: Beginning of a while loop.

```
while( (ch1 = System.in.read() ) != '#') ch2 = ch1;
```

4.47

Java provides very strict type checking and generally refuses to compile statements with type mismatches.

(There is a lot of complicated code in Listing 4 (p. 461) that I haven't previously explained. I will explain that code later in this and future modules.)

Another variable declaration

The program in Listing 1 (p. 458) also makes another variable declaration shown by the statement in Listing 5 (p. 461) .

Listing 5: Beginning of the main method.

```
public static void main(String[] args) //define main method
```

4.48

An array of String references

In Listing 5 (p. 461) , the formal argument list of the **main** method declares an argument named **args** *(first cousin to a variable)* as a reference to an array object of type **String** .

Capturing command-line arguments in Java

As you learned in an earlier module, this is the feature of Java that is used to capture arguments entered on the command line, and is required whether arguments are entered or not. In this case, no command-line arguments were entered, and the variable named **args** is simply ignored by the remainder of the program.

The purpose of the type of a variable

NOTE: **All variables must have a declared type** The type determines the set of values that can be stored in the variable and the operations that can be performed on the variable.

For example, the **int** type can only contain whole numbers *(integers)* . A whole host of operations are possible with an **int** variable including add, subtract, divide, etc.

Signed vs. unsigned variables

Unlike C++, all variables of type **int** in Java contain signed values. In fact, with the exception of type **char** , all primitive numeric types in Java contain signed values.

Platform independence

At this point in the history of Java, a variable of a specified type is represented exactly the same way regardless of the platform on which the application or applet is being executed.

This is one of the features that causes compiled Java programs to be platform-independent.

4.22.5.1 Primitive types

In Java, there are two major categories of data types:

- primitive types
- reference *(or object)* types.

Primitive variables contain a single value of one of the eight primitive types shown in Listing 2 (p. 459) .

Reference variables contain references to objects *(or null, meaning that they don't refer to anything)* .

The eight primitive types in Java?

The table in Image 2 (p. 462) lists all of the primitive types in Java along with their size and format, and a brief description of each.

Image 2: Information about the primitive types in Java.

Type	Size/Format	Description
byte	8-bit two's complement	Byte-length integer
short	16-bit two's complement	Short integer
int	32-bit two's complement	Integer
long	64-bit two's complement	Long Integer
float	32-bit IEEE 754 format	Single-precision floating point
double	64-bit IEEE 754 format	Double-precision floating point
char	16-bit Unicode character	Single character
boolean	true or false	True or False

4.49

The char type

The **char** type is a 16-bit Unicode character value that has the possibility of representing more than 65,000 different characters.

Evaluating a primitive variable

A reference to the name of a primitive variable in program code evaluates to the value stored in the variable. In other words, when you call out the name of a primitive variable in your code, what you get back is the value stored in the variable.

4.22.5.1.1 Object-oriented wrappers for primitive types

Primitive types are not objects

Primitive data types in Java *(int, double, etc.)* are not objects. This has some ramifications as to how they can be used *(passing to methods, returning from methods, etc.)* .

The generic Object type

Later on in this course of study, you will learn that much of the power of Java derives from the ability to deal with objects of any type as the generic type **Object** . For example, several of the standard classes in the API *(such as the powerful Vector class)* are designed to work only with objects of type **Object** .

(Note that this document was originally published prior to the introduction of generics in Java. The introduction of generics makes it possible to cause the ***Vector*** *class to deal with objects of types other than* ***Object*** *. However, that doesn't eliminate the need for wrapper classes.)*

Converting primitives to objects

Because it is sometimes necessary to deal with a primitive value as though it were an object, Java provides wrapper classes that support object-oriented functionality for Java's primitive data types.

The Double wrapper class

This is illustrated in the program shown in Listing 6 (p. 464) that deals with a **double** type as an object of the class **Double** .

(Remember, Java is a case-sensitive language. Note the difference between the primitive ***double*** *type and the class named* ***Double*** *.)*

Listing 6: The program named wrapper1.

```
/*File wrapper1.java Copyright 1997, R.G.Baldwin
This Java application illustrates the use of wrappers
for the primitive types.

This program produces the following output:

My wrapped double is 5.5
My primitive double is 10.5

**********************************************************/
class wrapper1 { //define the controlling class
  public static void main(String[] args){//define main

    //The following is the declaration and instantiation of
    // a Double object, or a double value wrapped in an
    // object. Note the use of the upper-case D.
    Double myWrappedData = new Double(5.5);

    //The following is the declaration and initialization
    // of a primitive double variable. Note the use of the
    // lower-case d.
    double myPrimitiveData = 10.5;

    //Note the call to the doubleValue() method to obtain
    // the value of the double wrapped in the Double
    // object.
    System.out.println(
      "My wrapped double is " + myWrappedData.doubleValue());
    System.out.println(
             "My primitive double is " + myPrimitiveData );
  }//end main
}//End wrapper1 class.
```

4.50

The operation of this program is explained in the comments, and the output from the program is shown in the comments at the beginning.

4.22.5.2 Reference types

Once again, what is a primitive type?

Primitive types are types where the name of the variable evaluates to the value stored in the variable.

What is a reference type?

Reference types in Java are types where the name of the variable evaluates to the address of the location in memory where the object referenced by the variable is stored.

NOTE: **The above statement may not really be true?** However, we can think of it that way. Depending on the particular JVM in use, the reference variable may refer to a table in memory where the address of the object is stored. In that case the second level of indirection is handled behind the scenes and we don't have to worry about it.

Why would a JVM elect to implement another level of indirection? Wouldn't that make programs run more slowly?

One reason has to do with the need to compact memory when it becomes highly fragmented. If the reference variables all refer directly to memory locations containing the objects, there may be many reference variables that refer to the same object. If that object is moved for compaction purposes, then the values stored in every one of those reference variables would have to be modified.

However, if those reference variables all refer to a table that has one entry that specifies where the object is stored, then when the object is moved, only the value of that one entry in the table must be modified.

Fortunately, that all takes place behind the scenes and we as programmers don't need to worry about it.

Primitive vs. reference variables

We will discuss this in more detail in a future module. For now, suffice it to say that in Java, a variable is either a primitive type or a reference type, and cannot be both.

Declaring, instantiating, initializing, and manipulating a reference variable

The fragment of code shown in Listing 7 (p. 465) , *(which was taken from the program shown in Listing 6* (p. 464) *that deals with wrappers)* does the following. It

- declares,
- instantiates,
- initializes, and
- manipulates a variable of a reference type named **myWrappedData** .

In Listing 7 (p. 465) , the variable named **myWrappedData** contains a reference to an object of type **Double** .

Listing 7: Aspects of using a wrapper class.

```
Double myWrappedData = new Double(5.5);

//Code deleted for brevity

//Note the use of the doubleValue() method to obtain the
// value of the double wrapped in the Double object.
System.out.println
  ("My wrapped double is " + myWrappedData.doubleValue() );
```

4.51

4.22.5.3 Variable names

The rules for naming variables are shown in Image 3 (p. 466) .

Image 3: Rules for naming variables.

- Must be a legal Java identifier *(see below)* consisting of a series of Unicode characters. Unicode characters are stored in sixteen bits, allowing for a very large number of different characters. A subset of the possible character values matches the 127 possible characters in the ASCII character set, and the extended 8-bit character set, ISO-Latin-1 *(The Java Handbook, page 60, by Patrick Naughton)*.
- Must not be the same as a Java keyword and must not be true or false.
- Must not be the same as another variable whose declaration appears in the same scope.

4.52

The rules for legal identifiers are shown in Image 4 (p. 466) .

Image 4: Rules for legal identifiers.

- In Java, a legal identifier is a sequence of Unicode letters and digits of unlimited length.
- The first character must be a letter.
- All subsequent characters must be letters or numerals from any alphabet that Unicode supports.
- In addition, the underscore character (_) and the dollar sign ($) are considered letters and may be used as any character including the first one.

4.53

4.22.6 Scope

What is the scope of a Java variable?

The scope of a Java variable is defined by the *block of code* within which the variable is accessible.

(Briefly, a block of code consists of none, one, or more statements enclosed by a pair of matching curly brackets.)

The scope also determines when the variable is created *(memory set aside to contain the data stored in the variable)* and when it possibly becomes a candidate for destruction *(memory returned to the operating system for recycling and re-use)* .

Scope categories

The scope of a variable places it in one of the four categories shown in Image 5 (p. 467) .

Image 5: Scope categories.

- member variable
- local variable
- method parameter
- exception handler parameter

4.54

Member variable

A member variable is a member of a class *(class variable)* or a member of an object instantiated from that class *(instance variable)* . It must be declared within a class, but not within the body of a method or constructor of the class.

Local variable

A local variable is a variable declared within the body of a method or constructor or within a block of code contained within the body of a method or constructor.

Method parameters

Method parameters are the formal arguments of a method. Method parameters are used to pass values into and out of methods. The scope of a method parameter is the entire method for which it is a parameter.

Exception handler parameters

Exception handler parameters are arguments to exception handlers. Exception handlers will be discussed in a future module.

Illustrating different types of variables in Java

The Java program shown in Listing 8 (p. 468) illustrates

- member variables *(class and instance)* ,
- local variables, and
- method parameters.

An illustration of exception handler parameters will be deferred until exception handlers are discussed in a future module.

Listing 8: The program named member1.

```
/*File member1.java Copyright 1997, R.G.Baldwin
Illustrates class variables, instance
variables, local variables, and method parameters.

Output from this program is:

Class variable is 5
Instance variable is 6
Method parameter is 7
Local variable is 8

**************************************************************/
class member1 { //define the controlling class
  //declare and initialize class variable
  static int classVariable = 5;
  //declare and initialize instance variable
  int instanceVariable = 6;

  public static void main(String[] args){ //main method
    System.out.println("Class variable is "
                                        + classVariable);

    //Instantiate an object of the class to allow for
    // access to instance variable and method.
    member1 obj = new member1();
    System.out.println("Instance variable is "
                                    + obj.instanceVariable);
    obj.myMethod(7); //call the method

    //declare and intitialize a local variable
    int localVariable = 8;
    System.out.println("Local variable is "
                                        + localVariable);

  }//end main

  void myMethod(int methodParameter){
    System.out.println("Method parameter is "
                                       + methodParameter);
  }//end myMethod
}//End member1 class.
```

4.55

Declaration of local variables

In Java, local variables are declared within the body of a method or within a block of code contained within the body of a method.

Scope of local variables

The scope of a local variable extends from the point at which it is declared to the end of the block of code in which it is declared.

What is a "block" of code?

A block of code is defined by enclosing it within curly brackets as in { ... }.

Therefore, the scope of a local variable can be the entire method, or can reduced by declaring it within a block of code within the method.

NOTE: **Special case, scope within a for loop** Java treats the scope of a variable declared within the initialization clause of a **for** statement to be limited to the total extent of the **for** statement.

A future module will explain what is meant by a **for** statement or a **for** loop.

4.22.7 Initialization of variables

Initializing primitive local variables

Local variables of primitive types can be initialized when they are declared using statements such the one shown in Listing 9 (p. 469) .

Listing 9: Initialization of variables.

```
int MyVar, UrVar = 6, HisVar;
```

4.56

Initializing member variables

Member variables can also be initialized when they are declared.

In both cases, the type of the value used to initialize the variable must match the type of the variable.

Initializing method parameters and exception handler parameters

Method parameters and exception handler parameters are initialized by the values passed to the method or exception handler by the calling program.

4.22.8 Run the programs

I encourage you to copy the code from Listing 1 (p. 458) , Listing 6 (p. 464) , and Listing 8 (p. 468) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.22.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0200: Java OOP: Variables
- File: Jb0200.htm
- Originally published: 1997
- Published at cnx.org: 11/18/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.23 Jb0200r: Review[65]

4.23.1 Table of Contents

4.23.2 Preface

This module contains review questions and answers keyed to the module titled Jb0200: Java OOP: Variables [66] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.23.3 Questions

4.23.3.1 Question 1 .

Write a Java application that reads characters from the keyboard until encountering the # character. Echo each character to the screen as it is read. Terminate the program when the user enters the # character.

Answer 1 (p. 483)

4.23.3.2 Question 2

What is the common name for the Java program element that is used to contain data that changes during the execution of the program?

Answer 2 (p. 482)

4.23.3.3 Question 3

What must you do to make a variable available for use in a Java program?

Answer 3 (p. 482)

4.23.3.4 Question 4

True or false? In Java, you are required to initialize the value of all variables when they are declared.

Answer 4 (p. 482)

4.23.3.5 Question 5

Show the proper syntax for declaring two variables and initializing one of them using a single Java statement.

Answer 5 (p. 482)

[65] This content is available online at <http://cnx.org/content/m45173/1.6/>.
[66] http://cnx.org/content/m45150/latest/

4.23.3.6 Question 6

True or false? The Java compiler will accept statements with type mismatches provided that a suitable type conversion can be implemented by the compiler at compile time.
Answer 6 (p. 481)

4.23.3.7 Question 7

Show the proper syntax for the declaration of a variable of type **String[]** in the argument list of the **main** method of a Java program and explain its purpose.
Answer 7 (p. 481)

4.23.3.8 Question 8

Describe the purpose of the type definition in Java.
Answer 8 (p. 481)

4.23.3.9 Question 9

True or false? Variables of type **int** can contain either signed or unsigned values.
Answer 9 (p. 481)

4.23.3.10 Question 10

What is the important characteristic of type definitions in Java that strongly supports the concept of *platform independence* of compiled Java programs?
Answer 10 (p. 481)

4.23.3.11 Question 11

What are the two major categories of types in Java?
Answer 11 (p. 481)

4.23.3.12 Question 12

What is the maximum number of values that can be stored in a variable of a *primitive* type in Java?
Answer 12 (p. 481)

4.23.3.13 Question 13

List the *primitive* types in Java.
Answer 13 (p. 481)

4.23.3.14 Question 14

True or false? Java stores variables of type **char** according to the 8-bit extended ASCII table.
Answer 14 (p. 480)

4.23.3.15 Question 15

True or false? In Java, the name of a *primitive* variable evaluates to the value stored in the variable.
Answer 15 (p. 480)

4.23.3.16 Question 16

True or false? Variables of *primitive* data types in Java are true objects.
Answer 16 (p. 480)

4.23.3.17 Question 17

Why do we care that variables of *primitive* types are not true objects?
Answer 17 (p. 480)

4.23.3.18 Question 18

What is the name of the mechanism commonly used to convert variables of *primitive* types to true objects?
Answer 18 (p. 480)

4.23.3.19 Question 19

How can you tell the difference between a *primitive* type and a *wrapper* for the primitive type when the two are spelled the same?
Answer 19 (p. 480)

4.23.3.20 Question 20

Show the proper syntax for declaring a variable of type **double** and initializing its value to 5.5.
Answer 20 (p. 480)

4.23.3.21 Question 21

Show the proper syntax for declaring a variable of type **Double** and initializing its value to 5.5.
Answer 21 (p. 479)

4.23.3.22 Question 22

Show the proper syntax for extracting the value from a variable of type **Double** .
Answer 22 (p. 479)

4.23.3.23 Question 23

True or false? In Java, the name of a reference variable evaluates to the address of the location in memory where the variable is stored.
Answer 23 (p. 479)

4.23.3.24 Question 24

What is a *legal identifier* in Java?
Answer 24 (p. 479)

4.23.3.25 Question 25

What are the rules for variable names in Java?
Answer 25 (p. 478)

4.23.3.26 Question 26

What is meant by the *scope* of a Java variable?
Answer 26 (p. 478)

4.23.3.27 Question 27

What are the four possible *scope* categories for a Java variable?
Answer 27 (p. 478)

4.23.3.28 Question 28

What is a member variable?
Answer 28 (p. 478)

4.23.3.29 Question 29

Where are *local variables* declared in Java?
Answer 29 (p. 478)

4.23.3.30 Question 30

What is the scope of a local variable in Java?
Answer 30 (p. 478)

4.23.3.31 Question 31

What defines a *block* of code in Java?
Answer 31 (p. 478)

4.23.3.32 Question 32

What is the scope of a variable that is declared within a block of code that is defined within a method and which is a subset of the statements that make up the method?
Answer 32 (p. 477)

4.23.3.33 Question 33

What is the scope of a variable declared within the initialization clause of a *for* statement in Java? Provide an example code fragment.
Answer 33 (p. 477)

4.23.3.34 Question 34

What are *method parameters* and what are they used for?
Answer 34 (p. 477)

4.23.3.35 Question 35

What is the scope of a *method parameter* ?
Answer 35 (p. 477)

4.23.3.36 Question 36

What are *exception handler parameters* ?
Answer 36 (p. 477)

4.23.3.37 Question 37

Write a Java application that illustrates member variables *(class and instance)* , local variables, and method parameters.
Answer 37 (p. 477)

4.23.3.38 Question 38

True or false? Member variables in a Java class can be initialized when the class is defined.
Answer 38 (p. 477)

4.23.3.39 Question 39

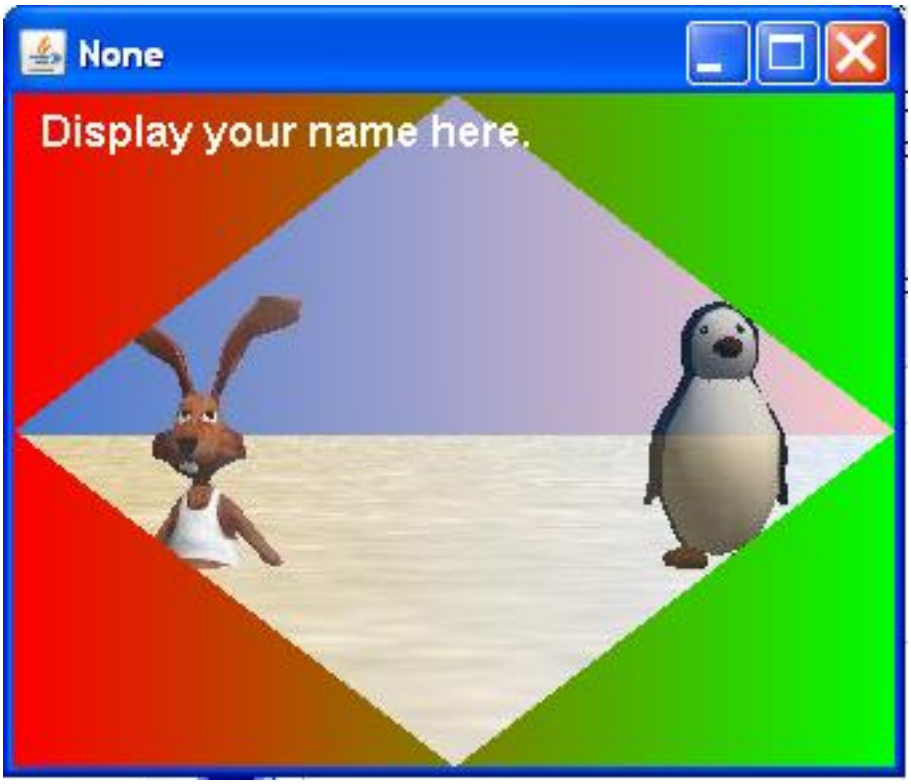
How are *method parameters* initialized in Java?
Answer 39 (p. 477)

4.23.4 Listings

- Listing 1 (p. 479) . Listing for Answer 22.
- Listing 2 (p. 483) . Listing for Answer 1.

What is the meaning of the following two images?
This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.23.5 Answers

4.23.5.1 Answer 39

Method parameters are initialized by the values passed to the method.
Back to Question 39 (p. 475)

4.23.5.2 Answer 38

True.
Back to Question 38 (p. 475)

4.23.5.3 Answer 37

See the application named **member1** in this module [67] for an example of such an application.
Back to Question 37 (p. 475)

4.23.5.4 Answer 36

Exception handler parameters are arguments to exception handlers, which will be discussed in a future module.
Back to Question 36 (p. 475)

4.23.5.5 Answer 35

The scope of a method parameter is the entire method for which it is a parameter.
Back to Question 35 (p. 474)

4.23.5.6 Answer 34

Method parameters are the formal arguments of a method. Method parameters are used to pass values into and out of methods.
Back to Question 34 (p. 474)

4.23.5.7 Answer 33

Java treats the scope of a variable declared within the initialization clause of a *for* statement to be limited to the total extent of the *for* statement. A sample code fragment follows where **cnt** is the variable being discussed:

NOTE:

```
for(int cnt = 0; cnt < max; cnt++){
  //do something
}//end of
```

Back to Question 33 (p. 474)

4.23.5.8 Answer 32

In Java, the scope can be reduced by placing it within a block of code within the method. The *scope* extends from the point at which it is declared to the end of the block of code in which it is declared.
Back to Question 32 (p. 474)

[67] http://cnx.org/content/m45150/latest/#Listing_8

4.23.5.9 Answer 31

A block of code is defined by enclosing it within curly brackets as shown below
{ ... } .
Back to Question 31 (p. 474)

4.23.5.10 Answer 30

The *scope* of a local variable extends from the point at which it is declared to the end of the block of code in which it is declared.
Back to Question 30 (p. 474)

4.23.5.11 Answer 29

In Java, *local variables* are declared within the body of a method or constructor, or within a block of code contained within the body of a method or constructor.
Back to Question 29 (p. 474)

4.23.5.12 Answer 28

A *member variable* is a member of a class (*class* variable) or a member of an object instantiated from that class (*instance* variable). It must be declared within a class, but not within the body of a method or constructor of the class.
Back to Question 28 (p. 474)

4.23.5.13 Answer 27

The *scope* of a variable places it in one of the following four categories:

- member variable
- local variable
- method parameter
- exception handler parameter

Back to Question 27 (p. 474)

4.23.5.14 Answer 26

The *scope* of a Java variable is the block of code within which the variable is accessible.
Back to Question 26 (p. 474)

4.23.5.15 Answer 25

The rules for Java variable names are as follows:

- Must be a legal Java identifier consisting of a series of *Unicode* characters.
- Must not be the same as a Java *keyword* and must not be *true* or *false*.
- Must not be the same as another variable whose declaration appears in the same scope.

Back to Question 25 (p. 473)

4.23.5.16 Answer 24

In Java, a legal identifier is a sequence of Unicode letters and digits of unlimited length. The first character must be a letter. All subsequent characters must be letters or numerals from any alphabet that Unicode supports. In addition, the underscore character (_) and the dollar sign ($) are considered letters and may be used as any character including the first one.

Back to Question 24 (p. 473)

4.23.5.17 Answer 23

False. The name of a reference variable evaluates to either null, or to information that can be used to access an object whose reference has been stored in the variable.

Back to Question 23 (p. 473)

4.23.5.18 Answer 22

Later versions of Java support either syntax shown in Listing 1 (p. 479) .

Listing 1: Listing for Answer 22.

```
class test{
  public static void main(String[] args){
    Double var1 = 5.5;
    double var2 = var1.doubleValue();
    System.out.println(var2);

    double var3 = var1;
    System.out.println(var3);
  }//end main
}//end class test
```

4.57

Back to Question 22 (p. 473)

4.23.5.19 Answer 21

The proper syntax for early versions of Java is shown below. Note the upper-case **D** . Also note the instantiation of a new object of type **Double** .

NOTE:

```
Double myWrappedData = new Double(5.5);
```

Later versions of Java support the following syntax with the new object of type **Double** being instantiated automatically:

NOTE:

```
Double myWrappedData = 5.5;
```

Back to Question 21 (p. 473)

4.23.5.20 Answer 20

The proper syntax is shown below. Note the lower-case **d** .

NOTE:

```
double myPrimitiveData = 5.5;
```

Back to Question 20 (p. 473)

4.23.5.21 Answer 19

The name of the *primitive* type begins with a lower-case letter and the name of the *wrapper* type begins with an upper-case letter such as **double** and **Double** . Note that in some cases, however, that they are not spelled the same. For example, the **Integer** class is the wrapper for type **int** .

Back to Question 19 (p. 473)

4.23.5.22 Answer 18

Wrapper classes

Back to Question 18 (p. 473)

4.23.5.23 Answer 17

This has some ramifications as to how variables can be used *(passing to methods, returning from methods, etc.)* . For example, all variables of *primitive* types are passed by value to methods meaning that the code in the method only has access to a copy of the variable and does not have the ability to modify the variable.

Back to Question 17 (p. 473)

4.23.5.24 Answer 16

False. Primitive data types in Java *(int, double, etc.)* are not true objects.

Back to Question 16 (p. 473)

4.23.5.25 Answer 15

True.

Back to Question 15 (p. 472)

4.23.5.26 Answer 14

False. The **char** type in Java is a 16-bit Unicode character.

Back to Question 14 (p. 472)

4.23.5.27 Answer 13

- byte
- short
- int
- long
- float
- double
- char
- boolean

Back to Question 13 (p. 472)

4.23.5.28 Answer 12

Primitive types contain a single value.
Back to Question 12 (p. 472)

4.23.5.29 Answer 11

Java supports both *primitive* types and *reference* (or object) types.
Back to Question 11 (p. 472)

4.23.5.30 Answer 10

In Java, a variable of a specified type is represented exactly the same way regardless of the platform on which the application or applet is being executed.
Back to Question 10 (p. 472)

4.23.5.31 Answer 9

False. In Java, all variables of type **int** contain signed values.
Back to Question 9 (p. 472)

4.23.5.32 Answer 8

All variables in Java must have a defined *type* . The definition of the *type* determines the set of values that can be stored in the variable and the operations that can be performed on the variable.
Back to Question 8 (p. 472)

4.23.5.33 Answer 7

The syntax is shown in boldface below:

NOTE: public static void main(**String[] args**)

In this case, the type of variable declared is an array of type **String** named **args** *(type String[])* . The purpose of the **String** array variable in the argument list is to make it possible to capture arguments entered on the command line.
Back to Question 7 (p. 472)

4.23.5.34 Answer 6

False. Fortunately, Java provides very strict type checking and generally refuses to compile statements with type mismatches.
Back to Question 6 (p. 472)

4.23.5.35 Answer 5

NOTE:

```
int firstVariable, secondVariable = 10;
```

Back to Question 5 (p. 471)

4.23.5.36 Answer 4

False: In Java, it is possible to initialize the value of a variable when it is declared, but initialization is not required. *(Note however that in some situations, the usage of the variable may require that it be purposely initialized.)* .

Back to Question 4 (p. 471)

4.23.5.37 Answer 3

To use a variable, you must notify the compiler of the name and the type of the variable *(declare the variable).*

Back to Question 3 (p. 471)

4.23.5.38 Answer 2

variable

Back to Question 2 (p. 471)

4.23.5.39 Answer 1

Listing 2: Listing for Answer 1.

```
/*File simple4.java
This application reads characters from the keyboard until
encountering the # character and echoes each character to
the screen. The program terminates when the user enters
the # character.
**********************************************************/
class simple4 { //define the controlling class
  public static void main(String[] args)
                              throws java.io.IOException {
    int ch1 = 0;
    System.out.println(
                      "Enter some text, terminate with #");
    while( (ch1 = System.in.read() ) != '#')
      System.out.print((char)ch1);
    System.out.println("Goodbye");
  }//end main
}//End simple4 class.
```

4.58

Back to Question 1 (p. 471)

4.23.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0200r: Review: Variables
- File: Jb0200r.htm
- Published: 11/23/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.24 Jb0210: Java OOP: Operators[68]

4.24.1 Table of Contents

4.24.2 Preface

Earlier modules have touched briefly on the topic of **operators** . This module discusses Java **operators** in depth.

4.24.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.24.2.1.1 Listings

4.24.3 Introduction

The first step in learning to use a new programming language is usually to learn the foundation concepts such as

- variables,
- operators,
- types,
- expressions,
- flow-of-control, etc.

This module concentrates on the **operators** used in Java.

[68] This content is available online at <http://cnx.org/content/m45195/1.4/>.

4.24.4 Operators

Unary and binary operators

Java provides a set of operators that can be used to perform an action on one, two, or three (p. 485) operands. An operator that operates on one operand is called a *unary* operator. An operator that operates on two operands is called a *binary* operator. An operator that operates on three operands is called a *ternary* operator.

Some operators can behave either as a unary or as a binary operator. The best known such operator s probably the minus sign *(-)* . As a binary operator, the minus sign causes its right operand to be subtracted from its left operand. As a unary operator, the minus sign causes the algebraic sign of the right operand to be changed.

A ternary operator

Java has only one operator that takes three operands. It is a conditional operator, which I sometimes refer to as a cheap **if** statement.

The first operand is a **boolean** expression, which is followed by a question mark character *(?)* . The question mark is followed by a second operand, which is followed by a colon character *(:)* . The colon character is followed by the third operand.

If the **boolean** expression evaluates to true, the value of the operand following the ? is returned. Otherwise, the value of the operand following the : is returned.

An example of the syntax follows:

NOTE: **Ternary operator syntax** boolean expression ? value1 : value2

Overloaded operators

Unlike C++, Java does not support the creation of overloaded operators in program code. *(If you don't know what this means, don't worry about it.)*

Operators from previous programs

The statements in the following note box illustrate the use of the following operators from Java **programs in earlier modules** :

- =
- !=
- +
- (char)

NOTE: **Operators from previous programs**

```
int ch1, ch2 = '0';
while( (ch1 = System.in.read() ) != '#') ch2 = ch1;
System.out.println("The char before the # was "
                                        + (char)ch2);
```

The plus and cast operators

Of particular interest in this list (p. 485) is the plus sign *(+)* and the cast operator *(char)* .

In Java, the plus sign can be used to perform arithmetic addition. It can also be used to concatenate strings. When the plus sign is used in the manner shown above (p. 485) , the operand on the right is automatically converted to a character string before being concatenated with the operand on the left.

The cast operator is used in this case (p. 485) to purposely convert the integer value contained in the **int** variable **ch2** to a character type suitable for concatenating with the string on the left of the plus sign. Otherwise, Java would attempt to convert and display the value of the **int** variable as a series of digits representing the *numeric value* of the character because the character is stored in a variable of type **int** .

The increment operator

An extremely important *unary* operator is the increment operator identified by two plus characters with no space between them *(++)* .

The increment operator causes the value of its operand to be increased by one.

NOTE: **The decrement operator** There is also a decrement operator *(–)* that causes the value of its operand to be decreased by one.

The increment and decrement operators are used in both *prefix* and *postfix* notation.

Prefix and postfix increment and decrement operators

With the *prefix* version, the operand appears to the right of the operator *(++X)* , while with the *postfix* version, the operand appears to the left of the operator *(X++)* .

What's the difference in prefix and postfix?

The difference in prefix and postfix has to do with the point in the sequence of operations that the increment *(or decrement)* actually occurs if the operator and its operand appear as part of a larger overall expression.

(There is effectively no difference if the operator and its operand do not appear as part of a larger overall expression.)

Prefix behavior

With the *prefix* version, the variable is incremented *(or decremented)* before it is used to evaluate the larger overall expression.

Postfix behavior

With the *postfix* version, the variable is used to evaluate the larger overall expression before it is incremented *(or decremented)* .

Illustration of prefix and postfix behavior

The use of both the *prefix* and *postfix* versions of the increment operator is illustrated in the Java program shown in Listing 1 (p. 487) . The output produced by the program is show in the comments at the beginning of the program.

Listing 1: Illustration of prefix and postfix notation.

```
/*File incr01.java Copyright 1997, n
Illustrates the use of the prefix and the postfix increment
operator.

The output from the program follows:

a = 5
b = 5
a + b++ = 10
b = 6

c = 5
d = 5
c + ++d = 11
d = 6

**********************************************************/
class incr01 { //define the controlling class
  public static void main(String[] args){ //main method
    int a = 5, b = 5, c = 5, d = 5;
    System.out.println("a = " + a );
    System.out.println("b = " + b );
    System.out.println("a + b++ = " + (a + b++) );
    System.out.println("b = " + b );
    System.out.println();

    System.out.println("c = " + c );
    System.out.println("d = " + d );
    System.out.println("c + ++d = " + (c + ++d) );
    System.out.println("d = " + d );
  }//end main
}//End incr01 class.
```

4.59

Binary operators and infix notation

Binary operators use *infix* notation, which means that the operator appears between its operands.

General behavior of an operator

As a result of performing the specified action, an operator can be said to return a value *(or evaluate to a value)* of a given type. The type of value returned depends on the operator and the type of the operands.

NOTE: **Evaluating to a value** To evaluate to a value means that after the action is performed, the operator and its operands are effectively replaced in the expression by the value that is returned.

Operator categories

I will divide Java's operators into the following categories for further discussion:

- arithmetic operators
- relational and conditional *(logical)* operators
- bitwise operators
- assignment operators

4.24.4.1 Arithmetic operators

Java supports various arithmetic operators on all floating point and integer numbers.

The binary arithmetic operators

The following table lists the *binary* arithmetic operators supported by Java.

NOTE: **The binary arithmetic operators**

Operator	Description
+	Adds its operands
-	Subtracts the right operand from the left operand
*	Multiplies the operands
/	Divides the left operand by the right operand
%	Remainder of dividing the left operand by the right operand

String concatenation

As mentioned earlier, the plus operator *(+)* is also used to concatenate strings as in the following code fragment:

NOTE: **String concatenation**

```
"MyVariable has a value of  "
                      + MyVariable + " in this program."
```

Coercion

Note that this operation (p. 488) also coerces the value of **MyVariable** to a string representation for use in the expression only. However, the value stored in the variable is not modified in any lasting way.

Unary arithmetic operators

Java supports the following *unary* arithmetic operators.

NOTE: **Unary arithmetic operators**

Operator	Description
+	Indicates a positive value
-	Negates, or changes algebraic sign
++	Adds one to the operand, both prefix and postfix
--	Subtracts one from operand, both prefix and postfix

The result of the increment and decrement operators being either *prefix* or *postfix* was discussed earlier (p. 486) .

4.24.4.2 Relational and conditional (logical) operators

Binary Relational operators

Java supports the set of *binary* relational operators shown in the following table. Relational operators in Java return either *true* or *false* as a **boolean** type.

NOTE: **Binary Relational operators**

```
Operator    Returns true if
   >        Left operand is greater than right operand
   >=       Left operand is greater than or equal to
            right operand
   <        Left operand is less than right operand
   <=       Left operand is less than or equal to
            right operand
   ==       Left operand is equal to right operand
   !=       Left operand is not equal to right operand
```

Conditional expressions

Relational operators are frequently used in the conditional expressions of control statement such as the one in the code fragment shown below.

NOTE: **Conditional expressions**

```
if(LeftVariable <= RightVariable). . .
```

Illustration of relational operators

The program shown in Listing 2 (p. 490) illustrates the result of applying relational operators in Java. The output is shown in the comments at the beginning of the program. Note that the program automatically displays **true** and **false** as a result of applying the relational operators.

Listing 2: Illustration of relational operators.

```
/*File relat01.java Copyright 1997, R.G.Baldwin
Illustrates relational operators.

Output is

The relational 6<5 is false
The relational 6>5 is true

********************************************************/
class relat01 { //define the controlling class
  public static void main(String[] args){ //main method
    System.out.println("The relational 6<5 is "
      +(6<5));
    System.out.println("The relational 6>5 is "
      +(6>5));
  }//end main
}//End relat01 class.
```

4.60

Conditional operators

The relational operators are often combined with another set of operators *(referred to as conditional or logical operators)* to construct more complex expressions.

Java supports three such operators as shown in the following table.

NOTE: **Conditional or logical operators**

Operator	Typical Use	Returns true if
&&	Left && Right	Left and Right are both true
||	Left || Right	Either Left or Right is true
!	! Right	Right is false

The operands shown in the table (p. 490) must be **boolean** types, or must have been created by the evaluation of an expression that returns a **boolean** type.

Left to right evaluation

An important characteristic of the behavior of the logical ***and*** and the logical ***or*** operators is that the expressions are evaluated from left to right, and the evaluation of the expression is terminated as soon as the result of evaluating the expression can be determined.

For example, in the following expression, if the variable **a** is less than the variable **b** , there is no need to evaluate the right operand of the || to determine that the result of evaluating the entire expression would be **true** . Therefore, evaluation will terminate as soon as the answer can be determined.

NOTE: **Left to right evaluation**

```
(a < b) || (c < d)
```

Don't confuse bitwise and with logical and

As discussed in the next section, symbols shown below are the bitwise ***and*** and the bitwise ***or*** .

NOTE: **Bitwise *and* and bitwise *or***

```
& bitwise and
| bitwise or
```

One author states that in Java, the bitwise ***and*** operator can be used as a synonym for the logical ***and*** and the bitwise ***or*** can be used as a synonym for the logical ***inclusive or*** if both of the operands are **boolean** . *(I recommend that you don't do that because it could cause confusion for someone reading your code.)*

Note however that according to a different author, in this case, the evaluation of the expression is not terminated until all operands have been evaluated, thus eliminating the possible advantage of the left-to-right evaluation.

4.24.4.3 Bitwise operators

Java provides a set of operators that perform actions on their operands one bit at a time as shown in the following table.

NOTE: **Bitwise operators**

Operator	Typical Use	Operation
≫	OpLeft ≫ Dist	Shift bits of OpLeft right by Dist bits (signed)
≪	OpLeft ≪ Dist	Shift bits of OpLeft left by Dist bits
≫>	OpLeft ≫> Dist	Shift bits of OpLeft right by Dist bits (unsigned)
&	OpLeft & OpRight	Bitwise and of the two operands
|	OpLeft | OpRight	Bitwise

Populating vacated bits for shift operations

The *signed* right shift operation populates the vacated bits with the sign bit, while the left shift and the *unsigned* right shift populate the vacated bits with zeros.

In all cases, bits shifted off the end are lost.

The rule for bitwise *and*

The bitwise **and** operation operates according to the rule that the bitwise **and** of two 1 bits is a 1 bit.

Any other combination results in a 0 bit.

Bitwise inclusive *or*

For the *inclusive or* , if either bit is a 1, the result is a 1.

Otherwise, the result is a 0.

Bitwise exclusive *or*

For the *exclusive or* , if either but not both bits is a 1, the result is a 1.

Otherwise, the result is a 0.

Another way to state this is if the bits are different, the result is a 1. If the two bits are the same, the result is a 0.

The complement operator

Finally, the complement operator changes each 1 to a 0 and changes each 0 to a 1.

4.24.4.4 Assignment operators

Simple assignment operator

The *(=)* is a value assigning *binary* operator in Java. The value stored in memory and represented by the right operand is copied into the memory represented by the left operand.

Using the assignment operator with reference variables

You need to be careful and think about what you are doing when you use the assignment operator with reference variables in Java. If you assign one reference variable to another, you simply end up with two reference variables that refer to the same object. You do not end up with two different objects.

(If what you need is another copy of the object, you may be able to use the ***clone*** *method to accomplish that.)*

Shortcut assignment operators

Java supports the following list of *shortcut* assignment operators. These operators allow you to perform an assignment and another operation with a single operator.

NOTE: **Shortcut assignment operators**

```
+=
-=
*=
/=
%=
&=
|=
^=
≪=
≫=
≫>=
```

For example, the two statements that follow perform the same operation.

NOTE: **Illustration of shortcut assignment operation**

```
x += y;
x = x + y;
```

4.24.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0210: Java OOP: Operators
- File: Jb0210
- Originally published: 1997

- Published at cnx.org: 11/23/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.25 Jb0210r Review[69]

4.25.1 Table of Contents

4.25.2 Preface

This module contains review questions and answers keyed to the module titled Jb0210: Java OOP: Operators [70] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.25.3 Questions

4.25.3.1 Question 1 .

An operator performs an action on what? Provide the name.
Answer 1 (p. 510)

4.25.3.2 Question 2

What do we call an operator that operates on only one operand?
Answer 2 (p. 510)

4.25.3.3 Question 3

What do we call an operator that operates on two operands?
Answer 3 (p. 510)

4.25.3.4 Question 4

Is the minus sign a *unary* or a *binary* operator, or both? Explain your answer.
Answer 4 (p. 510)

4.25.3.5 Question 5

Describe operator overloading.
Answer 5 (p. 510)

[69] This content is available online at <http://cnx.org/content/m45186/1.4/>.
[70] http://cnx.org/content/m45195

4.25.3.6 Question 6

True or false? Java programmers may overload operators.
Answer 6 (p. 510)

4.25.3.7 Question 7

Show the symbols used for the following operators in Java: *assignment* , *not equal* , *addition* , *cast* .
Answer 7 (p. 510)

4.25.3.8 Question 8

Are any operators automatically overloaded in Java? If so, identify one and describe its overloaded behavior.
Answer 8 (p. 510)

4.25.3.9 Question 9

What is the purpose of the cast operator?
Answer 9 (p. 509)

4.25.3.10 Question 10

True or false? The increment operator is a *binary* operator.
Answer 10 (p. 509)

4.25.3.11 Question 11

Show the symbol for the increment operator.
Answer 11 (p. 509)

4.25.3.12 Question 12

Describe the appearance and the behavior of the increment operator with both *prefix* and *postfix* notation. Show example, possibly incomplete, code fragments illustrating both notational forms.
Answer 12 (p. 509)

4.25.3.13 Question 13

Show the output that would be produced by the Java application in Listing 1 (p. 496) .

Listing 1: Listing for Question 13.

```
class incr01 { //define the controlling class
  public static void main(String[] args){ //define main
    int x = 5, X = 5, y = 5, Y = 5;
    System.out.println("x = " + x );
    System.out.println("X = " + X );
    System.out.println("x + X++ = " + (x + X++) );
    System.out.println("X = " + X );
    System.out.println();
    System.out.println("y = " + y );
    System.out.println("Y = " + Y );
    System.out.println("y + ++Y = " + (y + ++Y) );
    System.out.println("Y = " + Y );
  }//end main
}//End incr01 class.  Note no semicolon required
//End Java application
```

4.61

Answer 13 (p. 509)

4.25.3.14 Question 14

True or false? *Binary* operators use *outfix* notation. If your answer is False, explain why.
Answer 14 (p. 509)

4.25.3.15 Question 15

In practice, what does it mean to say that an operator that has performed an action returns a value *(or evaluates to a value)* of a given type?
Answer 15 (p. 509)

4.25.3.16 Question 16

Show and describe at least five of the *binary arithmetic* operators supported by Java (Clarification: *binary* operators does not mean *bitwise* operators).
Answer 16 (p. 508)

4.25.3.17 Question 17

In addition to arithmetic addition, what is another use for the plus operator *(+)* ? Show an example code fragment to illustrate your answer. The code fragment need not be a complete statement.
Answer 17 (p. 508)

4.25.3.18 Question 18

When the plus operator *(+)* is used as a concatenation operator, what is the nature of its behavior if its left operand is of type **String** and its right operand is not of type **String** ? If the right operand is a variable that is not of type **String** , what is the impact of this behavior on that variable.

Answer 18 (p. 508)

4.25.3.19 Question 19

Show and describe four *unary* arithmetic operators supported by Java.

Answer 19 (p. 508)

4.25.3.20 Question 20

What is the type returned by *relational* operators in Java?

Answer 20 (p. 507)

4.25.3.21 Question 21

Show and describe six different *relational* operators supported by Java.

Answer 21 (p. 507)

4.25.3.22 Question 22

Show the output that would be produced by the Java application shown in Listing 2 (p. 497) .

Listing 2: Listing for Question 22.

```
class relat01 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("The relational 6<5 is " + (6<5 ));
    System.out.println("The relational 6>5 is " + (6>5 ));
  }//end main
}//End relat01 class.  Note no semicolon required
//End Java application
```

Answer 22 (p. 507)

4.25.3.23 Question 23

Show and describe three operators *(frequently referred to as conditional or logical operators)* that are often combined with relational operators to construct more complex expressions *(often called conditional expressions)* . Hint: The || operator returns true if either the left operand, the right operand, or both operands are true. What are the other two and how do they behave?

Answer 23 (p. 507)

4.25.3.24 Question 24

Describe the special behavior of the || operator in the following expression for the case where the value of the variable **a** is less than the value of the variable **b** .

NOTE:

```
(a < b) || (c < d)
```

Answer 24 (p. 506)

4.25.3.25 Question 25

Show the symbols used for the bitwise *and* operator and the bitwise *inclusive or* operator.

Answer 25 (p. 506)

4.25.3.26 Question 26

Show and describe seven operators in Java that perform actions on the operands one bit at a time *(bitwise operators)* .

Answer 26 (p. 506)

4.25.3.27 Question 27

True or false? In Java, the *signed* right shift operation populates the vacated bits with the zeros, while the left shift and the *unsigned* right shift populate the vacated bits with the sign bit. If your answer is False, explain why.

Answer 27 (p. 506)

4.25.3.28 Question 28

True or false? In a *signed* right-shift operation in Java, the bits shifted off the right end are lost. If your answer is False, explain why.

Answer 28 (p. 506)

4.25.3.29 Question 29

Using the symbols 1 and 0, construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise and* operation on these four combinations of 1 and 0.

Answer 29 (p. 505)

4.25.3.30 Question 30

Using the symbols 1 and 0 construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise inclusive or* operation on these four combinations of 1 and 0.

Answer 30 (p. 505)

4.25.3.31 Question 31

Using the symbols 1 and 0 construct a truth table showing the four possible combinations of 1 and 0. Using a 1 or a 0, show the result of the *bitwise exclusive or* operation on these four combinations of 1 and 0.

Answer 31 (p. 505)

4.25.3.32 Question 32

True or false? For the *exclusive or* , if the two bits are different, the result is a 1. If the two bits are the same, the result is a 0. If your answer is False, explain why.

Answer 32 (p. 505)

4.25.3.33 Question 33

Is the *assignment* operator a *unary* operator or a *binary* operator. Select one or the other.

Answer 33 (p. 505)

4.25.3.34 Question 34

True or false? In Java, when using the assignment operator, the value stored in memory and represented by the right operand is copied into the memory represented by the left operand. If your answer is False, explain why.

Answer 34 (p. 505)

4.25.3.35 Question 35

Show two of the *shortcut* assignment operators and explain how they behave by comparing them with the regular *(non-shortcut)* versions. Hint: the *(^=)* operator is a *shortcut* assignment operator.

Answer 35 (p. 504)

4.25.3.36 Question 36

Write a Java application that clearly illustrates the difference between the prefix and the postfix versions of the increment operator. Provide a termination message that displays your name.

Answer 36 (p. 504)

4.25.3.37 Question 37

Write a Java application that illustrates the use of the following relational operators:

NOTE:

```
<
>
<=
>=
==
!=
```

Provide appropriate text in the output. Also provide a termination message with your name.

Answer 37 (p. 503)

4.25.3.38 Question 38

write a Java application that illustrates the use of the following logical or conditional operators:

NOTE: **Logical or conditional operators**

```
&&
||
!
```

Provide appropriate text in the output. Also provide a termination message with your name.
Answer 38 (p. 502)

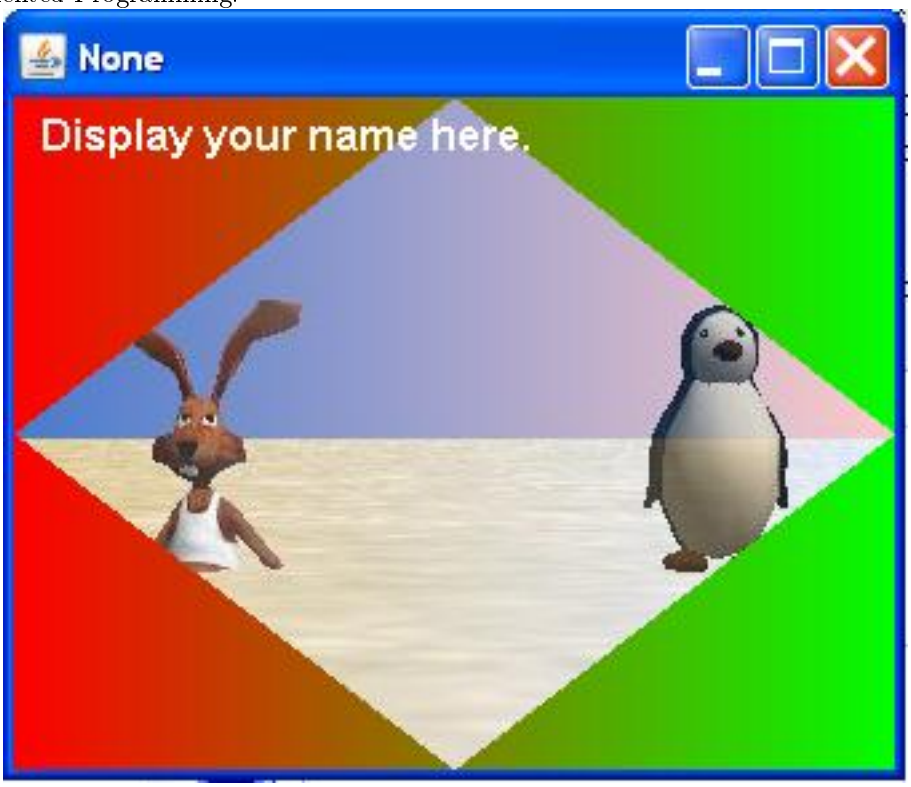
4.25.4 Listings

- Listing 1 (p. 496) . Listing for Question 13.
- Listing 2 (p. 497) . Listing for Question 22.
- Listing 3 (p. 502) . Listing for Answer 38.
- Listing 4 (p. 503) . Listing for Answer 37.
- Listing 5 (p. 504) . Listing for Answer 36.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.


Prob05a.jpg
Put your name here

4.25.5 Answers

4.25.5.1 Answer 38

Listing 3: Listing for Answer 38.

```
/*File SampProg09.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the use of the following
logical or conditional operators:

&&  ||  !

Provide appropriate text in the output.  Also provide
a termination message with your name.
**********************************************************/
class SampProg09 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("true and true is "
      + (true && true) );
    System.out.println("true and false is "
      + (true && false) );

    System.out.println("true or true is "
      + (true || true) );
    System.out.println("true or false is "
      + (true || false) );
    System.out.println("false or false is "
      + (false || false) );

    System.out.println("not true is " + (! true) );
    System.out.println("not false is " + (! false) );

    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg09 class.  Note no semicolon required
```

4.63

Back to Question 38 (p. 499)

4.25.5.2 Answer 37

Listing 4: Listing for Answer 37.

```
/*File SampProg08.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the use of the following
relational operators:

< > <= >= == !=

Provide appropriate text in the output.  Also provide
a termination message with your name.
**********************************************************/
class SampProg08 { //define the controlling class
  public static void main(String[] args){ //define main
    System.out.println("The relational 6<5 is "
                                          + (6<5 ) );
    System.out.println("The relational 6>5 is "
                                           + (6>5 ) );
    System.out.println("The relational 5>=5 is "
                                         + (5>=5 ) );
    System.out.println("The relational 5<=5 is "
                                         + (5<=5 ) );
    System.out.println("The relational 6==5 is "
                                            + (6==5 ) );
    System.out.println("The relational 6!=5 is "
                                            + (6!=5 ) );
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg08 class.  Note no semicolon required
```


Back to Question 37 (p. 499)

4.25.5.3 Answer 36

Listing 5: Listing for Answer 36.

```
/*File SampProg07.java from module 22
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that clearly illustrates the difference between
the prefix and the postfix versions of the increment
operator.

Provide a termination message that displays your name.
**********************************************************/
class SampProg07{
  static public void main(String[] args){
    int x = 3;
    int y = 3;
    int z = 10;
    System.out.println("Prefix version gives  "
                                          + (z + ++x));
    System.out.println("Postfix version gives "
                                          + (z + y++));
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//end class SampProg07
```

4.65

Back to Question 36 (p. 499)

4.25.5.4 Answer 35

Java supports the following list of *shortcut* assignment operators. These operators allow you to perform an assignment and another operation with a single operator.

NOTE:

```
+=
-=
*=
/=
%=
&=
|=
^=
≪=
```

≫=
≫>=

For example, the two statements that follow perform the same operation.

- **x += y;**
- **x = x + y;**

The behavior of each of the *shortcut* assignment operators follows this same pattern.
Back to Question 35 (p. 499)

4.25.5.5 Answer 34

True.
Back to Question 34 (p. 499)

4.25.5.6 Answer 33

The assignment operator is a *binary* operator.
Back to Question 33 (p. 499)

4.25.5.7 Answer 32

True.
Back to Question 32 (p. 499)

4.25.5.8 Answer 31

The answer for the bitwise *exclusive or* is:

- 11 1 xor 1 produces 0
- 10 1 xor 0 produces 1
- 01 0 xor 1 produces 1
- 00 0 xor 0 produces 0

Back to Question 31 (p. 498)

4.25.5.9 Answer 30

The answer for the bitwise *inclusive or* is:

- 11 1 or 1 produces 1
- 10 1 or 0 produces 1
- 01 0 or 1 produces 1
- 00 0 or 0 produces 0

Back to Question 30 (p. 498)

4.25.5.10 Answer 29

The answer for the bitwise ***and*** is:

- 11 1 and 1 produces 1
- 10 1 and 0 produces 0
- 01 0 and 1 produces 0
- 00 0 and 0 produces 0

Back to Question 29 (p. 498)

4.25.5.11 Answer 28

True: Bits shifted off the right end are lost.
Back to Question 28 (p. 498)

4.25.5.12 Answer 27

False: In Java, the *signed* right shift operation populates the vacated bits with the sign bit, while the left shift and the *unsigned* right shift populate the vacated bits with zeros.
Back to Question 27 (p. 498)

4.25.5.13 Answer 26

The following table shows the seven bitwise operators supported by Java.

NOTE: **Bitwise operators**

Operator	Typical Use	Operation
≫	OpLeft ≫ Dist	Shift bits of OpLeft right by Dist bits (signed)
><	OpLeft ≪ Dist	Shift bits of OpLeft left by Dist bits
≫>	OpLeft ≫> Dist	Shift bits of OpLeft right by Dist bits (unsigned)
&	OpLeft & OpRight	Bitwise and of the two operands
|	OpLeft | OpRight	Bitwise

Back to Question 26 (p. 498)

4.25.5.14 Answer 25

The bitwise ***and*** operator and the bitwise ***inclusive or*** operator are shown below.

NOTE: **Two bitwise operators**

```
& bitwise and
| bitwise inclusive or
```

Back to Question 25 (p. 498)

4.25.5.15 Answer 24

An important characteristic of the behavior of the ***logical and*** operator and the ***logical or*** operator in Java is that the expressions containing them are evaluated from *left to right*. The evaluation of the expression is. terminated as soon as the result of evaluating the expression can be determined. For example, in the expression given in Question 24 (p. 498) , if the variable **a** is less than the variable **b** , there is no need to evaluate the right operand of the || operator to determine the value of the entire expression. Therefore, evaluation will terminate as soon as it is determined that **a** is less than **b** .
Back to Question 24 (p. 498)

4.25.5.16 Answer 23

The following three *logical* or *conditional* operators are supported by Java.

NOTE: **The logical or conditional operators**

```
Operator  Typical Use       Returns true if
&&        Left && Right     Left and Right are both true
||        Left || Right     Either Left or Right is true
!         ! Right           Right is false
```

Back to Question 23 (p. 497)

4.25.5.17 Answer 22

This program produces the following output:

NOTE:

```
The relational 6<5 is false
The relational 6>5 is true
```

Back to Question 22 (p. 497)

4.25.5.18 Answer 21

Java supports the following set of *relational* operators:

NOTE: **Relational operators**

```
Operator    Returns true if
   >        Left operand is greater than right operand
   >=       Left operand is greater than or equal
            to right operand
   <        Left operand is less than right operand
   <=       Left operand is less than or equal
            to right operand
   ==       Left operand is equal to right operand
   !=       Left operand is not equal to right operand
```

Back to Question 21 (p. 497)

4.25.5.19 Answer 20

Relational operators return the **boolean** type in Java.
Back to Question 20 (p. 497)

4.25.5.20 Answer 19

Java supports the following four *unary* arithmetic operators.

NOTE: **Unary arithmetic operators**

```
Operator    Description
   +        Indicates a positive value
   -        Negates, or changes algebraic sign
   ++       Adds one to the operand,
            both prefix and postfix
   --       Subtracts one from operand,
            both prefix and postfix
```

Back to Question 19 (p. 497)

4.25.5.21 Answer 18

The operator coerces the value of the right operand to a string representation for use in the expression only. If the right operand is a variable, the value stored in the variable is not modified in any way.

Back to Question 18 (p. 497)

4.25.5.22 Answer 17

The plus operator *(+)* is also used to concatenate strings as in the following code fragment:

NOTE: **String concatenation**

```
"MyVariable has a value of  "
     + MyVariable + " in this program."
```

Back to Question 17 (p. 496)

4.25.5.23 Answer 16

Java support various arithmetic operators on floating point and integer numbers. The following table lists five of the *binary* arithmetic operators supported by Java.

NOTE: **Binary arithmetic operators**

```
Operator    Description

   +        Adds its operands
   -        Subtracts the right operand from the left
            operand
   *        Multiplies the operands
   /        Divides the left operand by the right
            operand
   %        Remainder of dividing the left operand by
            the right operand
```

Back to Question 16 (p. 496)

4.25.5.24 Answer 15

As a result of performing the specified action, an operator can be said to return a value *(or evaluate to a value)* of a given type. The type depends on the operator and the type of the operands. To *evaluate to a value* means that after the action is performed, the operator and its operands are effectively replaced in the expression by the value that is returned.

Back to Question 15 (p. 496)

4.25.5.25 Answer 14

False: *Binary* operators use *infix* notation, which means that the operator appears between its operands.

Back to Question 14 (p. 496)

4.25.5.26 Answer 13

The output from this Java application follows:

- x = 5
- X = 5
- x + X++ = 10
- X = 6
- y = 5
- Y = 5
- y + ++Y = 11
- Y = 6

Back to Question 13 (p. 495)

4.25.5.27 Answer 12

The increment operator may be used with both *prefix* and *postfix* notation. Basically, the increment operator causes the value of the variable to which it is applied to be increased by one.

With *prefix* notation, the operand appears to the right of the operator *(++X)* , while with *postfix* notation, the operand appears to the left of the operator *(X++)* .

The difference in behavior has to do with the point in the sequence of operations that the increment actually occurs.

With the *prefix* version, the variable is incremented before it is used to evaluate the larger overall expression in which it appears. With the *postfix* version, the variable is used to evaluate the larger overall expression and then the variable is incremented.

Back to Question 12 (p. 495)

4.25.5.28 Answer 11

The symbol for the increment operator is two plus signs with no space between them *(++)*.

Back to Question 11 (p. 495)

4.25.5.29 Answer 10

False: The increment operator is a *unary* operator.

Back to Question 10 (p. 495)

4.25.5.30 Answer 9

The cast operator is used to purposely convert from one type to another.

Back to Question 9 (p. 495)

4.25.5.31 Answer 8

The plus sign *(+)* is automatically overloaded in Java. The plus sign can be used to perform arithmetic addition. It can also be used to concatenate strings. However, the plus sign does more than concatenate strings. It also performs a conversion to **String** type. When the plus sign is used to concatenate strings and one operand is a string, the other operand is automatically converted to a character string before being concatenated with the existing string.

Back to Question 8 (p. 495)

4.25.5.32 Answer 7

The operators listed in order are:

- =
- !=
- +
- (char)

where the cast operator is being used to cast to the type **char** .

Back to Question 7 (p. 495)

4.25.5.33 Answer 6

Java does not support operator overloading by programmers.

Back to Question 6 (p. 495)

4.25.5.34 Answer 5

For those languages that support it *(such as C++)* operator overloading means that the programmer can redefine the behavior of an operator with respect to objects of a new type defined by that program.

Back to Question 5 (p. 494)

4.25.5.35 Answer 4

Both. As a *binary* operator, the minus sign causes its right operand to be subtracted from its left operand. As a *unary* operator, the minus sign causes the algebraic sign of the right operand to be changed.

Back to Question 4 (p. 494)

4.25.5.36 Answer 3

An operator that operates on two operands is called a *binary* operator.

Back to Question 3 (p. 494)

4.25.5.37 Answer 2

An operator that operates on only one operand is called a *unary* operator.

Back to Question 2 (p. 494)

4.25.5.38 Answer 1

An operator performs an action on one or two operands.

Back to Question 1 (p. 494)

4.25.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0210r Review: Operators
- File: Jb0210r.htm
- Originally published: 1997
- Published at cnx.org: 11/23/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.26 Jb0220: Java OOP: Statements and Expressions[71]

4.26.1 Table of Contents

4.26.2 Preface

Java programs are composed of statements, and statements are constructed from expressions. This module takes a very brief look at Java statements and expressions.border width

4.26.3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, etc. This module concentrates on expressions and statements.

[71] This content is available online at <http://cnx.org/content/m45192/1.3/>.

4.26.4 Expressions

The hierarchy

Java programs are composed of statements, and statements are constructed from expressions.

An expression is a specific combination of operators and operands, that evaluates to a single value. The operands can be variables, constants, or method calls.

A method call evaluates to the value returned by the method.

Named constants

Java supports named constants that are implemented through the use of the **final** keyword.

The syntax for creating a named constant in Java is as follows:

NOTE: **Named constants**

```
final float PI = 3.14159;
```

While this is not a constant type, it does produce a value that can be referenced in the program and which cannot be modified.

The **final** keyword prevents the value of **PI** from being modified in this case (p. 512) . You will learn later that there are some other uses for the **final** keyword in Java as well.

Operator precedence

In some cases, the order in which the operations are performed determines the result. You can control the order of evaluation by the use of matching parentheses.

If you don't provide such parentheses, the order will be determined by the precedence of the operators *(you should find and review a table of Java operator precedence)* with the operations having higher precedence being evaluated first.

4.26.5 Statements

According to The Java Tutorials [72] , *"A statement forms a complete unit of execution."*

A statement is constructed by combining one or more expressions into a compound expression and terminating that expression with a semicolon.

4.26.6 Further reading

As of November 2012, a good tutorial on this topic is available on the Oracle website titled Expressions, Statements, and Blocks [73] .

4.26.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0220: Java OOP: Statements and Expressions
- File: Jb0220.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12
- Revised: 01/02/13

[72] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html
[73] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.27 Jb0220r Review[74]

4.27.1 Table of Contents

4.27.2 Preface

This module contains review questions and answers keyed to the module titled Jb0220: Java OOP: Statements and Expressions [75] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.27.3 Questions

4.27.3.1 Question 1 .

A Java program is composed of a series of what?

Answer 1 (p. 517)

4.27.3.2 Question 2

Statements in Java are constructed from what?

Answer 2 (p. 517)

4.27.3.3 Question 3

Describe an expression in Java.

Answer 3 (p. 517)

4.27.3.4 Question 4

What does a method call evaluate to in Java?

Answer 4 (p. 517)

4.27.3.5 Question 5

True or false? Java supports named constants. If false, explain why.

Answer 5 (p. 517)

4.27.3.6 Question 6

Provide a code fragment that illustrates the syntax for creating a named constant in Java.

Answer 6 (p. 516)

[74] This content is available online at <http://cnx.org/content/m45189/1.4/>.
[75] http://cnx.org/content/m45192

4.27.3.7 Question 7

True or false? Java supports a **constant** type. If false, explain why.

Answer 7 (p. 516)

4.27.3.8 Question 8

What is the common method of controlling the order of evaluation of expressions in Java?

Answer 8 (p. 516)

4.27.3.9 Question 9

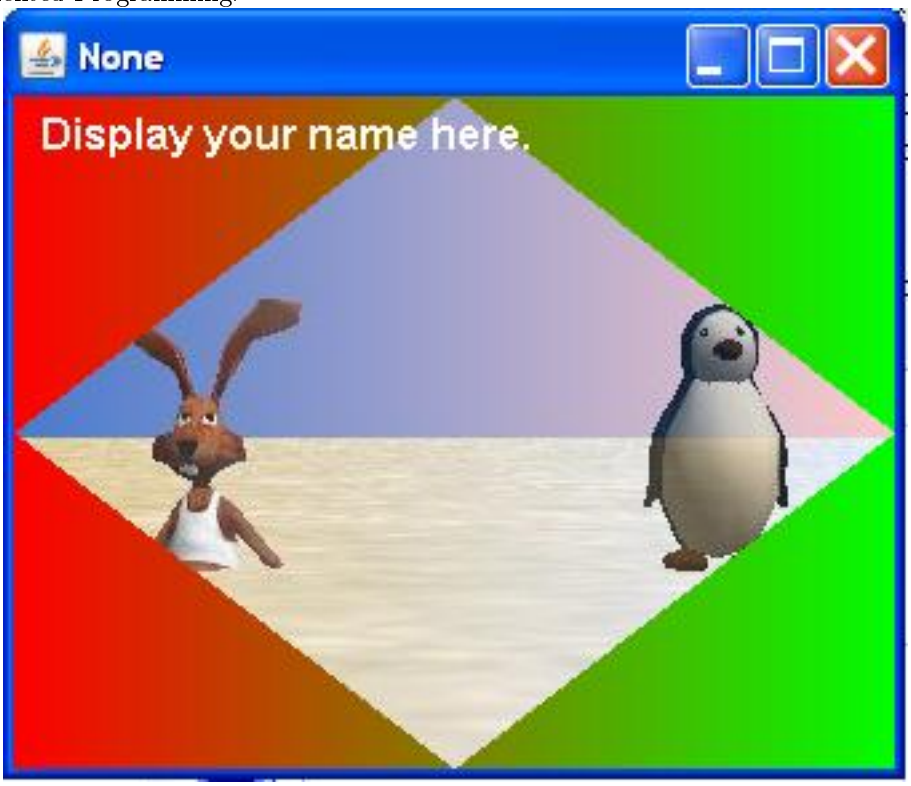
If you don't use matching parentheses to control the order of evaluation of expressions, what is it that controls the order of evaluation?

Answer 9 (p. 516)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.27.4 Answers

4.27.4.1 Answer 9

If you don't provide matching parentheses to control the order of evaluation, the order will be determined by the precedence of the operators with the operations having higher precedence being evaluated first. For example, multiply and divide have higher precedence than add and subtract.

Back to Question 9 (p. 515)

4.27.4.2 Answer 8

You can control the order of evaluation by the use of matching parentheses.

Back to Question 8 (p. 515)

4.27.4.3 Answer 7

False. Java does not support a constant type. However, in Java, it is possible to achieve the same result by declaring and initializing a variable and making it **final** .

Back to Question 7 (p. 515)

4.27.4.4 Answer 6

The syntax for creating a named constant in Java is shown below.

NOTE: **A named constant in Java**

```
final float PI = 3.14159;
```

Back to Question 6 (p. 514)

4.27.4.5 Answer 5

True. Java supports named constants that are constructed using variable declarations with the **final** keyword.

Back to Question 5 (p. 514)

4.27.4.6 Answer 4

A method call evaluates to the value returned by the method.

Back to Question 4 (p. 514)

4.27.4.7 Answer 3

An expression is a specific combination of operators and operands that evaluates to a particular value. The operands can be variables, constants, or method calls.

Back to Question 3 (p. 514)

4.27.4.8 Answer 2

Statements in Java re constructed from expressions.

Back to Question 2 (p. 514)

4.27.4.9 Answer 1

A Java program is composed of a series of statements.

Back to Question 1 (p. 514)

4.27.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0220r Review: Statements and Expressions
- File: Jb0220r.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12
- Revised: 01/02/13

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.28 Jb0230: Java OOP: Flow of Control[76]

4.28.1 Table of Contents

4.28.2 Preface

Java supports several different statements designed to alter or control the logical flow of the program. This module explores those statements.

4.28.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

4.28.2.1.1 Images

[76] This content is available online at <http://cnx.org/content/m45196/1.4/>.

4.28.2.1.2 Listings

4.28.3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, etc. This module concentrates on *flow-of-control* .

4.28.4 Flow of control

What is flow of control?

Java supports several different kinds of statements designed to alter or control the logical flow of the program.

The ability to alter the logical flow of the program is often referred to as *Flow of Control* .

Statements that support flow of control

Image 1 (p. 520) lists the statements supported by Java for controlling the logical flow of the program.

Image 1: Statements that support flow of control.

Statement	Type
if-else	selection
switch-case	selection
for	loop
for-each	loop
while	loop
do-while	loop
try-catch-finally	exception handling
throw	exception handling
break	miscellaneous
continue	miscellaneous
label:	miscellaneous
return	miscellaneous
goto	reserved by Java but not supported

4.66

4.28.4.1 The while statement

We've seen the **while** statement in earlier modules. Several of the programs in earlier modules contained a **while** statement designed to control the logical flow of the program.

Syntax of a while statement

The general syntax of a **while** statement is shown in Image 2 (p. 521) .

Image 2: Syntax of a while statement.

```
while (conditional expression)
  statement or compound statement;
```

4.67

Behavior of a while statement

The **three pillars** of procedural programming are

- sequence
- selection
- loop

The **while** statement is commonly used to create a loop structure, often referred to as a *while loop* .

Once the **while** statement is encountered in the sequence of code, the program will continue to execute the statement or compound statement shown in Image 2 (p. 521) for as long as the conditional expression evaluates to true. *(Note that a compound statement is created by enclosing two or more statements inside a pair of matching curly brackets, thus creating a block of code as the body of the* **while** *statement or loop.)*

Sample Java *while* statement

The **while** statement shown in Listing 1 (p. 521) was extracted from a Java program in an earlier module.

Listing 1: Sample Java while statement.

```
while( (ch1 = System.in.read() ) != '#')
  ch2 = ch1;
```

4.68

The *in* variable of the *System* class

The **System** class defines a *class* variable named **in** . Because it is a *class* variable, it can be accessed using the name of the **System** class without the requirement to instantiate an object of the **System** class.

What the *in* variable contains

The **in** variable refers to an instance of a class that provides a **read** method that returns a character from the standard input device *(typically the keyboard)* .

Therefore, the expression **System.in.read()** in Listing 1 (p. 521) constitutes a call to the **read** method of the object referred to by the **in** variable of the **System** class.

A *while* loop is an entry condition loop

The **while** statement is used to form an *entry condition* loop. The significance of an entry condition loop is that the conditional expression is tested before the statements in the loop are executed. If it tests false initially, the statements in the loop are never executed.

The **while** loop shown in Listing 1 (p. 521) will continue reading characters from the keyboard for as long as the character entered is not the # character. *(Recall the not equal (!=) operator from an earlier module.)*

4.28.4.2 The if-else statement

The general syntax of an **if-else** statement is shown in Image 3 (p. 522) .

Image 3: Syntax of an if-else statement.

```
if(conditional expression)
  statement or compound statement;
else //optional
  statement or compound statement; //optional
```

4.69

The **if-else** statement is the most basic of the statements used to control the logical flow of a Java program. It is used to satisfy the *selection* pillar mentioned earlier (p. 521) .

This statement will execute a specified block of code if some particular condition is true, and optionally, will execute a different block of code if the condition is not true.

The **else** clause shown in Image 3 (p. 522) is optional. If it is not provided and the condition is not true, control simply passes to the next statement following the **If** statement with none of the code in the body of the **if** statement being executed. If the condition is true, the code in the body of the **if** statement is executed.

If the **else** clause is provided and the condition is true, the code in the body of the **if** clause is executed and the code in the body of the **else** clause is ignored.

If the **else** clause is provided and the condition is false, the code in the body of the **if** clause is ignored and the code in the body of the **else** clause is executed.

In all cases, control passes to the next statement following the **if-else** statement when the code in the **if-else** statement has finished executing. In other words, this is not a loop structure.

4.28.4.3 The switch-case statement

The **switch-case** statement is another implementation of the *selection* pillar mentioned earlier (p. 521) . The general syntax of a **switch-case** statement is shown in Image 4 (p. 523) .

Image 4: Syntax of a switch-case statement.

```
switch(expression){
  case constant:
    //sequence of optional statements
    break; //optional
  case constant:
    //sequence of optional statements
    break; //optional
.
.
.
  default //optional
    //sequence of optional statements
}
```

4.70

The type of the *expression*

According to the book, *Java Language Reference*, by Mark Grand, the expression shown in the first line in Image 4 (p. 523) must be of type **byte**, **char**, **short**, or **int**.

The behavior of the switch-case statement

The expression is tested against a series of *case* constants of the same type as the expression. If a match is found, the sequence of optional statements associated with that *case* is executed.

Execution of statements continues until the optional **break** is encountered. When **break** is encountered, execution of the switch statement is terminated and control passes to the next statement following the switch statement.

If there is no **break** statement, all of the statements following the matching case will be executed including those in cases further down the page.

The optional default keyword

If no match is found and the optional default keyword along with a sequence of optional statements has been provided, those statements will be executed.

Labeled break

Java also supports labeled break statements. This capability can be used to cause Java to exhibit different behavior when switch statements are nested. This will be explained more fully in a later section on labeled break statements.

4.28.4.4 The for loop

The **for** statement is another implementation of the *loop* pillar mentioned earlier (p. 521).

Actions of a *for loop*

The operation of a loop normally involves three actions in addition to executing the code in the body of the loop:

- Initialize a control variable.
- Test the control variable in a conditional expression.
- Update the control variable.

Grouping the actions

Java provides the **for** loop construct that groups these three actions in one place.

The syntax of a for loop

A **for** loop consists of three clauses separated by semicolons as shown in Image 5 (p. 524) .

Image 5: Syntax of a for loop.

```
for (first clause; second clause; third clause)
  single or compound statement
```

4.71

Contents of the clauses

The first and third clauses can contain one or more expressions, separated by the *comma operator* .

The *comma operator*

The comma operator guarantees that its left operand will be executed before its right operand.

(While the comma operator has other uses in C++, this is the only use of the comma operator in Java.)

Behavior and purpose of the first clause

The expressions in the first clause are executed only once, at the beginning of the loop. Any legal expression(s) may be contained in the first clause, but typically the first clause is used for initialization.

Declaring and initializing variables in the first clause

Variables can be declared and initialized in the first clause, and this has an interesting ramification regarding scope that will be discussed later.

Behavior of the second clause

The second clause consists of a single expression that must evaluate to a **boolean** type with a value of true or false. The expression in the second clause must eventually evaluate to false to cause the loop to terminate.

Typically relational expressions or relational and conditional expressions are used in the second clause.

When the test is performed

The value of the second clause is tested when the statement first begins execution, and at the beginning of each iteration thereafter. Therefore, just like the **while** loop, the **for** loop is an *entry condition loop* .

When the third clause is executed

Although the third clause appears physically at the top of the loop, it isn't executed until the statements in the body of the loop have completed execution.

This is an important point since this clause is typically used to update the control variable, and perhaps other variables as well.

What the third clause can contain

Multiple expressions can appear in the third clause, separated by the comma operator. Again, those expressions will be executed from left to right. If variables are updated in the third clause and used in the body of the loop, it is important to understand that they do not get updated until the execution of the body is completed.

Declaring a variable in a *for* loop

As mentioned earlier, it is allowable to declare variables in the first clause of a **for** loop.

You can declare a variable with a given name outside *(prior to)* the **for** loop, or you can declare it inside the **for** loop, but not both.

If you declare it outside the **for** loop, you can access it either outside or inside the loop.

If you declare it inside the loop, you can access it only inside the loop. In other words, the scope of variables declared inside a **for** loop is limited to the loop.

This is illustrated in following sequence of four simple programs.

This program won't compile

The Java program shown in Listing 2 (p. 525) refuses to compile with a complaint that a variable named **cnt** has already been declared in the method when the attempt is made to declare it in the **for** loop.

Listing 2: A program that won't compile.

```
/*File for1.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
named cnt is declared twice.
**********************************************************/
class for1 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
          "Value of method var named cnt is " + cnt);

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
              "Value of loop var named cnt is " + cnt);

    System.out.println(
            "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

4.72

The program shown in Listing 3 (p. 526) also won't compile, but for a different reason.

Listing 3: Another program that won't compile.

```
/*File for2.java Copyright 1997, R.G.Baldwin
This program will not compile because the variable
declared inside the for loop is not accessible
outside the loop.
**********************************************************/
class for2 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);

    System.out.println(
                "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

4.73

The declaration of the variable named **cnt** , outside the **for** loop, was removed from Listing 3 (p. 526) and the declaration inside the loop was allowed to remain. This eliminated the problem of attempting to declare the variable twice.

However, this program refused to compile because an attempt was made to access the variable named **cnt** outside the for loop. This was not allowed because the variable was declared inside the **for** loop and the scope of the variable was limited to the loop.

This program will compile

The Java program shown in Listing 4 (p. 527) will compile and run because the variable named **cnt** that is declared inside the **for** loop is accessed only inside the **for** loop. No reference to a variable with the same name appears outside the loop.

Listing 4: A program that will compile.

```
/*File for3.java Copyright 1997, R.G.Baldwin
This program will compile because the variable declared
inside the for loop is accessed only inside the loop.
**********************************************************/
class for3 { //define the controlling class
  public static void main(String[] args){ //main method

    for(int cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);
  }//end main
}//End controlling class.
```

4.74

This program will also compile

Similarly, the program shown in Listing 5 (p. 528) will compile and run because the variable named **cnt** was declared outside the **for** loop and was not declared inside the **for** loop. This made it possible to access that variable both inside and outside the loop.

Listing 5: Another program that will compile.

```
/*File for4.java Copyright 1997, R.G.Baldwin
This program will compile and run because the variable
named cnt is declared outside the for loop and is not
declared inside the for loop.
**********************************************************/
class for4 { //define the controlling class
  public static void main(String[] args){ //main method
    int cnt = 5; //declare local method variable
    System.out.println(
              "Value of method var named cnt is " + cnt);

    for(cnt = 0; cnt < 2; cnt++)
      System.out.println(
                  "Value of loop var named cnt is " + cnt);

    System.out.println(
                "Value of method var named cnt is " + cnt);
  }//end main
}//End controlling class.  Note no semicolon required
```

4.75

Empty clauses in a *for* loop

The first and third clauses in a **for** loop can be left empty but the semicolons must be there as placeholders.

One author suggests that even the middle clause can be empty, but it isn't obvious to this author how the loop would ever terminate if there is no conditional expression to be evaluated. Perhaps the loop could be terminated by using a break inside the loop, but in that case, you might just as well use a **while** loop.

4.28.4.5 The for-each loop

There is another form of loop structure that is often referred to as a **for-each** loop. In order to appreciate the benefits of this loop structure, you need to be familiar with Java collections and iterators, both of which are beyond the scope of this module.

As near as I can tell, there is nothing that you can do with the **for-each** loop that you cannot also do with the conventional **for** loop described above. Therefore, I rarely use it. You can find a description of the **for-each** loop on this Oracle website [77] .

I don't plan to discuss it further in this module. However, before you go for a job interview, you should probably do some online research and learn about it because an interviewer could use a question about the **for-each** loop to trip you up in the Q and A portion of the interview.

[77] http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html

4.28.4.6 The do-while loop

The **do-while** loop is another implementation of the *loop* pillar mentioned earlier (p. 521) . However, it differs from the **while** loop and the **for** loop in one important respect; it is an *exit-condition* loop.

An exit-condition loop

Java provides an *exit-condition* loop having the syntax shown in Image 6 (p. 529) .

Image 6: Syntax of a do-while loop.

```
do {
    statements
              } while (conditional expression);
```

4.76

Behavior

The statements in the body of the loop continue to be executed for as long as the conditional expression evaluates to true. An exit-condition loop guarantees that the body of the loop will be executed at least one time, even if the conditional expression evaluates to false the first time it is tested.

4.28.4.7 The break and continue statements

General behavior

Although some authors suggest that the **break** and **continue** statements provide an alternative to the infamous **goto** statement of earlier programming languages, it appears that the behaviors of the **labeled break** and **labeled continue** statements are much more restrictive than a general **goto** .

4.28.4.8 Unlabeled break and continue

The **break** and **continue** statements are supported in both labeled and unlabeled form.

First consider the behavior of break and continue in their unlabeled configuration.

Use of a *break* statement

The **break** statement can be used in a switch statement or in a loop. When encountered in a switch statement, break causes control to be passed to the next statement outside the innermost enclosing switch statement.

When break is encountered in a loop, it causes control to be passed to the next statement outside the innermost enclosing loop.

As you will see later, labeled break statements can be used to pass control to the next statement following switch or loop statements beyond the innermost switch or loop statement when those statements are nested.

Use of a continue statement

The continue statement cannot be used in a switch statement, but can be used inside a loop.

When an unlabeled continue statement is encountered, it causes the current iteration of the current loop to be terminated and the next iteration to begin.

A labeled continue statement can cause control to be passed to the next iteration of an outer enclosing loop in a nested loop situation.

An example of the use of an unlabeled switch statement is given in the next section.

4.28.4.9 Labeled break and continue statements

This section discusses the use of labeled break and continue statements.

4.28.4.9.1 Labeled break Statements

One way to describe the behavior of a labeled break in Java is to say: "Break all the way out of the labeled statement."

Syntax of a labeled statement

To begin with, the syntax of a labeled statement is a label followed by a colon ahead of the statement as shown in Image 7 (p. 530) .

Image 7: Syntax of a labeled statement.

```
myLabel:  myStatement;
```

4.77

The label can be any legal Java identifier.

Behavior of labeled break

The behavior of a labeled break can best be illustrated using nested switch statements. For a comparison of labeled and unlabeled switch statements, consider the program shown in Listing 6 (p. 531) named **switch1** , which does not use a labeled break. Even though this program has a labeled statement, that statement is not referenced by a **break** . Therefore, the label is of no consequence.

Listing 6: The program named switch1.java.

```
/*File switch1.java
This is a Java application which serves as a baseline
comparison for switch2.java which uses a labeled break.
Note that the program uses nested switch statements.

The program displays the following output:

Match and break from here
Case 6 in outer switch
Default in outer switch
Beyond switch statements

**********************************************************/
class switch1 { //define the controlling class
  public static void main(String[] args){ //main method

  //Note that the following labeled switch statement is
  // not referenced by a labeled break in this program.
  // It will be referenced in the next program.
  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
    //Note that the code for this case is not followed
    // by break. Therefore, execution will fall through
    // the case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                           "Match and break from here");
                break; //break with no label
        case 2: System.out.println(
                          "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println("Case 6 in outer switch");
    default: System.out.println(
                             "Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

4.78

After reviewing **switch1.java** , consider the same program named **switch2.java** shown in Listing 7 (p. 533) , which was modified to use a labeled break.

The outputs from both programs are shown in the comments at the beginning of the program. By examining the second program, and comparing the output from the second program with the first program, you should be able to see how the use of the labeled break statement causes control to break all the way out of the labeled switch statement.

Listing 7: The program named switch2.java.

```
/*File switch2.java
This is a Java application which uses a labeled break.
Note that the program uses nested switch statements.

See switch1.java for a comparison program which does not
use a labeled break.

The program displays the following output:

Match and break from here
Beyond switch statements
**********************************************************/
class switch2 { //define the controlling class
  public static void main(String[] args){ //main method

  outerSwitch: switch(5){//labeled outer switch statement
    case 5: //execute the following switch statement
    //Note that the code for this case is not followed by
    // break. Therefore, except for the labeled break at
    // case 1, execution would fall through the case 6 and
    // the default as demonstrated in the program named
    // switch1. However, the use of the labeled break
    // causes control to break all the way out of the
    // labeled switch bypassing case 6 and the default.
      switch(1){ //inner switch statement
        case 1: System.out.println(
                              "Match and break from here");
                break outerSwitch; //break with label
        case 2: System.out.println(
                             "No match for this constant");
                break;
      }//end inner switch statement

    case 6: System.out.println(
                                 "Case 6 in outer switch");
    default: System.out.println("Default in outer switch");
  }//end outer switch statement

  System.out.println("Beyond switch statements");
  }//end main
}//End switch1 class.
```

The modified program in Listing 7 (p. 533) uses a labeled break statement in the code group for *case 1* whereas the original program in Listing 6 (p. 531) has an unlabeled break in that position.

By comparing the output from this program with the output from the previous program, you can see that execution of the labeled break statement caused control to break all the way out of the labeled switch statement completely bypassing *case 6* and default.

As you can see from examining the output, the labeled break statement causes the program to break all the way out of the switch statement which bears a matching label.

A similar situation exists when a labeled break is used in nested loops with one of the enclosing outer loops being labeled. Control will break out of the enclosing loop to which the labeled break refers. It will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

4.28.4.9.2 Labeled continue statements

Now consider use of the labeled continue statement. A **continue** statement can only be used in a loop; it cannot be used in a switch. The behavior of a labeled continue statement can be described as follows: "Terminate the current iteration and continue with the next iteration of the loop to which the label refers."

Again, it will be left as an exercise for the student to demonstrate this behavior to his or her satisfaction.

4.28.4.10 The return statement

Use of the return statement

Java supports the use of the **return** statement to terminate a method and *(optionally)* return a value to the calling method.

The return type

The type of value returned must match the type of the declared return value for the method.

The void return type

If the return value is declared as **void** , you can use the syntax shown in Image 8 (p. 534) to terminate the method. *(You can also simply allow the method to run out of statements to execute.)*

Image 8: An empty return statement.

```
return;
```

4.80

Returning a value

If the method returns a value, follow the word return with an expression *(or constant)* that evaluates to the value being returned as shown in Image 9 (p. 535) .

Image 9: Returning a value from a method.

```
return x+y;
```

4.81

Return by value only

You are allowed to return only by *value* . In the case of primitive types, this returns a copy of the returned item. In the case of objects, returning by value returns a copy of the object's reference.

What you can do with a copy the object's reference

Having a copy of the reference is just as good as having the original reference. A copy of the reference gives you access to the object.

When Java objects are destroyed

All objects in Java are stored in dynamic memory and that memory is not overwritten until all references to that memory cease to exist.

Java uses a garbage collector running on a background thread to reclaim memory from objects that have become *eligible for garbage collection* .

An object becomes eligible for garbage collection when there are no longer any variables, array elements, or similar storage locations containing a reference to the object. In other words, it becomes eligible when there is no way for the program code to find a reference to the object.

4.28.4.11 Exception handling

Exception handling is a process that modifies the flow of control of a program, so it merits being mentioned in this module. However, it is a fairly complex topic, which will be discussed in detail in future modules.

Suffice it at this point to say that whenever an exception is detected, control is transferred to exception handler code if such code has been provided. Otherwise, the program will terminate. Thus, the exception handling system merits being mentioned in discussions regarding flow of control.

4.28.5 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [78]
- Java OOP: Objects and Encapsulation [79]

[78] http://cnx.org/content/m44148
[79] http://cnx.org/content/m44153

4.28.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0230: Java OOP: Flow of Control
- File: Jb0230.htm
- Originally published: 1997
- Published at cnx.org: 11/24/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.29 Jb0230r Review[80]

4.29.1 Table of Contents

4.29.2 Preface

This module contains review questions and answers keyed to the module titled Jb0230: Java OOP: Flow of Control [81] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.29.3 Questions

4.29.3.1 Question 1 .

List and describe eight of the statements used in Java programs to alter or control the logical flow of the program.

Answer 1 (p. 544)

4.29.3.2 Question 2

Provide pseudo-code that illustrates the general syntax of a **while** statement.

Answer 2 (p. 544)

4.29.3.3 Question 3

True or false? During the execution of a **while** statement, the program will continue to execute the statement or compound statement for as long as the conditional expression evaluates to true, or until a **break** or **continue** statement is encountered. If false, explain why.

Answer 3 (p. 544)

4.29.3.4 Question 4

True or false? A **while** loop is an *entry condition* loop. If false, explain why.

Answer 4 (p. 543)

4.29.3.5 Question 5

What is the significance of an *entry condition* loop?

Answer 5 (p. 543)

[80] This content is available online at <http://cnx.org/content/m45218/1.4/>.
[81] http://cnx.org/content/m45196

4.29.3.6 Question 6

Provide pseudo-code illustrating the general syntax of the **if-else** statement.
Answer 6 (p. 543)

4.29.3.7 Question 7

Provide pseudo-code illustrating the general syntax of the **switch-case** statement.
Answer 7 (p. 543)

4.29.3.8 Question 8

Describe the behavior of a **switch-case** statement. Provide a pseudo-code fragment that illustrates your description of the behavior. Do not include a description of labeled break statements.
Answer 8 (p. 542)

4.29.3.9 Question 9

What are the three actions normally involved in the operation of a loop *(in addition to executing the code in the body of the loop)* ?
Answer 9 (p. 542)

4.29.3.10 Question 10

True or false? A **for** loop header consists of three clauses separated by colons. If false, explain why.
Answer 10 (p. 542)

4.29.3.11 Question 11

Provide pseudo-code illustrating the general syntax of a **for** loop
Answer 11 (p. 542)

4.29.3.12 Question 12

True or false? In a **for** loop, the first and third clauses within the parentheses can contain one or more expressions, separated by the comma operator. If False, explain why.
Answer 12 (p. 542)

4.29.3.13 Question 13

What is the guarantee made by the *comma operator* ?
Answer 13 (p. 541)

4.29.3.14 Question 14

True or false? The expressions within the first clause in the parentheses in a **for** loop are executed only once during each iteration of the loop. If false, explain why.
Answer 14 (p. 541)

4.29.3.15 Question 15

While any legal expression(s) may be contained in the first clause within the parentheses of a **for** loop, the first clause has a specific purpose. What is that purpose?
Answer 15 (p. 541)

4.29.3.16 Question 16

True or false? Variables can be declared and initialized within the first clause in the parentheses of a for loop. If false, explain why.

Answer 16 (p. 541)

4.29.3.17 Question 17

True or false? The second clause in the parentheses of a **for** loop consists of a single expression which must eventually evaluate to true to cause the loop to terminate. If false, explain why.

Answer 17 (p. 541)

4.29.3.18 Question 18

True or false? A **for** loop is an *exit condition* loop. If false, explain why.

Answer 18 (p. 541)

4.29.3.19 Question 19

True or false? Because a **for** loop is an *entry condition* loop, the third clause inside the parentheses is executed at the beginning of each iteration. If false, explain why.

Answer 19 (p. 541)

4.29.3.20 Question 20

True or false? A return statement is used to terminate a method and *(optionally)* return a value to the calling method. If False, explain why.

Answer 20 (p. 541)

4.29.3.21 Question 21

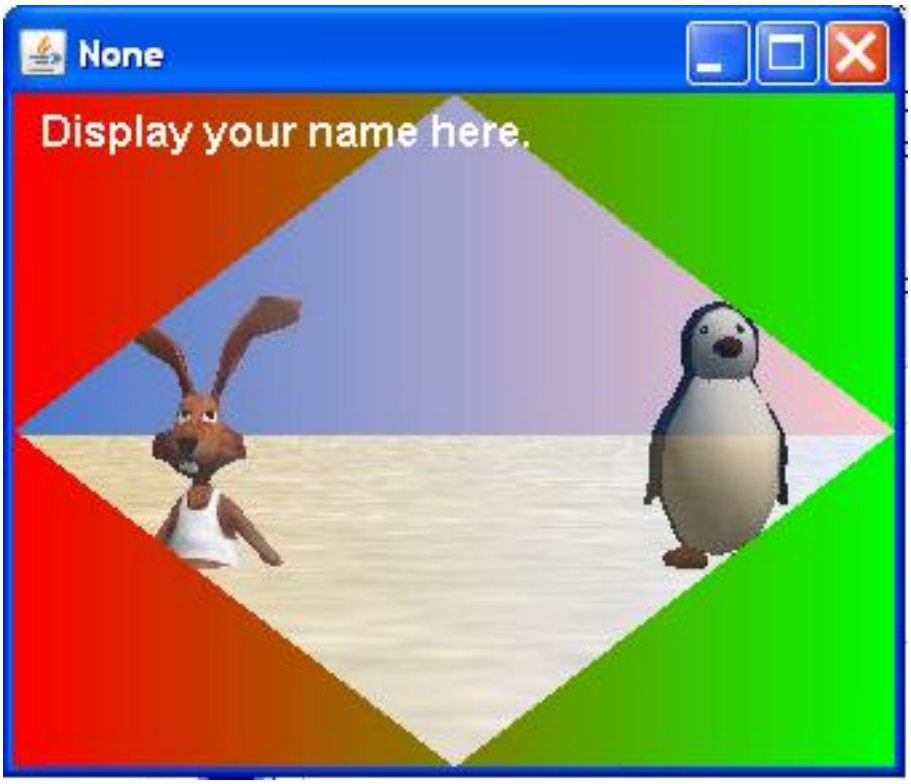
True or false? Exception handling modifies the flow of control of a Java program. If false, explain why.

Answer 21 (p. 541)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.29.4 Answers

4.29.4.1 Answer 21

True.
Back to Question 21 (p. 539)

4.29.4.2 Answer 20

True.
Back to Question 20 (p. 539)

4.29.4.3 Answer 19

False. Although the third clause appears physically at the top of the loop, it isn't executed until the statements in the body of the loop have completed execution. This is an important point since this clause is typically used to update the control variable, and perhaps other variables as well. If variables are updated in the third clause and used in the body of the loop, it is important to understand that they do not get updated until the execution of the body is completed.
Back to Question 19 (p. 539)

4.29.4.4 Answer 18

False. The value of the second clause is tested when the statement first begins execution, and at the beginning of each iteration thereafter. Therefore, the **for** loop is an *entry condition* loop.
Back to Question 18 (p. 539)

4.29.4.5 Answer 17

False. The second clause consists of a single expression which must eventually evaluate to false *(not true)* to cause the loop to terminate.
Back to Question 17 (p. 539)

4.29.4.6 Answer 16

True.
Back to Question 16 (p. 539)

4.29.4.7 Answer 15

Typically the first clause is used for initialization. The intended purpose of the first clause is initialization.
Back to Question 15 (p. 538)

4.29.4.8 Answer 14

False. The expressions in the first clause are executed only once, at the beginning of the loop, regardless of the number of iterations.
Back to Question 14 (p. 538)

4.29.4.9 Answer 13

The *comma operator* guarantees that its left operand will be executed before its right operand.
Back to Question 13 (p. 538)

4.29.4.10 Answer 12

True.
Back to Question 12 (p. 538)

4.29.4.11 Answer 11

The general syntax of a **for** loop follows:

NOTE: **Syntax of a for loop**

```
for (first clause; second clause; third clause)
  single or compound statement
```

Back to Question 11 (p. 538)

4.29.4.12 Answer 10

False: A **for** loop header consists of three clauses separated by semicolons, not colons.
Back to Question 10 (p. 538)

4.29.4.13 Answer 9

The operation of a loop normally involves the following three actions in addition to executing the code in the body of the loop:

- Initialize a control variable.
- Test the control variable in a conditional expression.
- Update the control variable.

Back to Question 9 (p. 538)

4.29.4.14 Answer 8

The pseudo-code fragment follows:

NOTE: **Syntax of a switch-case statement**

```
switch(expression){
  case constant:
    sequence of optional statements
    break; //optional
  case constant:
    sequence of optional statements
    break; //optional
.
.
.
  default //optional
    sequence of optional statements
}
```

An expression is tested against a series of unique integer constants. If a match is found, the sequence of optional statements associated with the matching constant is executed. Execution of statements continues until an optional **break** is encountered. When **break** is encountered, execution of the **switch** statement is terminated and control is passed to the next statement following the **switch** statement.

If no match is found and the optional **default** keyword along with a sequence of optional statements has been provided, those statements will be executed.

Back to Question 8 (p. 538)

4.29.4.15 Answer 7

The general syntax of the **switch-case** statement follows:

NOTE: **Syntax of a switch-case statement**

```
switch(expression){
  case constant:
    sequence of optional statements
    break; //optional
  case constant:
    sequence of optional statements
    break; //optional
.
.
.
  default //optional
    sequence of optional statements
}
```

Back to Question 7 (p. 538)

4.29.4.16 Answer 6

The general syntax of the if-else statement is:

NOTE: **Syntax of an if-else statement**

```
if(conditional expression)
  statement or compound statement;
else //optional
  statement or compound statement; //optional
```

Back to Question 6 (p. 538)

4.29.4.17 Answer 5

The significance of an *entry condition* loop is that the conditional expression is tested before the statements in the loop are executed. If it tests false initially, the statements in the loop will not be executed.

Back to Question 5 (p. 537)

4.29.4.18 Answer 4

True.

Back to Question 4 (p. 537)

4.29.4.19 Answer 3

True.
Back to Question 3 (p. 537)

4.29.4.20 Answer 2

The general syntax of a **while** statement follows :

NOTE: **Syntax of a while statement**

```
while (conditional expression)
   statement or compound statement;
```

Back to Question 2 (p. 537)

4.29.4.21 Answer 1

The following table lists the statements supported by Java for controlling the logical flow of the program.

NOTE: **Flow of control statements**

```
Statement            Typeif-else              selection
switch-case          selection
for                  loop
for-each             loop
while                loop
do-while             loop
try-catch-finally    exception handling
throw                exception handling
break                miscellaneous
continue             miscellaneous
label:               miscellaneous
return               miscellaneous
goto                 reserved by Java but not supported
```

Back to Question 1 (p. 537)

4.29.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0230r Review: Flow of Control
- File: Jb0230r.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.30 Jb0240: Java OOP: Arrays and Strings[82]

4.30.1 Table of Contents

4.30.2 Preface

This module takes a preliminary look at arrays and strings. More in-depth discussions will be provided in future modules. For example, you will find a more in-depth discussions of array objects in the following modules:

- Java OOP: Array Objects, Part 1 [83]
- Java OOP: Array Objects, Part 2 [84]
- Java OOP: Array Objects, Part 3 [85]

[82] This content is available online at <http://cnx.org/content/m45214/1.4/>.
[83] http://cnx.org/content/m44198
[84] http://cnx.org/content/m44199
[85] http://cnx.org/content/m44200

4.30.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

4.30.2.1.1 Images

4.30.2.1.2 Listings

4.30.3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, arrays, strings, etc. This module concentrates on arrays and strings.

Array and String types

Java provides a type for both arrays and strings from which objects of the specific type can be instantiated. Once instantiated, the methods belonging to those types can be called by way of the object.

4.30.4 Arrays

Arrays and Strings

Java has a true array type and a true **String** type with protective features to prevent your program from writing outside the memory bounds of the array object or the **String** object. Arrays and strings are true objects.

Declaring an array

You must declare an array before you can use it. *(More properly, you must declare a reference variable to hold a reference to the array object.)* In declaring the array, you must provide two important pieces of information:

- the name of a variable to hold a reference to the array object
- the type of data to be stored in the elements of the array object

Different declaration formats

A reference variable capable of holding a reference to an array object can be declared using either format shown in Image 1 (p. 547) .

Image 1: Formats for declaring a reference variable for an array object.

```
int[] myArray;
int myArray[];
```

4.82

Declaration does not allocate memory

As with other objects, the declaration of the reference variable does not allocate memory to contain the array data. Rather it simply allocates memory to contain a reference to the array.

Allocating memory for the array object

Memory to contain the array object must be allocated from dynamic memory using statements such as those shown in Image 2 (p. 547) .

Image 2: Allocating memory for the array object.

```
int[] myArrayX = new int[15];
int myArrayY[] = new int[25];

int[] myArrayZ = {3,4,5};
```

4.83

The statements in Image 2 (p. 547) simultaneously declare the reference variable and cause memory to be allocated to contain the array.

Also note that the last statement in Image 2 (p. 547) is different from the first two statements. This syntax not only sets aside the memory for the array object, the elements in the array are initialized by evaluating the expressions shown in the coma-separated list inside the curly brackets.

On the other hand, the array elements in the first two statements in Image 2 (p. 547) are automatically initialized with the default value for the type.

Declaration and allocation can be separated

It is not necessary to combine these two processes. You can execute one statement to declare the reference variable and another statement to cause the array object to be instantiated some time later in the program

as shown in Image 3 (p. 548) .

Image 3: Declaration and instantiation can be separated.

```
int[] myArray;
. . .
myArray = new int[25];
```

4.84

Causing memory to be set aside to contain the array object is commonly referred to as instantiating the array object *(creating an instance of the array object)* .

If you prefer to declare the reference variable and instantiate the array object at different points in your program, you can use the syntax shown in Image 3 (p. 548) . This pattern is very similar to the declaration and instantiation of all objects.

General syntax for combining declaration and instantiation

The general syntax for declaring and instantiating an array object is shown in Image 4 (p. 548) .

Image 4: General syntax for combining declaration and instantiation.

```
typeOfElements[] nameOfRefVariable =
           new typeOfElements[sizeOfArray]
```

4.85

Accessing array elements

Having instantiated an array object, you can access the elements of the array using indexing syntax that is similar to many other programming languages. An example is shown in Image 5 (p. 549) .

Image 5: An example of array indexing syntax.

```
myArray[5] = 6;

myVar = myArray[5];
```

4.86

The value of the first index

Array indices always begin with 0.

The length property of an array

The code fragment in Image 6 (p. 549) illustrates another interesting aspect of arrays. *(Note the use of* ***length*** *in the conditional clause of the* ***for*** *loop.)*

Image 6: The use of the length property in the conditional clause of a for loop.

```
for(int cnt = 0; cnt < myArray.length; cnt++)
  myArray[cnt] = cnt;
```

4.87

All array objects have a **length** property that can be accessed to determine the number of elements in the array. *(The number of elements cannot change once the array object is instantiated.)*

Types of data that you can store in an array object

Array elements can contain any Java data type including primitive values and references to ordinary objects or other array objects.

Constructing multi-dimensional arrays

All array objects contains a one-dimensional array structure. You can create multi-dimensional arrays by causing the elements in one array object to contain references to other array objects. In effect, you can create a tree structure of array objects that behaves like a multi-dimensional array.

Odd-shaped multi-dimensional arrays

The program **array01** shown in Listing 1 (p. 551) illustrates an interesting aspect of the Java arrays. Java can produce multi-dimensional arrays that can be thought of as an array of arrays. However, the

secondary arrays need not all be of the same size.

In the program shown in Listing 1 (p. 551) , a two-dimensional array of integers is declared and instantiated with the primary size *(size of the first dimension)* being three. The sizes of the secondary dimensions *(sizes of each of the sub-arrays)* is 2, 3, and 4 respectively.

Can declare the size of secondary dimension later

When declaring a two-dimensional array, it is not necessary to declare the size of the secondary dimension when the primary array is instantiated. Declaration of the size of each sub-array can be deferred until later as illustrated in this program.

Accessing an array out-of-bounds

This program also illustrates the result of attempting to access an element that is out-of-bounds. Java protects you from such programming errors.

ArrayIndexOutOfBoundsException

An exception occurs if you attempt to access out-of-bounds, as shown in the program in in Listing 1 (p. 551) .

In this case, the exception was simply allowed to cause the program to terminate. The exception could have been caught and processed by an exception handler, a concept that will be explored in a future module.

The program named array01

The entire program is shown in Listing 1 (p. 551) . The output from the program is shown in the comments at the top of the listing.

Listing 1: The program named array01.

```
/*File array01.java Copyright 1997, R.G.Baldwin
Illustrates creation and manipulation of two-dimensional
array with the sub arrays being of different lengths.

Also illustrates detection of exception when an attempt is
made to store a value out of the array bounds.

This program produces the following output:

00
012
0246
Attempt to access array out of bounds
java.lang.ArrayIndexOutOfBoundsException:
     at array01.main(array01.java: 47)

**********************************************************/
class array01 { //define the controlling class
  public static void main(String[] args){ //main method
    //Declare a two-dimensional array with a size of 3 on
    // the primary dimension but with different sizes on
    // the secondary dimension.
    //Secondary size not specified initially
    int[][] myArray = new int[3][];
    myArray[0] = new int[2];//secondary size is 2
    myArray[1] = new int[3];//secondary size is 3
    myArray[2] = new int[4];//secondary size is 4

    //Fill the array with data
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        myArray[i][j] = i * j;
      }//end inner loop
    }//end outer loop

    //Display data in the array
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        System.out.print(myArray[i][j]);
      }//end inner loop
      System.out.println();
    }//end outer loop

    //Attempt to access an out-of-bounds array element
    System.out.println(
                  "Attempt to access array out of bounds");
    myArray[4][0] = 7;
    //The above statement produces an ArrayIndexOutOfBounds
    // exception.
```



```
  }//end main
}//End array01 class.
```

Assigning one array to another array – be careful

Java allows you to assign one array to another. You must be aware, however, that when you do this, you are simply making another copy of the reference to the same data in memory.

Then you simply have two references to the same data in memory, which is often not a good idea. This is illustrated in the program named **array02** shown in Listing 2 (p. 553) .

Listing 2: The program named array02.

```
/*File array02.java Copyright 1997, R.G.Baldwin
Illustrates that when you assign one array to another
array, you end up with two references to the same array.

The output from running this program is:

firstArray contents
0 1 2
secondArray contents
0 1 2
Change a value in firstArray and display both again
firstArray contents
0 10 2
secondArray contents
0 10 2
**********************************************************/
class array02 { //define the controlling class
  int[] firstArray;
  int[] secondArray;

  array02() {//constructor
    firstArray = new int[3];
    for(int cnt = 0; cnt < 3; cnt++) firstArray[cnt] = cnt;

    secondArray = new int[3];
    secondArray = firstArray;
  }//end constructor

  public static void main(String[] args){//main method
    array02 obj = new array02();
    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
    System.out.println(
      "Change value in firstArray and display both again");
    obj.firstArray[1] = 10;

    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
```

4.30.5 Arrays of Objects

An array of objects really isn't an array of objects

There is another subtle issue that you need to come to grips with before we leave our discussion of arrays. In particular, when you create an array of objects, it really isn't an array of objects.

Rather, it is an array of object references *(or null)* . When you assign primitive values to the elements in an array object, the actual primitive values are stored in the elements of the array.

However, when you assign objects to the elements in an array , the actual objects aren't actually stored in the array elements. Rather, the objects are stored somewhere else in memory. The elements in the array contain references to those objects.

All the elements in an array of objects need not be of the same actual type

The fact that the array is simply an array of reference variables has some interesting ramifications. For example, it isn't necessary that all the elements in the array be of the same type, provided the reference variables are of a type that will allow them to refer to all the different types of objects.

For example, if you declare the array to contain references of type **Object** , those references can refer to any type of object *(including array objects)* because a reference of type **Object** can be used to refer to any object.

You can do similar things using *interface* types. I will discuss interface types in a future module.

Often need to downcast to use an Object reference

If you store all of your references as type **Object** , you will often need to downcast the references to the true type before you can use them to access the instance variables and instance methods of the objects.

Doing the downcast no great challenge as long as you can decide what type to downcast them to.

The Vector class

There is a class named **Vector** that takes advantage of this capability. An object of type **Vector** is a self-expanding array of reference variables of type **Object** . You can use an object of type **Vector** to manage a group of objects of any type, either all of the same type, or mixed.

(Note that you cannot store primitive values in elements of a non-primitive or reference type. If you need to do that, you will need to wrap your primitive values in an object of a wrapper class as discussed in an earlier module.)

A sample program using the Date class

The sample program, named **array03** and shown in Listing 3 (p. 555) isn't quite that complicated. This program behaves as follows:

- Declare a reference variable to an array of type **Date** . *(The actual type of the variable is Date[].)*
- Instantiate a three-element array of reference variables of type **Date** .
- Display the contents of the array elements and confirm that they are all null as they should be. *(When created using this syntax, new array elements contain the default value, which is null for reference types.)*
- Instantiate three objects of type **Date** and store the references to those objects in the three elements of the array.
- Access the references from the array and use them to display the contents of the individual **Date** objects.

As you might expect from the name of the class, each object contains information about the date.

Listing 3: The program named Array03.

```
/*File array03.java Copyright 1997, R.G.Baldwin

Illustrates use of arrays with objects.

Illustrates that "an array of objects" is not really an
array of objects, but rather is an array of references
to objects.  The objects are not stored in the array,
but rather are stored somewhere else in memory and the
references in the array elements refer to them.

The output from running this program is:

myArrayOfRefs contains
null
null
null

myArrayOfRefs contains
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
**********************************************************/
import java.util.*;

class array03 { //define the controlling class
  Date[] myArrayOfRefs; //Declare reference to the array

  array03() {//constructor
    //Instantiate the array of three reference variables
    // of type Date.  They will be initialized to null.
    myArrayOfRefs = new Date[3];

    //Display the contents of the array.
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(this.myArrayOfRefs[cnt]);
    System.out.println();

    //Instantiate three objects and assign references to
    // those three objects to the three reference
    // variables in the array.
    for(int cnt = 0; cnt < 3; cnt++)
      myArrayOfRefs[cnt] = new Date();

  }//end constructor
  //-------------------------------------------------------//

  public static void main(String[] args){//main method
    array03 obj = new array03();
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(obj.myArrayOfRefs[cnt]);
    System.out.println();
  }//end main
```

4.30.6 Strings

What is a string?

A string is commonly considered to be a sequence of characters stored in memory and accessible as a unit.

Java implements strings using the **String** class and the **StringBuffer** class.

What is a string literal?

Java considers a series of characters surrounded by quotation marks as shown in Image 7 (p. 556) to be a string literal.

Image 7: A string literal.

```
"This is a string literal in Java."
```

4.91

This is just an introduction to strings

A major section of a future module will be devoted to the topic of strings, so this discussion will be brief.

String objects cannot be modified

String objects cannot be changed once they have been created. If you have that need, use the **StringBuffer** class instead.

StringBuffer objects can be used to create and manipulate character data as the program executes.

4.30.6.1 String Concatenation

Java supports string concatenation using the overloaded + operator as shown in Image 8 (p. 556) .

Image 8: String concatenation.

```
"My variable has a value of " + myVar
   + " at this point in the program."
```

4.92

Coercion of an operand to type String

The overloaded + operator is used to concatenate strings. If either operand is type **String** , the other operand is coerced into type **String** and the two strings are concatenated.

Therefore, in addition to concatenating the strings, Java also converts values of other types, such as **myVar** in Image 8 (p. 556) , to character-string format in the process.

4.30.6.2 Arrays of String References

Declaring and instantiating a String array

The statement in Image 9 (p. 557) declares and instantiates an array of references to five **String** objects.

Image 9: Declaring and instantiating a String array.

```
String[] myArrayOfStringReferences = new String[5];
```

4.93

No string data at this point

Note however, that this array doesn't contain the actual **String** objects. Rather, it simply sets aside memory for storage of five references of type **String** . *(The array elements are automatically initialized to null.)* No memory has been set aside to store the characters that make up the individual **String** objects. You must allocate the memory for the actual **String** objects separately using code similar to the code shown in Image 10 (p. 557) .

Image 10: Allocating memory to contain the String objects.

```
myArrayOfStringReferences[0] = new String(
    "This is the first string.");
myArrayOfStringReferences[1] = new String(
    "This is the second string.");
```

4.94

The new operator is not required for String class

Although it was used in Image 10 (p. 557) , the **new** operator is not required to instantiate an object of type **String** . I will discuss the ability of Java to instantiate objects of type **String** without the requirement to use the **new** operator in a future module.

4.30.7 Run the programs

I encourage you to copy the code from Listing 1 (p. 551) , Listing 2 (p. 553) , and Listing 3 (p. 555) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.30.8 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [86]
- Java OOP: Objects and Encapsulation [87]

4.30.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0240: Java OOP: Arrays and Strings
- File: Jb0240.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[86] http://cnx.org/content/m44148
[87] http://cnx.org/content/m44153

4.31 Jb0240r Review[88]

4.31.1 Table of Contents

4.31.2 Preface

This module contains review questions and answers keyed to the module titled Jb0240: Java OOP: Arrays and Strings [89] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.31.3 Questions

4.31.3.1 Question 1 .

True or false? Arrays and Strings are true objects. If false, explain why.

Answer 1 (p. 568)

4.31.3.2 Question 2

True or false? It is easy to write outsides the bounds of a **String** or an array. If false, explain why.

Answer 2 (p. 568)

4.31.3.3 Question 3

You must declare a variable capable of holding a reference to an array object before you can use it. In declaring the variable, you must provide two important pieces of information. What are they?

Answer 3 (p. 567)

4.31.3.4 Question 4

Provide code fragments that illustrate the two different syntaxes that can be used to declare a variable capable of holding a reference to an array object that will store data of type int.

Answer 4 (p. 567)

4.31.3.5 Question 5

True or false? When you declare a variable capable of holding a reference to an array object, the memory required to contain the array object is automatically allocated. If false, explain why and show how memory can be allocated.

Answer 5 (p. 567)

[88] This content is available online at <http://cnx.org/content/m45208/1.4/>.
[89] http://cnx.org/content/m45214

4.31.3.6 Question 6

True or false? It is required that you simultaneously declare the name of the variable and cause memory to be allocated to contain the array object in a single statement. If false, explain why and show code fragments to illustrate your answer.

Answer 6 (p. 567)

4.31.3.7 Question 7

True or false? Array indices always begin with 1. If false, explain why.

Answer 7 (p. 566)

4.31.3.8 Question 8

What is the name of the property of arrays that can be accessed to determine the number of elements in the array? Provide a sample code fragment that illustrates the use of this property.

Answer 8 (p. 566)

4.31.3.9 Question 9

What types of data can be stored in array objects?

Answer 9 (p. 566)

4.31.3.10 Question 10

True or false? Just as in other languages, when you create a multi-dimensional array, the secondary arrays must all be of the same size. If false, explain your answer. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings [90] that illustrates your answer.

Answer 10 (p. 566)

4.31.3.11 Question 11

True or false? Just as in other languages, when declaring a two-dimensional array, it is necessary to declare the size of the secondary dimension when the array is declared. If false, explain your answer. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings [91] that illustrates your answer.

Answer 11 (p. 566)

4.31.3.12 Question 12

True or false? Java allows you to assign one array to another. Explain what happens when you do this. Then provide a code fragment that illustrates your answer or refer to a sample program in Jb0240: Java OOP: Arrays and Strings [92] that illustrates your answer.

Answer 12 (p. 566)

4.31.3.13 Question 13

Give a brief description of the concept of a string and list the names of two classes used to implement strings?

Answer 13 (p. 566)

[90] http://cnx.org/content/m45214
[91] http://cnx.org/content/m45214
[92] http://cnx.org/content/m45214

4.31.3.14 Question 14

What is the syntax that is used to create a literal string? Provide a code fragment to illustrate your answer.
Answer 14 (p. 565)

4.31.3.15 Question 15

Explain the difference between objects of types **String** and **StringBuffer** .
Answer 15 (p. 565)

4.31.3.16 Question 16

Provide a code fragment that illustrates how to concatenate strings.
Answer 16 (p. 565)

4.31.3.17 Question 17

Provide a code fragment that declares and instantiates an array object capable of storing references to two **String** objects. Explain what happens when this code fragment is executed. Then show a code fragment that will allocate memory for the actual **String** objects.
Answer 17 (p. 565)

4.31.3.18 Question 18

Write a Java application that illustrates the creation and manipulation of a two-dimensional array with the sub arrays being of different lengths. Also cause your application to illustrate that an attempt to access an array element out of bounds results in an exception being thrown. Catch and process the exception. Display a termination message with your name.
Answer 18 (p. 564)

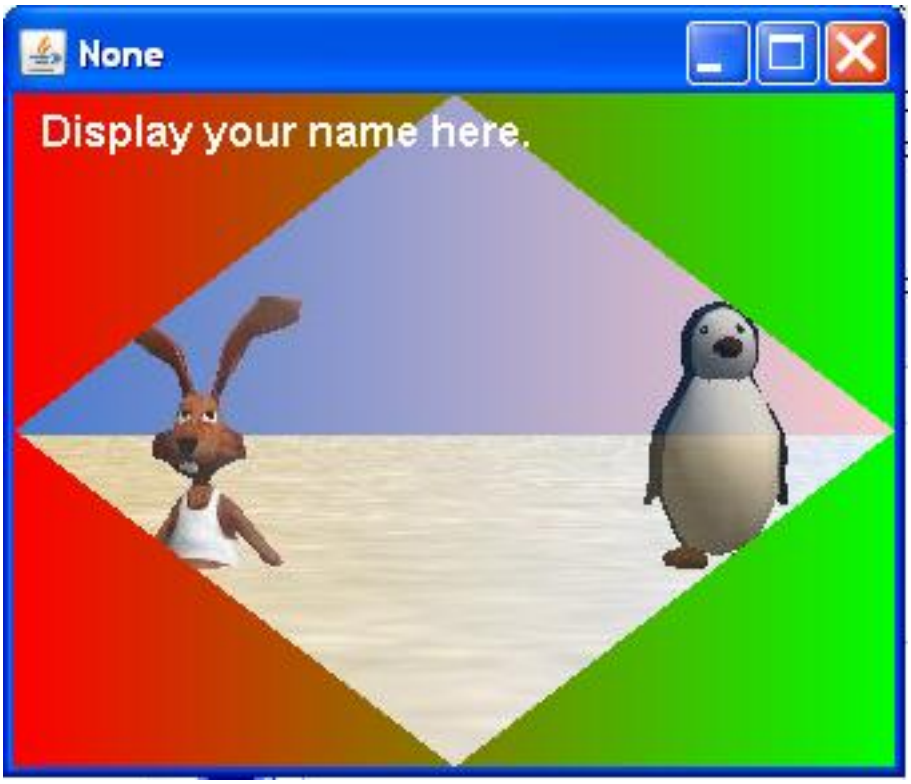
4.31.4 Listings

- Listing 1 (p. 564) . Listing for Answer 18.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

4.31.5 Answers

4.31.5.1 Answer 18

Listing 1: Listing for Answer 18.

```
class SampProg10 { //define the controlling class
  public static void main(String[] args){ //define main
    //Declare a two-dimensional array with a size of 3 on
    // the primary dimension but with different sizes on
    // the secondary dimension.
    //Secondary size not specified
    int[][] myArray = new int[3][];
    myArray[0] = new int[2];//secondary size is 2
    myArray[1] = new int[3];//secondary size is 3
    myArray[2] = new int[4];//secondary size is 4

    //Fill the array with data
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        myArray[i][j] = i * j;
      }//end inner loop
    }//end outer loop

    //Display data in the array
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        System.out.print(myArray[i][j]);
      }//end inner loop
      System.out.println();
    }//end outer loop

    //Attempt to access an out-of-bounds array element
    try{
      System.out.println(
                  "Attempt to access array out of bounds");
      myArray[4][0] = 7;
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println(e);
    }//end catch

    System.out.println("Terminating, Dick Baldwin");

  }//end main
}//End SampProg10 class.  Note no semicolon required
```

4.95

Back to Question 18 (p. 561)

4.31.5.2 Answer 17

The following statement declares and instantiates an array object capable of storing references to two **String** objects.

NOTE:

```
String[] myArrayOfStringReferences = new String[2];
```

Note however, that this array object doesn't contain the actual string data. Rather, it simply sets aside memory for storage of two references to **String** objects. No memory has been set aside to store the characters that make up the individual strings. You must allocate the memory for the actual **String** objects separately using code similar to the following.

NOTE:

```
myArrayOfStringReferences[0] = new String(
                    "This is the first string.");
myArrayOfStringReferences[1] = new String(
                   "This is the second string.");
```

Back to Question 17 (p. 561)

4.31.5.3 Answer 16

Java supports string concatenation using the overloaded + operator as shown in the following code fragment:

NOTE:

```
"My variable has a value of " + myVar +
" at this point in the program."
```

Back to Question 16 (p. 561)

4.31.5.4 Answer 15

String objects cannot be modified once they have been created. **StringBuffer** objects can be modified
Back to Question 15 (p. 561)

4.31.5.5 Answer 14

The Java compiler considers a series of characters surrounded by quotation marks to be a literal string, as in the following code fragment:

NOTE:

```
"This is a literal string in Java."
```

Back to Question 14 (p. 561)

4.31.5.6 Answer 13

A string is commonly considered to be a sequence of characters stored in memory and accessible as a unit. Java implements strings using the **String** class and the **StringBuffer** class.

Back to Question 13 (p. 560)

4.31.5.7 Answer 12

Java allows you to assign one array to another. When you do this, you are simply making another copy of the reference to the same data in memory. Then you have two references to the same data in memory. This is illustrated in the program named **array02.java** in Jb0240: Java OOP: Arrays and Strings [93] .

Back to Question 12 (p. 560)

4.31.5.8 Answer 11

False. When declaring a two-dimensional array, it is not necessary to declare the size of the secondary dimension when the array is declared. Declaration of the size of each sub-array can be deferred until later as illustrated in the program named **array01.java** in Jb0240: Java OOP: Arrays and Strings [94] .

Back to Question 11 (p. 560)

4.31.5.9 Answer 10

False. Java can be used to produce multi-dimensional arrays that can be viewed as an array of arrays. However, the secondary arrays need not all be of the same size. See the program named **array01.java** in Jb0240: Java OOP: Arrays and Strings [95] .

Back to Question 10 (p. 560)

4.31.5.10 Answer 9

Array objects can contain any Java data type including primitive values, references to ordinary objects, and references to other array objects.

Back to Question 9 (p. 560)

4.31.5.11 Answer 8

All array objects have a **length** property that can be accessed to determine the number of elements in the array as shown below.

NOTE:

```
for(int cnt = 0; cnt < myArray.length; cnt++)
  myArray[cnt] = cnt;
```

Back to Question 8 (p. 560)

4.31.5.12 Answer 7

False. Array indices always begin with 0.

Back to Question 7 (p. 560)

[93] http://cnx.org/content/m45214
[94] http://cnx.org/content/m45214
[95] http://cnx.org/content/m45214

4.31.5.13 Answer 6

False. While it is possible to simultaneously declare the name of the variable and cause memory to be allocated to contain the array object, it is not necessary to combine these two processes. You can execute one statement to declare the variable and another statement to cause the memory for the array object to be allocated as shown below.

NOTE:

```
int[] myArray;
.
.
.
myArray = new int[25];
```

Back to Question 6 (p. 560)

4.31.5.14 Answer 5

False. As with other objects. the declaration of the variable does not allocate memory to contain the array object. Rather it simply allocates memory to contain a reference to the array object. Memory to contain the array object must be allocated from dynamic memory using statements such as the following.

NOTE:

```
int[] myArray = new int[15];
int myArray[] = new int[25];
int[] myArray = {1,2,3,4,5}
```

Back to Question 5 (p. 559)

4.31.5.15 Answer 4

NOTE:

```
int[] myArray;
int myArray[];
```

Back to Question 4 (p. 559)

4.31.5.16 Answer 3

In declaring the variable, you must provide two important pieces of information:

- the name of the variable
- the type of the variable, which indicates the type of data to be stored in the array

Back to Question 3 (p. 559)

4.31.5.17 Answer 2

False. Java has a true array type and a true **String** type with protective features to prevent your program from writing outside the memory bounds of the array or the **String** .

Back to Question 2 (p. 559)

4.31.5.18 Answer 1

True.

Back to Question 1 (p. 559)

4.31.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0240r Review: Arrays and Strings
- File: Jb0240r.htm
- Originally published: 1997
- Published at cnx.org: 11/26/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.32 Jb0250: Java OOP: Brief Introduction to Exceptions[96]

4.32.1 Table of Contents

[96] This content is available online at <http://cnx.org/content/m45211/1.3/>.

4.32.2 Preface

This module provides a very brief treatment of exception handling. The topic is discussed in detail in the module titled Java OOP: Exception Handling [97] . The topic is included in this *Programming Fundamentals* section simply to introduce you to the concept.

4.32.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

4.32.2.1.1 Listings

- Listing 1 (p. 570) . The program named simple1.

4.32.3 Discussion

What is an exception?

According to The Java Tutorials [98] , *"An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions."*

A very common example of an exception given in textbooks is code that attempts to divide by zero *(this is easy to demonstrate)* .

Throwing an exception

Common terminology states that when this happens, the system *throws an exception* . If a thrown exception is not *caught* , a runtime error may occur.

Purpose of exception handling

The purpose of exception handling is to make it possible for the program to either attempt to recover from the problem, or at worst shut down the program in a graceful manner, whenever an exception occurs.

Java supports exception handling

Java, C++, and some other programming languages support exception handling in similar ways.

In Java, the exception can be thrown either by the system or by code created by the programmer. There is a fairly long list of exceptions that will be thrown automatically by the Java runtime system.

Checked exceptions cannot be ignored

Included in that long list of automatic exceptions is a subset known as "checked" exceptions. Checked exceptions cannot be ignored by the programmer. A method must either specify *(declare)* or catch all "checked" exceptions that can be thrown in order for the program to compile.

An example of specifying an exception

I explain the difference between specifying and catching an exception in Java OOP: Exception Handling [99] . For now, suffice it to say that the code that begins with the word "throws" in Listing 1 (p. 570) specifies *(declares)* an exception that can be thrown by the code inside the **main** method.

If this specification is not made, the program will not compile.

[97] http://cnx.org/content/m44202
[98] http://docs.oracle.com/javase/tutorial/essential/exceptions/
[99] http://cnx.org/content/m44202

Listing 1: The program named simple1.

```
/*File simple1.java Copyright 1997, R.G.Baldwin
************************************************************/
class simple1 { //define the controlling class
  public static void main(String[] args)
                                throws java.io.IOException {
    int ch1, ch2 = '0';

    System.out.println(
      "Enter some text, terminate with #");

    //Get and save individual bytes
    while( (ch1 = System.in.read() ) != '#') ch2 = ch1;

    //Display the character immediately before the #
    System.out.println("The char before the # was "
      + (char)ch2);
  }//end main
}//End simple1 class.
```

4.96

The program in Listing 1 (p. 570) does not throw any exceptions directly nor does it attempt to catch any exceptions. However, it can throw exceptions indirectly through its call to **System.in.read** .

Because **IOException** is a checked exception, the **main** method must either specify it or catch it . Otherwise the program won't compile. In this case, the **main** method specifies the exception as opposed to catching it.

Very brief treatment

As mentioned earlier, this is a very brief treatment of a fairly complex topic that is discussed in much more detail in the module titled Java OOP: Exception Handling [100] . The topic was included at this point simply to introduce you to the concept of exceptions.

4.32.4 Run the program

II encourage you to copy the code from Listing 1 (p. 570) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.32.5 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [101]

[100] http://cnx.org/content/m44202
[101] http://cnx.org/content/m44148

- Java OOP: Objects and Encapsulation [102]

4.32.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0250: Java OOP: Brief Introduction to Exceptions
- File: Jb0250.htm
- Originally published: 1997
- Published at cnx.org: 11/26/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.33 Jb0260: Java OOP: Command-Line Arguments[103]

4.33.1 Table of Contents

4.33.2 Preface

Although the use of command-line arguments is rare is this time of Graphical User Interfaces *(GUI)* , they are still useful for testing and debugging code. This module explains the use of command-line arguments in Java.

[102] http://cnx.org/content/m44153
[103] This content is available online at <http://cnx.org/content/m45246/1.4/>.

4.33.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

4.33.2.1.1 Listings

- Listing 1 (p. 573) . Illustration of command-line arguments.

4.33.3 Discussion

Familiar example from DOS

Java programs can be written to accept command-line-arguments.

DOS users will be familiar with commands such as the following:

NOTE: **Familiar DOS command**

```
copy fileA fileB
```

In this case, **copy** is the name of the program to be executed, while **fileA** and **fileB** are command-line arguments.

Java syntax for command-line arguments

The Java syntax for supporting command-line arguments is shown below *(note the formal argument list for the **main** method)* .

NOTE: **Java syntax for command-line arguments**

```
public static void main(String[] args){
. . .
}//end main method
```

In Java, the formal argument list for the **main** method must appear in the method signature whether or not the program is written to support the use of command-line arguments. If the argument isn't used, it is simply ignored.

Where the arguments are stored

The parameter **args** contains a reference to a one-dimensional array object of type **String** .

Each of the elements in the array *(including the element at index zero)* contains a reference to an object of type **String** . Each object of type String encapsulates one command-line argument.

The number of arguments entered by the user

Recall from an earlier module on arrays that the number of elements in a Java array can be obtained from the **length** property of the array. Therefore, the number of arguments entered by the user is equal to the value of the **length** property. If the user didn't enter any arguments, the value will be zero.

Command-line arguments are separated by the space character. If you need to enter an argument that contains a space, surround the entire argument with quotation mark characters as in *"My command line argument"* .

The first command-line argument is encapsulated in the **String** object referred to by the contents of the array element at index 0, the second argument is referred to by the element at index 1, etc.

Sample Java program

The sample program in Listing 1 (p. 573) illustrates the use of command-line arguments.

Listing 1: Illustration of command-line arguments.

```
/*File cmdlin01.java Copyright 1997, R.G.Baldwin
This Java application illustrates the use of Java
command-line arguments.

When this program is run from the command line as follows:

java cmdlin01 My command line arguments

the program produces the following output:

My
command
line
arguments
**********************************************************/
class cmdlin01 { //define the controlling class
  public static void main(String[] args){ //main method
    for(int i=0; i < args.length; i++)
    System.out.println( args[i] );
  }//end main
}//End cmdlin01 class.
```

4.97

The output from running this program for a specific input is shown in the comments at the beginning of the program.

4.33.4 Run the program

I encourage you to copy the code Listing 1 (p. 573) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.33.5 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [104]
- Java OOP: Objects and Encapsulation [105]

[104] http://cnx.org/content/m44148
[105] http://cnx.org/content/m44153

4.33.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0260: Java OOP: Command-Line Arguments
- File: Jb0260.htm
- Originally published: 1997
- Published at cnx.org: 11/27/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.34 Jb0260r Review[106]

4.34.1 Table of Contents

4.34.2 Preface

This module contains review questions and answers keyed to the module titled Jb0260: Java OOP: Command-Line Arguments [107] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.34.3 Questions

4.34.3.1 Question 1 .

Provide a common example of a command-line statement that illustrates the use of command-line-arguments.

Answer 1 (p. 579)

4.34.3.2 Question 2

Describe the purpose of command-line-arguments.

Answer 2 (p. 579)

4.34.3.3 Question 3

True or false? In Java, syntax provisions must be made in the method signature for the **main** method to accommodate command-line-arguments even if the remainder of the program is not designed to make use of them. If False, explain why.

Answer 3 (p. 579)

4.34.3.4 Question 4

Provide the method signature for the **main** method in a Java application that is designed to accommodate the use of command-line-arguments. Identify the part of the method signature that applies to command-line-arguments and explain how it works.

Answer 4 (p. 578)

4.34.3.5 Question 5

Explain how a Java application can determine the number of command-line-arguments actually entered by the user.

Answer 5 (p. 578)

[106] This content is available online at <http://cnx.org/content/m45244/1.5/>.
[107] http://cnx.org/content/m45246

4.34.3.6 Question 6

Write a program that illustrates the handling of command-line arguments in Java.
Answer 6 (p. 578)

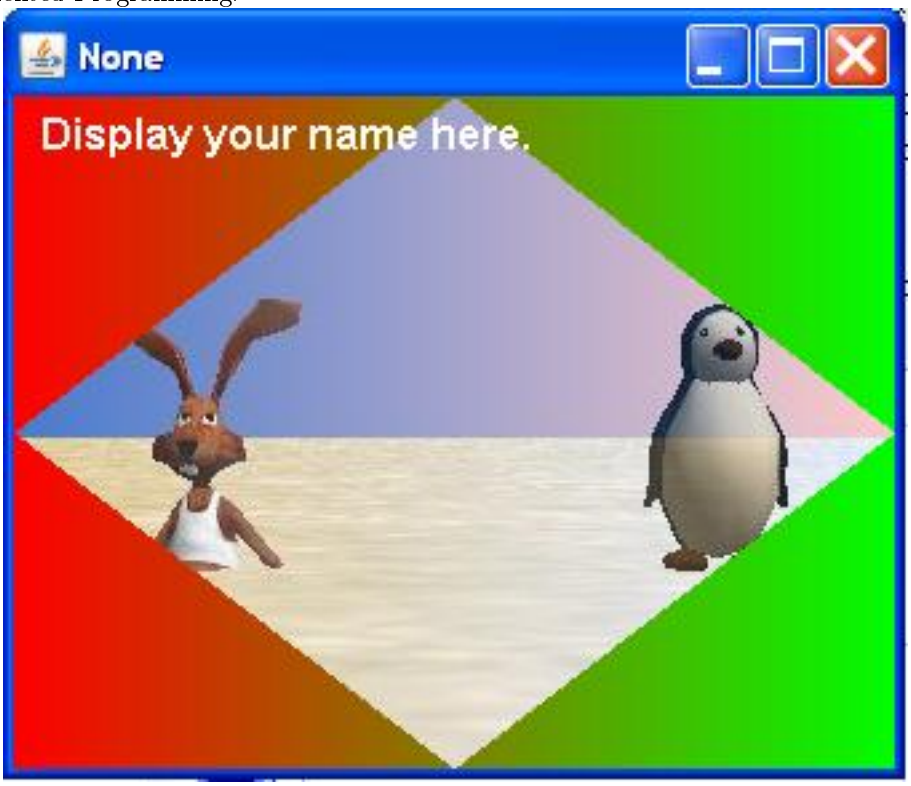
4.34.4 Listings

- Listing 1 (p. 578) . Handling command-line arguments in Java.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.


Prob05a.jpg
Put your name here

4.34.5 Answers

4.34.5.1 Answer 6

Listing 1: Handling command-line arguments in Java.

```
/*File SampProg11.java from module 32
Copyright 1997, R.G.Baldwin
Without reviewing the following solution, write a Java
application that illustrates the handling of command-line
arguments in Java.

Provide a termination message that displays your name.
**********************************************************/
class SampProg11 { //define the controlling class
  public static void main(String[] args){ //define main
    for(int i=0; i < args.length; i++)
      System.out.println( args[i] );
    System.out.println("Terminating, Dick Baldwin");
  }//end main
}//End SampProg11 class.
```

4.98

Back to Question 6 (p. 576)

4.34.5.2 Answer 5

The number of command-line arguments is equal to the number of elements in the array of references to **String** objects referred to by **args** . The number of elements is indicated by the value of the **length** property of the array. If the value is zero, the user didn't enter any command-line arguments.

Back to Question 5 (p. 575)

4.34.5.3 Answer 4

The Java syntax for command-line arguments is shown below.

NOTE: **Java syntax for command-line arguments.**

```
public static void main(String[] args){
. . .
}//end main method
```

Each of the elements in the array object referred to by **args** *(including the element at position zero)* contains a reference to a **String** object that encapsulates one of the command-line arguments.

Back to Question 4 (p. 575)

4.34.5.4 Answer 3

True.

Back to Question 3 (p. 575)

4.34.5.5 Answer 2

Command-line-arguments are used in many programming and computing environments to provide information to the program at startup that it will need to fulfill its mission during that particular invocation.

Back to Question 2 (p. 575)

4.34.5.6 Answer 1

DOS users will be familiar with commands such as the following:

NOTE: **Command-line arguments in DOS**

```
copy fileA fileB
```

In this case, **copy** is the name of the program to be executed, while **fileA** and **fileB** are command-line arguments.

Back to Question 1 (p. 575)

4.34.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0260r Review: Command-Line Arguments
- File: Jb0260r.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.35 Jb0270: Java OOP: Packages[108]

4.35.1 Table of Contents

4.35.2 Preface

This module explains the concept of packages and provides a sample program that illustrates the concept.

4.35.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.35.2.1.1 Listings

4.35.3 Introduction

Before you can understand much about packages, you will need to understand the *classpath environment variable* , so that is where I will begin the discussion.

After learning about the classpath environment variable, you will learn how to create your own packages.

4.35.4 Classpath environment variable

The purpose of the *classpath environment variable* is to tell the JVM and other Java applications *(such as the javac compiler)* where to find class files and class libraries. This includes those class files that are part of the JDK and class files that you may create yourself.

I am assuming that you already have some familiarity with the use of environment variables in your operating system. All of the discussion in this module will be based on the use of a generic form of Windows. *(By generic, I mean not tied to any particular version of Windows.)* Hopefully you will be able to translate the information to your operating system if you are using a different operating system.

[108] This content is available online at <http://cnx.org/content/m45229/1.4/>.

NOTE: **In a nutshell** Environment variables provide information that the operating system uses to do its job.

There are usually a fairly large number of environment variables installed on a machine at any give time. If you would like to see what kind of environment variables are currently installed on your machine, bring up a command-line prompt and enter the command **set** . This should cause the names of several environment variables, along with their settings to be displayed on your screen.

While you are at it, see if any of those items begin with **CLASSPATH=** . If so, you already have a classpath environment variable set on your machine, but it may not contain everything that you need.

I am currently using a Windows 7 operating system and no classpath environment variable is set on it. I tend to use the **-cp** switch option *(see Listing 4 (p. 587))* in the JDK to set the classpath on a temporary basis when I need it to be set.

Rather than trying to explain all of the ramifications regarding the classpath, I will simply refer you to an Oracle document on the topic titled Setting the class path [109] .

I will also refer you to Java OOP: The Guzdial-Ericson Multimedia Class Library [110] where I discuss the use of the classpath environment variable with a Java multimedia class library.

Some rules

There are some rules that you must follow when working with the classpath variable, and if you fail to do so, things simply won't work.

For example, if your class files are in a jar file, the classpath must end with the name of that jar file.

On the other hand, if the class files are not in a jar file, the classpath must end with the name of the folder that contains the class files.

Your classpath must contain a fully-qualified path name for every folder that contains class files of interest, or for every jar file of interest. The paths should begin with the letter specifying the drive and end either with the name of the jar file or the name of the folder that contains the class files. .

If you followed the default JDK installation procedure and are simply compiling and executing Java programs in the current directory you probably won't need to set the classpath. By default, the system already knows *(or can figure out)* how to allow you to compile and execute programs in the current directory and how to use the JDK classes that come as part of the JDK.

However, if you are using class libraries other than the standard Java library, are saving your class files in one or more different folders, or are ready to start creating your own packages, you will need to set the classpath so that the system can find the class files in your packages.

4.35.5 Developing your own packages

One of the problems with storing all of your class files in one or two folders is that you will likely experience name conflicts between class files.

Every Java program can consist of a large number of separate classes. A class file is created for each class that is defined in your program, even if they are all combined into a single source file.

It doesn't take very many programs to create a lot of class files, and it isn't long before you find yourself using the same class names over again. If you do, you will end up overwriting class files that were previously stored in the folder.

For me, it only takes two GUI programs to get in trouble because I tend to use the same class names in every program for certain standard operations such as closing a **Frame** or processing an **ActionEvent** . For the case of the **ActionEvent** , the body of the class varies from one application to the next so it doesn't make sense to turn it into a library class.

So we need a better way to organize our class files, and the Java package provides that better way.

The Java package approach allows us to store our class files in a hierarchy of folders *(or a jar file that represents that hierarchy)* while only requiring that the classpath variable point to the top of the

[109] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html
[110] http://cnx.org/content/m44148

hierarchy. The remaining hierarchy structure is encoded into our programs using package directives and import directives.

Now here is a little jewel of information that cost me about seven hours of effort to discover when I needed it badly.

When I first started developing my own packages, I spent about seven hours trying to determine why the compiler wouldn't recognize the top-level folder in my hierarchy of package folders.

I consulted numerous books by respected authors and none of them was any help at all. I finally found the following statement in the Java documentation *(when all else fails, read the documentation)* . By the way, a good portion of that wasted seven hours was spent trying to find this information in the documentation which is voluminous.

> NOTE: **The following text was extracted directly from the JDK 1.1 documentation**
> If you want the CLASSPATH to point to class files that belong to a package, you should specify a path name that includes the path to the directory one level above the directory having the name of your package.
>
> For example, suppose you want the Java interpreter to be able to find classes in the package mypackage. If the path to the mypackage directory is C:\java\MyClasses\mypackage, you would set the CLASSPATH variable as follows:
>
> set CLASSPATH=C:\java\MyClasses

If you didn't catch the significance of this, read it again. When you are creating a classpath variable to point to a folder containing classes, it must point to the folder. However, when you are creating a classpath variable to point to your package, it must point to the folder that is one level above the directory that is the top-level folder in your package.

Once I learned that I had to cause the classpath to point to the folder immediately above the first folder in the hierarchy that I was including in my package directives, everything started working.

4.35.5.1 The package directive

So, what is the purpose of a package directive, and what does it look like?

> NOTE: **Purpose of a package directive** The purpose of the package directive is to identify a particular class *(or group of classes contained in a single source file (compilation unit))* as belonging to a specific package.

This is very important information for a variety of reasons, not the least of which is the fact that the entire access control system is wrapped around the concept of a class belonging to a specific package. For example, code in one package can only access public classes in a different package.

Stated in the simplest possible terms, a package is a group of class files contained in a single folder on your hard drive.

At compile time, a class can be identified as being part of a particular package by providing a package directive at the beginning of the source code..

A package directive, if it exists, must occur at the beginning of the source code *(ignoring comments and white space)* . No text other than comments and whitespace is allowed ahead of the package directive.

If your source code file does not contain a package directive, the classes in the source code file become part of the *default package* . With the exception of the default package, all packages have names, and those names are the same as the names of the folders that contain the packages. There is a one-to-one correspondence between folder names and package names. The default package doesn't have a name.

Some examples of package directives that you will see in upcoming sample programs follow:

> NOTE: **Example package directives**

```
package Combined.Java.p1;
package Combined.Java.p2;
```

Given the following as the classpath on my hypothetical machine,

CLASSPATH=.;c:\Baldwin\JavaProg

these two package directives indicate that the class files being governed by the package directives are contained in the following folders:

NOTE:

```
c:\Baldwin\JavaProg\Combined\Java\p1
c:\Baldwin\JavaProg\Combined\Java\p2
```

Notice how I concatenated the package directive to the classpath setting and substituted the backslash character for the period when converting from the package directive to the fully-qualified path name.

Code in one package can refer to a class in another package *(if it is otherwise accessible)* by qualifying the class name with its **package name as follows** :

NOTE:

```
Combined.Java.p2.Access02 obj02 =
       new Combined.Java.p2.Access02();
```

Obviously, if we had to do very much of that, it would become very burdensome due to the large amount of typing required. As you already know, the *import directive* is available to allow us to specify the package containing the class just once at the beginning of the source file and then refer only to the class name for the remainder of that source file.

4.35.5.2 The import directive

This discussion will be very brief because you have been using import directives since the very first module. Therefore, you already know what they look like and how to use them.

If you are interested in the nitty-gritty details *(such as what happens when you provide two import directives that point to two different packages containing the same class file name)* , you can consult the Java Language Reference by Mark Grand, or you can download the Java language specification from Oracle's Java website.

The purpose of the import directive is to help us avoid the burdensome typing requirement (p. 583) described in the previous section when referring to classes in a different package.

An import directive makes the definitions of classes from other packages available on the basis of their file names alone.

You can have any number of import directives in a source file. However, they must occur after the package directive *(if there is one)* and before any class or interface declarations.

There are essentially two different forms of the import directive, one with and the other without a wild card character *(*)* . These two forms are illustrated in the following box.

NOTE: **Two forms of import directives**

```
import java.awt.*
import java.awt.event.ActionEvent
```

The first import directive makes all of the class files in the **java.awt** package available for use in the code in a different package by referring only to their file names.

The second import directive makes only the class file named **ActionEvent** in the **java.awt.event** package available by referring only to the file name.

4.35.5.3 Compiling programs with package directives

So, how do you compile programs containing package directives? There are probably several ways. I am going to describe one way that I have found to be successful.

First, you must create your folder hierarchy to match the package directive that you intend to use. Remember to construct this hierarchy downward relative to the folder specified at the end of your classpath setting. If you have forgotten the critical rule (p. 582) in this respect, go back and review it.

Next, place source code files in each of the folders where you intend for the class files associated with those source code files to reside. *(After you have compiled and tested the program, you can remove the source code files if you wish.)*

You can compile the source code files individually if you want to, but that isn't necessary.

One of the source code files will contain the *controlling class* . The controlling class is the class that contains the **main** method that will be executed when you run the program from the command line using the JVM.

Make the directory containing that source code file be the current directory. *(If you don't know what the current directory is, go out and get yourself a* ***DOS For Dummies*** *book and read it.)*

Each of the source code files must contain a package directive that specifies the package that will contain the compiled versions of all the class definitions in that source code file. Using the instructions that I am giving you, that package directive will also describe the folder that contains the source code file.

Any of the source code files containing code that refers to classes in a different package must also contain the appropriate import directives, or you must use fully-qualified package names to refer to those classes.

Then use the **javac** program with your favorite options to compile the source code file containing the controlling class. This will cause all of the other source code files containing classes that are linked to the code in the controlling class, either directly or indirectly, to be compiled also. At least an attempt will be made to compile them all. You may experience a few compiler errors if your first attempt at compilation is anything like mine.

Once you eliminate all of the compiler errors, you can test the application by using the **java** program with your favorite options to execute the controlling class.

Once you are satisfied that everything works properly, you can copy the source code files over to an archive folder and remove them from the package folders if you want to do so.

Finally, you can also convert the entire hierarchy of package folders to a jar file if you want to, and distribute it to your client. If you don't remember how to install it relative to the classpath variable, go back and review that part of the module.

Once you have reached this point, how do you execute the program. I will show you how to execute the program from the command line in the sample program in the next section. *(Actually I will encapsulate command-line commands in a batch file and execute the batch file. That is a good way to guard against typing errors.)*

4.35.5.4 Sample program

The concept of packages can get very tedious in a hurry. Let's take a look at a sample program that is designed to pull all of this together.

This application consists of three separate source files located in three different packages. Together they illustrates the use of package and import directives, along with **javac** to build a standalone Java application consisting of classes in three separate packages.

(If you want to confirm that they are really in different packages, just make one of the classes referred to by the controlling class non-public and try to compile the program.)

In other words, in this sample program, we create our own package structure and populate it with a set of cooperating class files.

A folder named **jnk** is a child of the root folder on the M-drive.
A folder named **SampleCode** is a child of the folder named **jnk** .
A folder named **Combined** is a child of the folder named **SampleCode** .
A folder named **Java** is a child of the folder named **Combined** .
Folders named **p1** and **p2** are children of the folder named **Java** .
The file named **Package00.java** , shown in Listing 1 (p. 585) is stored in the folder named **Java** .

Listing 1: File: Package00.java.

```
/*File Package00.java Copyright 1997, R.G.Baldwin
Illustrates use  of package and import directives to
build an application consisting of classes in three
separate packages.

The output from running the program follows:

Starting Package00
Instantiate obj of public classes in different packages
Constructing Package01 object in folder p1
Constructing Package02 object in folder p2
Back in main of Package00
**********************************************************/
package Combined.Java; //package directive

//Two import directives
import Combined.Java.p1.Package01;//specific form
import Combined.Java.p2.*; //wildcard form

class Package00{
  public static void main(String[] args){ //main method
    System.out.println("Starting Package00");

    System.out.println("Instantiate obj of public " +
                         "classes in different packages");
    new Package01();//Instantiate objects of two classes
    new Package02();// in different packages.

    System.out.println("Back in main of Package00");

  }//end main method
}//End Package00 class definition.
```

The file named **Package01.java** , shown in Listing 2 (p. 586) is stored in the folder named **p1** .

Listing 2: File Package01.java.

```
/*File Package01.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**********************************************************/
package Combined.Java.p1;
public class Package01 {
  public Package01(){//constructor
    System.out.println(
             "Constructing Package01 object in folder p1");
  }//end constructor
}//End Package01 class definition.
```

4.100

The file named **Package02.java** , shown in Listing 3 (p. 586) is stored in the folder named **p2** .

Listing 3: File Package02.java.

```
/*File Package02.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**********************************************************/
package Combined.Java.p2;
public class Package02 {
  public Package02(){//constructor
    System.out.println(
             "Constructing Package02 object in folder p2");
  }//end constructor
}//End Package02 class definition.
```

4.101

The file named **CompileAndRun** .bat, shown in Listing 4 (p. 587) is stored in the folder named **SampleCode** .

Listing 4: File: CompileAndRun.bat.

```
echo off
rem This file is located in folder named M:\SampleCode,
rem which is Parent of folder Combined.

del Combined\Java\*.class
del Combined\Java\p1\*.class
del Combined\Java\p2\*.class

javac -cp M:\jnk\SampleCode Combined\Java\Package00.java

java -cp M:\jnk\SampleCode Combined.Java.Package00

pause
```

4.102

The controlling class named **Package00** is stored in the package named **Combined.Java** , as declared in Listing 1 (p. 585) .

The class named **Package01** is stored in the package named **Combined.Java.p1** , as declared in Listing 2 (p. 586) .

The class named **Package02** is stored in the package named **Combined.Java.p2** , as declared in Listing 3 (p. 586) .

The controlling class named **Package00** imports **Combined.Java.p1.Package01** and **Combined.Java.p2.*** , as declared in Listing 1 (p. 585) .

Code in the **main** method of the controlling class in Listing 1 (p. 585) instantiates objects of the other two classes in different packages. The constructors for those two classes announce that they are being constructed.

The two classes being instantiated are **public** . Otherwise, it would not be possible to instantiate them from outside their respective packages.

This program was tested using JDK 7 under Windows by executing the batch file named **CompileAndRun.bat** .

The classpath is set to the parent folder of the folder named **Combined** *(M:\jnk\SampleCode)* by the **-cp** switch in the file named **CompileAndRun.bat** .

The output from running the program is shown in the comments at the beginning of Listing 1 (p. 585) .

4.35.6 Run the program

I encourage you to copy the code from Listing1 (p. 585) through Listing 4 (p. 587) into a properly constructed tree of folders. Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.35.7 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

Java OOP: The Guzdial-Ericson Multimedia Class Library [111]
- Java OOP: Objects and Encapsulation [112]

4.35.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0270: Java OOP: Packages
- File: Jb0270.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.36 Jb0280: Java OOP: String and StringBuffer[113]

4.36.1 Table of Contents

[111] http://cnx.org/content/m44148
[112] http://cnx.org/content/m44153
[113] This content is available online at <http://cnx.org/content/m45237/1.3/>.

4.36.2 Preface

This module discusses the **String** and **StringBuffer** classes in detail.

4.36.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

4.36.2.1.1 Listings

4.36.3 Introduction

A string in Java is an object. Java provides two different string classes from which objects that encapsulate string data can be instantiated:

- **String**
- **StringBuffer**

The **String** class is used for strings that are not allowed to change once an object has been instantiated *(an immutable object)* . The **StringBuffer** class is used for strings that may be modified by the program.

4.36.4 You can't modify a String object, but you can replace it

While the contents of a **String** object cannot be modified, a reference to a **String** object can be caused to point to a different **String** object as illustrated in the sample program shown in Listing 1 (p. 590) . Sometimes this makes it appear that the original **String** object is being modified.

Listing 1: File String01.java.

```
/*File String01.java Copyright 1997, R.G.Baldwin
This application illustrates the fact that while a String
object cannot be modified, the reference variable can be
modified to point to a new String object which can have
the appearance of modifying the original String object.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is

Display original string values
THIS STRING IS NAMED str1
This string is named str2
Replace str1 with another string
Display new string named str1
THIS STRING IS NAMED str1 This string is named str2
Terminating program

**********************************************************/

class String01{
  String str1 = "THIS STRING IS NAMED str1";
  String str2 = "This string is named str2";

  public static void main(String[] args){
    String01 thisObj = new String01();
    System.out.println("Display original string values");
    System.out.println(thisObj.str1);
    System.out.println(thisObj.str2);
    System.out.println("Replace str1 with another string");
    thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
    System.out.println("Display new string named str1");
    System.out.println(thisObj.str1);
    System.out.println("Terminating program");
  }//end main()
}//end class String01
```

4.103

It is important to note that the following statement does not modify the original object pointed to by the reference variable named **str1** .

NOTE:

```
thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
```

Rather, this statement creates a new object, which is concatenation of two existing objects and causes the reference variable named **str1** to point to the new object instead of the original object.

The original object then becomes eligible for garbage collection *(unless there is another reference to the object hanging around somewhere)* .

Many aspects of string manipulation can be accomplished in this manner, particularly when the methods of the **String** class are brought into play.

4.36.5 Why are there two string classes?

According to *The Java Tutorial* by Campione and Walrath:

> NOTE: "Because they are constants, Strings are typically cheaper than StringBuffers and they can be shared. So it's important to use Strings when they're appropriate."

4.36.6 Creating String and StringBuffer objects

The **String** and **StringBuffer** classes have numerous overloaded constructors and many different methods. I will attempt to provide a sampling of constructors and methods that will prepare you to explore other constructors and methods on your own.

The next sample program touches on some of the possibilities provided by the wealth of constructors and methods in the **String** and **StringBuffer** classes.

At this point, I will refer you to Java OOP: Java Documentation [114] where you will find a link to online Java documentation. Among other things, the online documentation provides a list of the overloaded constructors and methods for the **String** and **StringBuffer** classes.

As of Java version 7, there are four overloaded constructors in the **StringBuffer** class and about thirteen different overloaded versions of the **append** method. There are many additional methods in the **StringBuffer** class including about twelve overloaded versions of the **insert** method.

As you can see, there are lots of constructors and lots of methods from which to choose. One of your challenges as a Java programmer will be to find the right methods of the right classes to accomplish what you want your program to accomplish.

4.36.6.1 The sample program named String02

The sample program shown in Listing 2 (p. 592) illustrates a variety of ways to create and initialize **String** and **StringBuffer** objects.

[114] http://cnx.org/content/m45117

Listing 2: File String02.java.

```
/*File String02.java Copyright 1997, R.G.Baldwin
Illustrates different ways to create String objects and
StringBuffer objects.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is as follows.  In some cases,
manual line breaks were inserted to make the material fin
this presentation format.

Create a String the long way and display it
String named str2

Create a String the short way and display it
String named str1

Create, initialize, and display a StringBuffer using new
StringBuffer named str3

Try to create/initialize StringBuffer without
using new - not allowed

Create an empty StringBuffer of default length
Now put some data in it and display it
StringBuffer named str5

Create an empty StringBuffer and specify length
when it is created
Now put some data in it and display it
StringBuffer named str6

Try to create and append to StringBuffer without
using new -- not allowed

**********************************************************/

class String02{
  void d(String displayString){//method to display strings
    System.out.println(displayString);
  }//end method d()

  public static void main(String[] args){
    String02 o = new String02();//obj of controlling class

    o.d("Create a String the long way and display it");
    String str1 = new String("String named str2");
    o.d(str1 + "\n");

    o.d("Create a String the short way and display it");
    String str2 = "String named str1";
    o.d(str2 + "\n");

    o.d("Create, initialize, and display a StringBuffer " +
                                                "using new");
```

4.36.6.2 Alternative String instantiation constructs

The first thing to notice is that a **String** object can be created using either of the following constructs:

NOTE: **Alternative String instantiation constructs**

```
String str1 = new String("String named str2");

String str2 = "String named str1";
```

The first approach uses the **new** operator to instantiate an object while the shorter version doesn't use the new operator.

Later I will discuss the fact that

- the second approach is not simply a shorthand version of the first construct, but that
- they involve two different compilation scenarios with the second construct being more efficient than the first.

4.36.6.3 Instantiating StringBuffer objects

The next thing to notice is that a similar alternative strategy does not hold for the **StringBuffer** class.

For example, it is not possible to create a **StringBuffer** object without use of the **new** operator. *(It is possible to create a reference to a* ***StringBuffer*** *object but it is later necessary to use the* ***new*** *operator to actually instantiate an object.)*

Note the following code fragments that illustrate allowable and non-allowable instantiation scenarios for **StringBuffer** objects.

NOTE: **Instantiating StringBuffer objects**

```
//allowed
StringBuffer str3 = new StringBuffer(
                      "StringBuffer named str3");

//not allowed
//StringBuffer str4 = "StringBuffer named str4";


o.d("Try to create and append to StringBuffer " +
             "without using new -- not allowed");
//StringBuffer str7;
//str7.append("StringBuffer named str7");
```

4.36.6.4 Declaration, memory allocation, and initialization

To review what you learned in an earlier module, three steps are normally involved in creating an object *(but the third step may be omitted)* .

- declaration
- memory allocation
- initialization

The following code fragment performs all three steps:

NOTE: **Declaration, memory allocation, and initialization**

```
StringBuffer str3 =
  new StringBuffer("StringBuffer named str3");
```

The code
StringBuffer str3
declares the type and name of a reference variable of the correct type for the benefit of the compiler.
The **new** operator allocates memory for the new object.
The constructor call
StringBuffer("StringBuffer named str3")
constructs and initializes the object.

4.36.6.5 Instantiating an empty StringBuffer object

The instantiation of the **StringBuffer** object shown above (p. 594) uses a version of the constructor that accepts a **String** object and initializes the **StringBuffer** object when it is created.

The following code fragment instantiates an empty **StringBuffer** object of a default capacity and then uses a version of the **append** method to put some data into the object. *(Note that the data is actually a **String** object – a sequence of characters surrounded by quotation marks.)*

NOTE: **Instantiating an empty StringBuffer object**

```
//default initial length
StringBuffer str5 = new StringBuffer();

//modify length as needed
str5.append("StringBuffer named str5");
```

It is also possible to specify the capacity when you instantiate a **StringBuffer** object.

Some authors suggest that if you know the final length of such an object, it is more efficient to specify that length when the object is instantiated than to start with the default length and then require the system to increase the length "on the fly" as you manipulate the object.

This is illustrated in the following code fragment. This fragment also illustrates the use of the **length** method of the **String** class just to make things interesting. *(A simple integer value for the capacity of the **StringBuffer** object would have worked just as well.)*

NOTE: **Instantiating a StringBuffer object of a non-default length**

```
StringBuffer str6 = new StringBuffer(
            "StringBuffer named str6".length());
str6.append("StringBuffer named str6");
```

4.36.7 Accessor methods

The following quotation is taken directly from *The Java Tutorial* by Campione and Walrath.

> NOTE: "An object's instance variables are encapsulated within the object, hidden inside, safe from inspection or manipulation by other objects. With certain well-defined exceptions, the object's methods are the only means by which other objects can inspect or alter an object's instance variables. Encapsulation of an object's data protects the object from corruption by other objects and conceals an object's implementation details from outsiders. This encapsulation of data behind an object's methods is one of the cornerstones of object-oriented programming."

The above statement lays out an important consideration in good object-oriented programming.

The methods used to obtain information about an object are often referred to as *accessor methods* .

4.36.7.1 Constructors and methods of the String class

I told you in an earlier section (p. 591) that the **StringBuffer** class provides a large number of overloaded constructors and methods. The same holds true for the **String** class.

Once again, I will refer you to Java OOP: Java Documentation [115] where you will find a link to online Java documentation. Among other things, the documentation provides a list of the overloaded constructors and methods for the **String** class

4.36.7.2 String objects encapsulate data

The characters in a **String** object are not directly available to other objects. However, as you can see from the documentation, there are a large number of methods that can be used to access and manipulate those characters. For example, in an earlier sample program *(Listing 2* (p. 592) *)* , I used the **length** method to access the number of characters stored in a **String** object as shown in the following code fragment.

NOTE:

```
StringBuffer str6 = new StringBuffer(
                    "StringBuffer named str6".length());
```

In this case, I applied the **length** method to a literal string, but it can be applied to any valid representation of an object of type **String** .

I then passed the value returned by the **length** method to the constructor for a **StringBuffer** object.

As you can determine by examining the argument lists for the various methods of the **String** class,

- some methods return data stored in the string while
- other methods return information about that data.

For example, the **length** method returns information about the data stored in the **String** object.

Methods such as **charAt** and **substring** return portions of the actual data.

Methods such **toUpperCase** can be thought of as returning the data, but returning it in a different format.

4.36.7.3 Creating String objects without calling the constructor

Methods in other classes and objects may create **String** objects without an explicit call to the constructor by the programmer. For example the **toString** method of the **Float** class receives a **float** value as an incoming parameter and returns a reference to a **String** object that represents the **float** argument.

[115] http://cnx.org/content/m45117

4.36.8 Memory management by the StringBuffer class

If the additional characters cause the size of the **StringBuffer** to grow beyond its current capacity when characters are added, additional memory is automatically allocated.

However, memory allocation is a relatively expensive operation and you can make your code more efficient by initializing **StringBuffer** capacity to a reasonable first guess. This will minimize the number of times memory must be allocated for it.

When using the **insert** methods of the **StringBuffer** class, you specify the index *before which* you want the data inserted.

4.36.9 The toString method

Frequently you will need to convert an object to a **String** object because you need to pass it to a method that accepts only **String** values *(or perhaps for some other reason)* .

All classes inherit the **toString** method from the **Object** class. Many of the classes *override* this method to provide an implementation that is meaningful for objects of that class.

In addition, you may sometimes need to *override* the **toString** method for classes that you define to provide a meaningful **toString** behavior for objects of that class.

I explain the concept of overriding the **toString** method in detail in the module titled Java OOP: Polymorphism and the Object Class [116] .

4.36.10 Strings and the Java compiler

In Java, you specify literal strings between double quotes as in:

NOTE: **Literal strings**

```
"I am a literal string of the String type."
```

You can use literal strings anywhere you would use a **String** object.

You can also apply **String** methods directly to a literal string as in an earlier program (p. 592) that calls the **length** method on a literal string as shown below.

NOTE: **Using String methods with literal strings**

```
StringBuffer str6 = new StringBuffer(
       StringBuffer named str6".length());
```

Because the compiler automatically creates a new **String** object for every literal string, you can use a literal string to initialize a **String** object *(without use of the new operator)* as in the following code fragment from a previous program (p. 590) :

NOTE:

```
String str1 = "THIS STRING IS NAMED str1";
```

The above construct is equivalent to, but more efficient than the following, which, according to *The Java Tutorial* by Campione and Walrath, ends up creating two **String** objects instead of one:

[116] http://cnx.org/content/m44190

NOTE:

```
String str1 = new String("THIS STRING IS NAMED str1");
```

In this case, the compiler creates the first **String** object when it encounters the literal string, and the second one when it encounters **new String()** .

4.36.11 Concatenation and the + operator

The plus (+) operator is overloaded so that in addition to performing the normal arithmetic operations, it can also be used to concatenate strings.

This will come as no surprise to you because we have been using code such as the following since the beginning of this group of *Programming Fundamentals* modules:

NOTE:

```
String cat = "cat";


System.out.println("con" + cat + "enation");
```

According to Campione and Walrath, Java uses **StringBuffer** objects behind the scenes to implement concatenation. They indicate that the above code fragment compiles to:

NOTE:

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").
                      append(cat).append("enation"));
```

Fortunately, that takes place behind the scenes and we don't have to deal directly with the syntax.

4.36.12 Run the programs

I encourage you to copy the code from Listing 1 (p. 590) and Listing 2 (p. 592) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.36.13 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP [117] tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [118]
- Java OOP: Objects and Encapsulation [119]

[117] http://cnx.org/content/m45222
[118] http://cnx.org/content/m44148
[119] http://cnx.org/content/m44153

4.36.14 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0280: Java OOP: String and StringBuffer
- File: Jb0280.htm
- Originally published: 1997
- Published at cnx.org: 11/25/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.37 Jb0280r Review[120]

4.37.1 Table of Contents

4.37.2 Preface

This module contains review questions and answers keyed to the module titled Jb0280: Java OOP: String and StringBuffer [121] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

4.37.3 Questions

4.37.3.1 Question 1 .

Java provides two different string classes from which string objects can be instantiated. What are they?

Answer 1 (p. 607)

4.37.3.2 Question 2

True or false? The **StringBuffer** class is used for strings that are not allowed to change. The **String** class is used for strings that are modified by the program. If false, explain why.

Answer 2 (p. 607)

4.37.3.3 Question 3

True or false? While the contents of a **String** object cannot be modified, a reference to a **String** object can be caused to point to a different **String** object. If false, explain why.

Answer 3 (p. 607)

4.37.3.4 Question 4

True or false? The use of the **new** operator is required for instantiation of objects of type **String** . If false, explain your answer.

Answer 4 (p. 607)

4.37.3.5 Question 5

True or false? The use of the **new** operator is required for instantiation of objects of type **StringBuffer** . If false, explain your answer

Answer 5 (p. 606)

[120] This content is available online at <http://cnx.org/content/m45241/1.4/>.
[121] http://cnx.org/content/m45237

4.37.3.6 Question 6

Provide a code fragment that illustrates how to instantiate an empty **StringBuffer** object of a default length and then use a version of the **append** method to put some data into the object.

Answer 6 (p. 606)

4.37.3.7 Question 7

Without specifying any explicit numeric values, provide a code fragment that will instantiate an empty **StringBuffer** object of the correct initial length to contain the string *"StringBuffer named str6"* and then store that string in the object.

Answer 7 (p. 606)

4.37.3.8 Question 8

Provide a code fragment consisting of a single statement showing how to use the **Integer** wrapper class to convert a string containing digits to an integer and store it in a variable of type **int** .

Answer 8 (p. 606)

4.37.3.9 Question 9

Explain the difference between the **capacity** method and the **length** method of the **StringBuffer** class.

Answer 9 (p. 606)

4.37.3.10 Question 10

True or false? The following is a valid code fragment. If false, explain why.

NOTE:

```
StringBuffer str6 =
  new StringBuffer("StringBuffer named str6".length());
```

Answer 10 (p. 606)

4.37.3.11 Question 11

Which of the following code fragments is the most efficient, first or second?

NOTE:

```
String str1 = "THIS STRING IS NAMED str1";
```

```
String str1 = new String("THIS STRING IS NAMED str1");
```

Answer 11 (p. 606)

4.37.3.12 Question 12

Write a Java application that illustrates the fact that while a **String** object cannot be modified, the reference variable can be modified to point to a new **String** object, which can have the appearance of modifying the original **String** object.

Answer 12 (p. 605)

4.37.3.13 Question 13

Write a Java application that illustrates different ways to create **String** objects and **StringBuffer** objects.
Answer 13 (p. 604)

4.37.3.14 Question 14

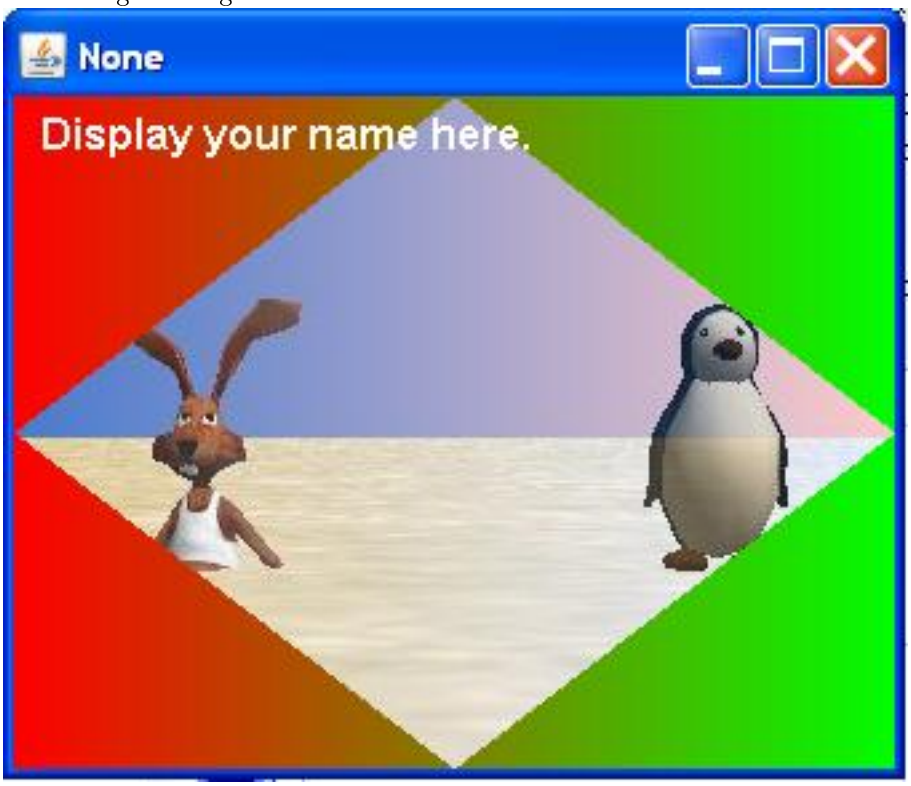
Write a Java application that illustrates conversion from string to numeric.
Answer 14 (p. 603)

4.37.4 Listings

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.


Prob05a.jpg
Put your name here

4.37.5 Answers

4.37.5.1 Answer 14

Listing 1: File SampProg26.java.

```
/*File SampProg26.java from module 50
Copyright 1997, R.G.Baldwin
Without viewing the solution that follows, write a Java
application that illustrates conversion from string to
numeric, similar to the atoi() function in C.

The output from the program should be:
The value of the int variable num is 3625
============================================================
*/

class SampProg26{
  public static void main(String[] args){
    int num = new Integer("3625").intValue();
    System.out.println(
      "The value of the int variable num is " + num);
  }//end main()
}//end class SampProg26
```

4.105

Back to Question 14 (p. 601)

4.37.5.2 Answer 13

Listing 2: File SampProg25.java.

```
/*File SampProg25.java from module 50
Copyright 1997, R.G.Baldwin
Write a Java application that illustrates different ways to
create String objects and StringBuffer objects.

The output from this program should be (line breaks
manually inserted to make it fit the format):

Create a String using new and display it
String named str2

Create a String without using new and display it
String named str1

Create, initialize, and display a StringBuffer using new
StringBuffer named str3

Try to create/initialize StringBuffer without using new

Create an empty StringBuffer of default length
Now put some data in it and display it
StringBuffer named str5

Create an empty StringBuffer and specify length when
created
Now put some data in it and display it
StringBuffer named str6

Try to create and append to StringBuffer without using new
**********************************************************/

class SampProg25{
  void d(String displayString){//method to display strings
    System.out.println(displayString);
  }//end method d()

  public static void main(String[] args){
    //instantiate an object to display methods
    SampProg25 o = new SampProg25();

    o.d("Create a String using new and display it");
    String str1 = new String("String named str2");
    o.d(str1 + "\n");

    o.d(
    "Create a String without using new and display it");
    String str2 = "String named str1";
    o.d(str2 + "\n");

    o.d("Create, initialize, and display a StringBuffer "
      + "using new");
```

Back to Question 13 (p. 601)

4.37.5.3 Answer 12

Listing 3: File SampProg24.java.

```
/*File SampProg24.java from module 50
Copyright 1997, R.G.Baldwin
Without viewing the solution that follows, Write a Java
application that illustrates the fact that while a String
object cannot be modified, the reference variable can be
modified to point to a new String object which can have the
appearance of modifying the original String object.

The output from this program should be

Display original string values
THIS STRING IS NAMED str1
This string is named str2
Replace str1 with another string
Display new string named str1
THIS STRING IS NAMED str1 This string is named str2
Terminating program
**********************************************************/

class SampProg24{
  String str1 = "THIS STRING IS NAMED str1";
  String str2 = "This string is named str2";

  public static void main(String[] args){
    SampProg24 thisObj = new SampProg24();
    System.out.println("Display original string values");
    System.out.println(thisObj.str1);
    System.out.println(thisObj.str2);
    System.out.println(
      "Replace str1 with another string");
    thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
    System.out.println("Display new string named str1");
    System.out.println(thisObj.str1);
    System.out.println("Terminating program");
  }//end main()
}//end class SampProg24
```

4.107

Back to Question 12 (p. 600)

4.37.5.4 Answer 11

The first code fragment is the most efficient.
Back to Question 11 (p. 600)

4.37.5.5 Answer 10

True.
Back to Question 10 (p. 600)

4.37.5.6 Answer 9

The **capacity** method returns the amount of space currently allocated for the **StringBuffer** object. The **length** method returns the amount of space used.
Back to Question 9 (p. 600)

4.37.5.7 Answer 8

NOTE:

```
int num = new Integer("3625").intValue();
```

Back to Question 8 (p. 600)

4.37.5.8 Answer 7

NOTE:

```
StringBuffer str6 =
  new StringBuffer("StringBuffer named str6".length());
str6.append("StringBuffer named str6");
```

Back to Question 7 (p. 600)

4.37.5.9 Answer 6

NOTE:

```
StringBuffer str5 =
  new StringBuffer();//accept default initial length
str5.append(
      "StringBuffer named str5");//modify length as needed
```

Back to Question 6 (p. 600)

4.37.5.10 Answer 5

True.
Back to Question 5 (p. 599)

4.37.5.11 Answer 4

False. A String object can be instantiated using either of the following statements:

NOTE:

```
String str1 = new String("String named str2");

String str2 = "String named str1";
```

Back to Question 4 (p. 599)

4.37.5.12 Answer 3

True.
Back to Question 3 (p. 599)

4.37.5.13 Answer 2

False. This statement is backwards. The **String** class is used for strings that are not allowed to change. The **StringBuffer** class is used for strings that are modified by the program.
Back to Question 2 (p. 599)

4.37.5.14 Answer 1

The two classes are:

- String
- StringBuffer

Back to Question 1 (p. 599)

4.37.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0280r Review: String and StringBuffer
- File: Jb0280r.htm
- Originally published: 1997
- Published at cnx.org: 11/29/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

4.38 Jb0290: The end of Programming Fundamentals[122]

4.38.1 Looking ahead

You have now reached the end of this group of Programming Fundamentals [123] modules.

The next stop along your journey to become a Java/OOP programmer should be either the OOP Self-Assessment [124] , or the course material for the ITSE 2321 OOP [125] tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [126]
- Java OOP: Objects and Encapsulation [127]

4.38.2 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0290: Java OOP: The end of Programming Fundamentals
- File: Jb0290.htm
- Published: 11/29/12
- Revised: 01/02/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[122] This content is available online at <http://cnx.org/content/m45257/1.3/>.
[123] http://cnx.org/content/m45179
[124] http://cnx.org/content/m45252
[125] http://cnx.org/content/m45222
[126] http://cnx.org/content/m44148
[127] http://cnx.org/content/m44153

Chapter 5

ITSE 2321 Object-Oriented Programming (Java)

5.1 Jy0020: Java OOP: Preface to ITSE 2321[1]

5.1.1 Table of Contents

5.1.2 Welcome

Welcome to the course material for ***ITSE 2321 - Object-Oriented Programming (Java)*** , which I teach at Austin Community College [2] in Austin, TX.

The college website for this course is: http://www.austincc.edu/baldwin/ [3]

5.1.3 Course description

As of November 2012, the description for this course reads:

"ITSE 2321 - Object-Oriented Programming (Java)

Introduction to object-oriented programming. Emphasis on the fundamentals of structured design with classes, including development, testing, implementation, and documentation. Includes object-oriented programming techniques, classes, and objects."

[1] This content is available online at <http://cnx.org/content/m45222/1.8/>.
[2] http://www.austincc.edu/
[3] http://www.austincc.edu/baldwin/

5.1.4 Course prerequisite

The prerequisite for the course is COSC 1336 or department approval.

As of November 2012, the description for the prerequisite course reads:

"COSC 1336 - Programming Fundamentals I

Introduces the fundamental concepts of structured programming. Topics include software development methodology, data types, control structures, functions, arrays, and the mechanics of running, testing, and debugging. This course assumes computer literacy. This course requires the same math skills necessary for College Algebra. Students should either have taken or be currently enrolled in College Algebra or a course that requires College Algebra."

5.1.4.1 Prerequisite waiver

Beginning in August of 2013, you might want to petition the department head for a waiver of the prerequisite course if you meet the following requirements:

- You understand and can answer at least 80-percent of the questions in modules Ap0005 [4] through Ap0060 [5] in the section titled OOP Self-Assessment [6] in a *"closed-book"* setting.
- You understand and can write at least 80-percent of the programs in the *Challenge program questions* in modules Ap0005 [7] through Ap0060 [8] in the section titled OOP Self-Assessment [9] in a *"closed-book"* setting.
- You understand and can answer at least 80-percent of the questions posed on the *Review* pages in the section titled Programming Fundamentals [10] in a *"closed-book"* setting

5.1.5 Prior to enrolling

I recommend that you understand and be able to answer at least 80-percent of the questions in modules AP010 [11] through AP0060 [12] in the self-assessment test [13] in a *"closed-book"* setting.

I also recommend that you understand and can write at least 80-percent of the programs in the *Challenge program questions* in modules Ap0005 [14] through Ap0060 [15] in the section titled OOP Self-Assessment [16] in a *"closed-book"* setting.

I also recommend that you read and/or study all of the modules in the Programming Fundamentals [17] section in whatever depth is necessary to ensure that you can answer at least 80-percent of the questions posed on the *Review* pages of that section in a *"closed-book"* setting.

5.1.6 Course material

This course material consists of a more than 30 different modules arranged in the following major sections:

- Essence of OOP

[4] http://cnx.org/content/m45252
[5] http://cnx.org/content/m45264
[6] http://cnx.org/content/m45252
[7] http://cnx.org/content/m45252
[8] http://cnx.org/content/m45264
[9] http://cnx.org/content/m45252
[10] http://cnx.org/content/m45179
[11] http://cnx.org/content/m45284
[12] http://cnx.org/content/m45264
[13] http://cnx.org/content/m45252
[14] http://cnx.org/content/m45252
[15] http://cnx.org/content/m45264
[16] http://cnx.org/content/m45252
[17] http://cnx.org/content/m45179

- Multimedia
- Practice Tests

5.1.6.1 Essence of OOP

The modules in the Essence of OOP section are more or less theoretical in nature. Sample programs in this section are intended to illustrate the OOP concepts being discussed with no effort being made to cause those programs to have any relation to real-world applications.

5.1.6.2 Multimedia

The modules in the Multimedia section are intended to illustrate OOP concepts using sample programs that clearly represent real-world applications. In particular, most of the sample programs in this section use OOP concepts to manipulate digital images of the sort that are produced by your digital camera. *(See some examples here* [18] *.)*

5.1.7 Classroom schedule

During a typical 16-week semester, I normally attempt to discuss about two modules per week. Usually I ping-pong back and forth between the *Essence* modules and the *Multimedia* modules, discussing one of each type during each week of the semester.

5.1.8 Downloads

I encourage you to take advantage of all of the download options that cnx.org has to offer in order to customize this material for use in your organized courses or for personal self study.

And if you find the material useful, I would like to hear more about how you are using it.

5.1.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0020: Java OOP: Preface to ITSE 2321
- File: Jy0020.htm
- Published: 11/25/12
- Revised: 01/17/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

[18] http://cnx.org/content/m44148

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.2 Essence of OOP

5.2.1 Java1600: Objects and Encapsulation[19]

5.2.1.1 Table of Contents

5.2.1.2 Preface

This module is the first in a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.1.2.1 The essence of OOP

My dictionary provides several definitions for the word essence. Among those definitions are the following:

- The property necessary to the nature of a thing
- The most significant property of a thing

Thus, this miniseries will describe and discuss the necessary and most significant aspects of OOP using Java. In other words, I will discuss the essence of OOP using Java. For the first few modules, I will provide that information in a high-level format, devoid of any requirement to understand detailed Java syntax. In those cases where an understanding of Java syntax is required, I will provide the necessary syntax information in the form of supplementary notes.

Therefore, if you have a general understanding of computer programming, you should be able to read and understand the modules in this miniseries, even if you don't have a strong background in the Java programming language.

5.2.1.2.2 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.1.2.2.1 Listings

[19] This content is available online at <http://cnx.org/content/m44153/1.3/>.

5.2.1.3 Preview

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module will concentrate on encapsulation. Encapsulation will be used as a springboard for a discussion of objects.

A description of an object-oriented program will be provided, along with a description of an object, and how it relates to encapsulation.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects. For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects. You will learn that a Java object is an instance of a class.

You will see a little bit of Java code, used to create an object, and then to send a message to that object (invoke a method on the object).

You will learn about Java references and reference variables. You will also learn a little about memory allocation for objects and variables in Java.

5.2.1.4 Discussion and sample code

Purpose of the miniseries

As mentioned earlier, I will describe and discuss the necessary and most significant aspects of OOP using Java.

The three pillars

Most books on OOP will tell you that in order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

(Some books will also add abstraction and/or late binding to the list. I tend to think of these two topics as being included in one or more of the three concepts listed above.)

Begin with encapsulation

Generally, speaking, these three concepts increase in difficulty going down the list from top to bottom. Therefore, I will begin with Encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program?

Many authors would answer this question something like the following:

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object?

An object is a software construct that encapsulates data, along with the ability to use or modify that data, into a software entity.

What is encapsulation?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

"Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface.

The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. However, I will try to illustrate it in the paragraphs that follow.

A real-world analogy

Abstract concepts, such as the concept of an object or encapsulation, can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the radio in your car.

The ability to store data

Your car radio probably has the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data through use of the human interface that is provided by the manufacturer of the radio.)*

The data that can be stored in your car radio probably includes a list of five or more frequencies that correspond to your favorite radio stations.

Using the stored data

The radio provides a mechanism *(human interface)* that allows you to use the data stored therein.

When you press one of the frequency-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)*

If you have previously stored a favorite frequency in the storage location corresponding to that button, pressing the button *(sending the message)* will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

Please change your state

What I have done here is to send a message to the radio object asking it to change its state. The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. *(Alternately, we might say that the radio object sent a message back to me.)*

We say that an object has changed its state when one or more data values stored in the object have been modified.

We also say that when an object responds to a message, it will usually perform an action, change its state, return a value, or some combination of the above.

Please perform an action

Following this, when I press that button *(send a message)* , the radio object will be automatically tuned to that frequency.

NOTE: **Historical note:** While the ability to cause your car radio to remember your list of favorite stations may seem like a miracle of modern digital electronics, the truth is that radios

had this capability long before they contained digital electronics. My first car had a radio that accomplished this feat using strings, pulleys, and levers.

As I recall, in order to set the frequency for a button, I had to manually tune the radio to a station by turning a knob, pull one of the buttons out about a quarter of an inch, and then push it in again. From that point until I did the same thing again, whenever I pressed that button, some kind of a mechanical contraption caused a big rotary capacitor to turn just the right amount to tune for a particular radio station.

Also, I remember my grandfather having a table-model radio in the early 1940's that had radio buttons. He used them to select his favorite stations, as he surfed the airwaves.

(Interestingly, the term radio button has now become a part of programming jargon, signifying certain visual components used in graphical user interfaces.)

Enough of that, now back to my modern car radio

If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set* the frequency value associated with that button to correspond to one of the radio stations in Dallas. *(Again, I would be sending a message to the radio object asking it to change its state.)*

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the *state* of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

NOTE: **Identical objects with identical states:** It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the frequency-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action *(tune to a particular station)* . That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Invoking or calling a method

Java-speak is a little more specific than general OOP-speak. In Java-speak, we might say that pressing one of the selector buttons on the front of the radio invokes or calls a method on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Java is to cause that object to execute one of its methods.

Similarly, we might say that storing a new frequency that corresponds to a particular button invokes a *setter* method on the radio object.

(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)

Behavior

In addition to state, objects are often also said to have *behavior* . The overall behavior of an object is determined by the combined behaviors of its individual methods.

For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers.

Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from?

In order to mass-produce car radios, someone must first create a set of plans, *(drawings, or blueprints)* for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A class definition is a set of plans

The same is true for software objects. In order to create a software object in Java, it is necessary for someone to first create a plan.

In Java, we refer to that plan as a *class* .

The class is defined by a Java programmer. Once the class definition is available, that programmer, *(or other programmers)* , can use it to produce millions of nearly identical objects.

(While millions may sound like a lot of objects, I'm confident that since Java was released into the programming world around 1997, Java programmers around the world have created millions of objects using the standard Java class named ***Button*** *.)*

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a brand new radio and say that it is an instance of the plans used to produce the radio. *(Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.)*

However, it is common jargon to refer to a software object as an instance of a class.

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

A little bit of code

It is time to view a little bit of Java code.

Assuming that you have access to a class definition, there are several different ways that you can create an object in Java. The most common way is using syntax similar to that shown in Listing 1 (p. 616) below.

Listing 1: Instantiating a new Radio object.

```
Radio myObjRef = new Radio();
```

5.1

What does this mean?

Technically, the expression on the right-hand side of the equal sign in Listing 1 (p. 616) applies the new operator to a constructor for the class named **Radio** in order to cause the new object to come into being and to occupy memory.

(Suffice it at this point to say that a constructor is code that assists in the creation of an object according to the plans contained in a class definition. The primary purpose of a constructor is to provide initial values for the new object, but the constructor is not restricted to that behavior alone.)

A reference to the object

The right-hand expression in Listing 1 (p. 616) returns a reference to the new object.

What can you do with a reference?

The reference can later be used to send messages to the new object *(call methods belonging to the new object)* .

Saving the reference

In order to use the reference later, it is necessary to save it for later use.

The expression on the left-hand side of the equal sign in Listing 1 (p. 616) declares a variable of the type **Radio** named **myObjRef** .

(Because this type of variable will ultimately be used to store a reference to an object, we often refer to it by the term ***reference variable*** *.)*

What does this mean?

Declaring a variable causes memory to be set aside for use by the variable. Values can then be stored in that memory space and accessed later by calling up the name given to the variable when it was declared.

Assignment of values

The equal sign in Listing 1 (p. 616) causes the object's reference returned by the right-hand expression to be assigned to, or saved as a value in, the reference variable named **myObjRef** *(created by the left-hand expression)* .

Memory allocation

Once the code in Listing 1 (p. 616) has finished execution, two distinct and different chunks of memory have been allocated and populated.

One *(potentially large)* chunk of memory has been allocated *(by the right-hand expression)* to contain the object itself. This chunk of memory has been populated according to the plans contained in the definition of the class named **Radio** .

The other chunk of memory is a relatively small chunk allocated *(by the left-hand expression)* for the reference variable containing the reference to the object.

Calling a method on the object

Assume that the definition of the Radio class defines a method with the following format *(also assume that this method is intended to simulate pressing a frequency-selector button on the front of the radio)* :

public void playStation(int stationNumber)

What does this mean?

Generally, in our radio-object context, this format implies that the behavior of the method named **playStation** will cause the specific station identified by an integer value passed as **stationNumber** to be selected for play.

Public and void

The *void* return type means that the method doesn't return a value.

The *public* modifier means that the button can be pressed by anyone in the car who can reach it.

(Car radios don't have frequency-selector buttons corresponding to the private modifier in Java.)

The method signature

Continuing with our exposure of jargon, some authors would say that the following constitutes the *method signature* for the method identified above:

playStation(int stationNumber)

A little more Java code

Listing 2 (p. 618) shows the code from the earlier listing, expanded to cause the method named **playStation** to be called.

Listing 2: Calling the playStation method.

```
    Radio myObjRef = new Radio();

myObjRef.playStation(3);
```

5.2

The first statement in Listing 2 (p. 618) is a repeat of the statement from the earlier listing. It is repeated here simply to maintain continuity.

Method invocation syntax

The second statement in Listing 2 (p. 618) is new.

This statement shows the syntax used to send a message to a Java object, or to call a method on that object *(depending on whether you prefer OOP-speak or Java-speak)* .

Join the method name to the reference

The syntax required to call a method on a Java object joins the name of the method to the object's reference, using a period as the joining operator.

(In this case, the object's reference is stored in the reference variable named ***myObjRef*** *. However, there are cases where an object's reference may be created and used in the same expression without storing it in a reference variable. We often refer to such an object as an anonymous object.)*

Pressing a radio button

Given the previous discussion, the numeric value 3, passed to the method when it is called, simulates the pressing of the third button on the front of the radio *(or the fourth button if you elect to number your buttons 0, 1, 2, 3, 4, 5)* .

5.2.1.5 Summary

This is the first in a miniseries of modules that describe and discuss the necessary and most significant *(essential)* aspects of OOP using Java.

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module has concentrated on encapsulation. Encapsulation was used as a springboard for a discussion of objects.

A description of an object-oriented program was provided, along with a description of an object, and how it relates to encapsulation.

In order to relate object-oriented programming to the real world, a car radio was used to illustrate and discuss several aspects of software objects.

You learned that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You learned that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You learned some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You learned where objects come from, and you learned that a class is a set of plans that can be used to construct objects. You learned that a Java object is an instance of a class.

You saw a little bit of Java code, used to create an object, and then to send a message to that object (invoke a method on the object).

You learned about Java references and reference variables. You learned a little about memory allocation for objects and variables in Java.

5.2.1.6 What's next?

The next module in the miniseries will introduce you to the java class.

Continuing with the real-world example introduced in this module, the next module will provide a complete Java program that simulates the manufacture and use of a car radio.

Along the way, you will see examples of (or read about) class definitions, constructing objects, saving references to objects, setter methods, sending messages to objects, instance variables and methods, class variables, array objects, persistence, and objects performing actions.

5.2.1.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Objects and Encapsulation
- File: Java1600.htm
- Published: 12/10/01
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.2.2 Java1602: Classes[20]

5.2.2.1 Table of Contents

[20] This content is available online at <http://cnx.org/content/m44150/1.3/>.

5.2.2.2 Preface

This module is the second in a collection of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.2.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.2.2.1.1 Images

5.2.2.2.1.2 Listings

5.2.2.3 Preview

This module will concentrate primarily on a discussion of the Java class.

A simple Java program will be discussed to illustrate the definition and use of two different classes. Taken in combination, these two classes simulate the manufacture and use of the car radio object discussed in an earlier module.

You will see how to write code to create a new **Radio** object by applying the **new** operator to the class named **Radio** . You will also see how to save that object's reference in a reference variable of type **Radio** .

You will see how to write code that is used to simulate the association of a radio button with a particular radio station.

You will see how to write code that is used to simulate the pressing of a radio button to play the radio station associated with that button.

You will see the definition of a class named **Radio01** . This class consists simply of the **main** method. The **main** method of a Java application is executed by the Java Virtual Machine when the application is run. Thus, it is the driver for the entire application.

You will see the definition of a class named **Radio** . This class includes one instance variable and two instance methods.

(The instance variable is a reference variable that refers to a special kind of object that I refer to as an array object. I will provide a very brief discussion on array objects in this module. I will have more to say about array objects in a subsequent module.)

I will provide a short discussion of class variables, which are not used in this program. I will explain that the use of class variables can often lead to undesirable side effects.

Finally, I will provide a very brief discussion of the syntax of a simple class definition in Java.

5.2.2.4 Discussion and sample code

What is a class?

I explained in an earlier module that a class is a plan from which many objects can be created. I likened the class definition to the plans from which millions of nearly identical car radios can be produced.

A simple Java program

In order to help you to get started on the right foot, and in support of future discussions, it will be advantageous to provide and discuss a simple Java program in this module.

The car radio example

Harking back to an earlier module, Listing 9 (p. 630) , near the end of this module, shows the code for a simple Java application that simulates the manufacture and use of a car radio.

Explain in fragments

In order to help you to focus specifically on important sections of code, I will explain the behavior of this program in fragments.

Top-level classes

This program contains two top-level class definitions. *(Java also supports inner classes as opposed to top-level classes. Inner classes will be explained in detail in subsequent modules in this series.)*

The class named Radio01

One of those class definitions, named **Radio01** , is shown in its entirety in Listing 1 (p. 621) . The other class named **Radio** will be discussed later.

Listing 1: The class named Radio01.

```
    public class Radio01{
  public static void main(
                        String[] args){
    Radio myObjRef = new Radio();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
  }//end main
}//end class Radio01
```

5.3

The class named **Radio01** consists simply of the **main** method. The **main** method of a Java application is executed by the Java Virtual Machine when the application is run. Thus, it is the driver for the entire application.

The driver class

The code in Listing 1 (p. 621) simulates the manufacturer of the radio and the use of the radio by the end user. Without getting into a lot of detail regarding Java syntax, I will further subdivide and discuss this code in the following listings.

Constructing a Radio object

As discussed in a previous module, the code in Listing 2 (p. 622) applies the **new** operator to the constructor for the **Radio** class, causing a new object to be created according to the plans specified in the class named **Radio** .

Listing 2: Constructing a Radio object.

```
Radio myObjRef = new Radio();
```

5.4

Saving a reference to the Radio object

Also as discussed in a previous module, the code in Listing 2 (p. 622) declares a reference variable of type **Radio** and stores the new object's reference in that variable.

Programming the radio buttons

The code in Listing 3 (p. 622) is new to this discussion. This statement simulates the process of associating a particular radio station with a particular button - programming a button on the radio.

As I explained in a previous module, this is accomplished for my car radio by manually tuning the radio to a desired station and then holding the radio button down until it beeps. You have probably done something similar to this to the radio in your car.

Listing 3: Programming the radio buttons.

```
myObjRef.setStationNumber(3, 93.5);
```

5.5

The statement in Listing 3 (p. 622) accomplishes the association of a simulated button to a simulated radio station by calling the method named **setStationNumber** on the reference to the **Radio** object. *(Recall that this sends a message to the object asking it to change its state.)*

The parameters passed to the method cause radio button number 3 to be associated with the frequency 93.5 MHz. *(The value 93.5 is stored in the variable that represents button number 3.)*

Sending a message to the object

Using typical OOP jargon, the statement in Listing 3 (p. 622) sends a message to the **Radio** object, asking it to change its state according to the values passed as parameters.

Pressing a button on the radio

Finally, the code in Listing 4 (p. 623) calls the method named **playStation** on the **Radio** object, passing the integer value 3 *(the button number)* as a parameter.

Listing 4: Pressing a button on the radio.

```
myObjRef.playStation(3);
```

5.6

Another message

This code sends a message to the object asking it to perform an action. In this case, the action requested by the message is:

- Tune yourself to the frequency previously associated with button number 3
- Play the radio station that you find at that frequency through the speakers

How does this simulated radio play?

This simple program doesn't actually play music. As you will see later, this causes the message shown in Image 1 (p. 623) to appear on the computer screen, simulating the selection and playing of a specific radio station.

Image 1: Screen output.

```
Playing the station at 93.5 Mhz
```

5.7

The Radio class

Listing 5 (p. 624) shows the class definition for the **Radio** class in its entirety.

Listing 5: The Radio class .

```
    class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  protected double[] stationNumber =
                        new double[5];

  public void setStationNumber(
                int index,double freq){
    stationNumber[index] = freq;
  }//end method setStationNumber

  public void playStation(int index){
    System.out.println(
            "Playing the station at "
               + stationNumber[index]
               + " Mhz");
  }//end method playStation

}//end class Radio
```

5.8

Note that the code in Listing 5 (p. 624) does not contain an explicit constructor. If you don't define a constructor when you define a new class, a default version of the constructor is provided on your behalf. That is the case for this simple program.

(Constructors will be explained in detail in subsequent modules.)

The plans for an object

The code in Listing 5 (p. 624) provides the plans from which one or more objects that simulate physical radios can be constructed.

An object instantiated *(an object is an instance of a class)* from the code in Listing 5 (p. 624) simulates a physical radio. I will subdivide this code into fragments and discuss it in the following listings.

An instance variable

In a previous module, I explained that we often say that an object is an instance of a class. *(A physical radio is one instance of the plans used to produce it.)* The code in Listing 6 (p. 625) shows the declaration and initialization of what is commonly referred to as an instance variable.

Listing 6: An instance variable.

```
protected double[] stationNumber =
                   new double[5];
```

5.9

Why call it an instance variable?

The name instance variable comes from the fact that every instance of the class *(object)* has one. *(Every radio produced from the same set of plans has the ability to associate a frequency with each selector button on the front of the radio.)*

Class variables - an aside

Note that Java also supports something called a class variable, which is different from an instance variable.

Class variables are shared among all of the objects created from a given class. Stated differently, no matter how many objects are instantiated from a class definition, they all share a single copy of each class variable.

There is no analogy to a class variable in a physical radio object. Radios are installed in different cars separated from each other by thousands of miles. Therefore, there can be no sharing of anything among different physical radio objects.

(Well, that may not be entirely true. In today's technology, different radio objects could potentially share something at a common location via satellite communications, but my car radio doesn't do anything like that.)

Class variables can cause undesirable side effects

While class variables are relatively easy to use in Java, they are difficult to explain from an OOP viewpoint. Also, it is my opinion that from a good overall design viewpoint, class variables should be used very sparingly, if at all.

Therefore, for the first several modules, I will exclude the possibility of class variables in this series of modules. *(I will explain the use of class variables in Java in a subsequent module.)*

Reference to an array object

Now, let's get back to the instance variable named **stationNumber** shown in Listing 6 (p. 625) . Without getting into a lot of detail, this variable is also a reference variable, referring to an array object.

The array object encapsulates a simple one-dimensional array with five elements of type **double** . *(Java array indices begin with zero, so the index values for this array extend from 0 to 4 inclusive. I will also discuss array objects in more detail in a subsequent module.)*

Persistence

The array object is where the data is stored that associates the frequency of a radio station with the simulated physical button on the front of the radio.

Each element in the array corresponds to one frequency-selector button on the front of the radio. Hence, the radio simulated by an object of the **Radio** class has five simulated frequency-selector buttons.

The array object exists when the code in Listing 6 (p. 625) has finished executing. Each element in the array has been initialized to a value of 0.0 *(double-precision float value of zero)* .

The setStationNumber method

Listing 7 (p. 626) shows the setStationNumber method in its entirety

Listing 7: The setStationNumber method.

```
    public void setStationNumber(
              int index,double freq){
  stationNumber[index] = freq;
}//end method setStationNumber
```

5.10

Associates radio station with button

This is the method that is used to simulate the behavior of having the user associate a particular button with a particular radio station. *(Recall that this is accomplished on my car radio by manually tuning the radio to a specific station and then holding the button down until it beeps. Your car radio probably operates in some similar way.)*

This method receives two incoming parameters:

- An integer that corresponds to a button number *(button numbers are assumed to begin with 0 and extend through 4 in order to match array indexes)*
- A frequency value to be associated with the indicated button.

Save the frequency value

The code in the method stores the frequency value in an element of the array object discussed earlier.

The element number is specified by the value of index shown in square brackets in the assignment expression. *(This syntax is similar to storing a value in an array element in most programming languages that I am familiar with.)*

Pressing a radio button to select a station

Listing 8 (p. 626) shows the **playStation** method. This is the method that simulates the result of having the user press a button on the front of the radio to select a particular radio station for play.

Listing 8: The playStation method.

```
    public void playStation(int index){
  System.out.println(
          "Playing the station at "
             + stationNumber[index]
             + " Mhz");
}//end method playStation
```

5.11

Selecting and playing a radio station

The method receives an integer index value as an incoming parameter. This index corresponds to the number of the button pressed by the user. This method simulates the playing of the radio station by

- extracting the appropriate frequency value from the array object, and
- displaying that value on the computer screen along with some surrounding text.

When called by code in the **main** method of this program, this method produces the message shown in Image 1 (p. 623) on the computer screen

That pretty-well summarizes the behavior of this simple program.

Class definition syntax

There are a number of items that can appear in a class definition, including the following:

- Instance variables
- Class variables
- Instance methods
- Class methods
- Constructors
- Static initializer blocks
- Inner classes

Let's keep it simple

In order to make these modules as easy to understand as possible, the first several modules will ignore the possibility of class variables, class methods, static initializer blocks, and inner classes.

As mentioned in the earlier discussion of class variables, these elements aren't particularly difficult to use, but they create a lot of complications when attempting to explain OOP from the viewpoint of Java programming.

Therefore, the first several modules in the series will assume that class definitions are limited to the following elements:

- Instance variables
- Instance methods
- Constructors

A constructor

A constructor is used only once in the lifetime of an object. It participates in the task of creating *(instantiating)* and initializing the object. Following instantiation, the state and behavior of an object depends entirely on instance variables, class variables, instance methods, and class methods.

Instance variables and methods

The class named Radio discussed earlier contains

- one instance variable named **stationNumber** , and
- two instance methods named **setStationNumber** and **playStation** .

5.2.2.5 Summary

This module has concentrated primarily on a discussion of the Java class.

A simple Java program was discussed to illustrate the definition and use of two different classes. Taken in combination, these two classes simulate the manufacture and use of the car radio object introduced in an earlier module.

You saw how to write code to create a new **Radio** object by applying the **new** operator to the class named **Radio** .

You also saw how to save that object's reference in a reference variable of type **Radio** .

You saw how to write code *(in an instance method named **setStationNumber**)* used to simulate the association of a radio button with a particular radio station.

You saw how to write code *(in an instance method named **playStation**)* to simulate the pressing of a radio button to play the radio station associated with that button.

You saw the definition of the class named **Radio01** , which consists simply of the **main** method. The **main** method of a Java application is executed by the Java Virtual Machine when the application is run.

You saw the definition of the class named **Radio** . This class includes one instance variable and two instance methods. *(The instance variable is a reference variable that refers to a special kind of object that I refer to as an array object. I provided a very brief discussion on array objects. I will have more to say on this topic in a subsequent module.)*

I provided a short discussion of class variables, which are not used in this program. I explained that the use of class variables can often lead to undesirable side effects.

Finally, I provided a very brief discussion of the syntax of a simple class definition in Java.

5.2.2.6 What's next?

Recall that in order to understand OOP, you must understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

The next module will begin a discussion of inheritance. Overall, the discussion of inheritance will require more than one module. In the next module, I will discuss how the definition of a class defines a new data type. I will show you how to extend an existing class. I will explain what is inherited through inheritance. I will discuss code reuse and explicit constructors.

Finally, I will illustrate all of the above in a simple program that extends the **Radio** class discussed in this module into a new class named **Combo** that simulates an upgraded radio containing a tape player. *(Yes, at one point in history, car radios did contain tape players.)*

5.2.2.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Classes
- File: Java1602.htm
- Published: 12/24/01
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.2.8 Complete program listing

Listing 9 provides a complete listing of the program named **Radio01** .

Listing 9: The program named Radio01.

```
    /*File Radio01.java
Copyright 2001, R.G.Baldwin
Simulates manufacture and use of a
car radio.

This program produces the following
output on the computer screen:

Playing the station at 93.5 Mhz
************************************/

public class Radio01{
  //This class simulates the
  // manufacturer and the human user
  public static void main(
                        String[] args){
    Radio myObjRef = new Radio();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
  }//end main
}//end class Radio01
  //--------------------------------//

class Radio{
  //This class simulates the plans from
  // which the radio object is created.
  protected double[] stationNumber =
                          new double[5];

  public void setStationNumber(
                int index,double freq){
    stationNumber[index] = freq;
  }//end method setStationNumber

  public void playStation(int index){
    System.out.println(
            "Playing the station at "
               + stationNumber[index]
               + " Mhz");
  }//end method playStation

}//end class Radio
```

5.12

-end-

5.2.3 Java1604: Inheritance, Part 1[21]

5.2.3.1 Table of Contents

5.2.3.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.3.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.3.2.1.1 Images

5.2.3.2.1.2 Listings

[21] This content is available online at <http://cnx.org/content/m44193/1.3/>.

5.2.3.3 Preview

Extending a class

This module shows you how to extend an existing class to create a new class. The new class is the blueprint for a new type.

Inheritance and code reuse

The existing class is often called the *superclass* and the new class is often called the *subclass* . This is the mechanism for class inheritance in Java. Inheritance provides a formal mechanism for code reuse.

The subclass inherits all of the variables and all of the methods defined in the superclass.

Car radios with tape players

A class from a previous module *(whose objects represent car radios)* is extended to define a new class, whose objects represent expanded car radios that contain tape players. *(Yes, at one point in history, car radios did contain tape players instead of CDs.)*

Sending messages to the object

Objects of the new class know how to respond to messages for inserting, playing, and removing a tape, in addition to those messages that are appropriate for objects of the original Radio class.

5.2.3.4 Discussion and sample code

The three pillars of OOP

In an earlier module, I explained that most books on OOP will tell you that in order to understand OOP, you must understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Encapsulation

The first module in this series provided an explanation of encapsulation.

Inheritance

This module *(and some modules to follow)* will provide an explanation of inheritance. I will use another simple program to explain the concept of inheritance.

Polymorphism

Polymorphism is the most complex of the three, and will be explained in future modules.

A new data type

Whenever you define a class in Java, you cause a new data type to become available to the program. Therefore, whenever you need a new data type, you can define a new class to make that type available.

Extending a class

Defining a new class *(to create a new type)* can involve a lot of effort. Sometimes you have an option that can greatly reduce the effort required to create your new type. If a class *(type)* already exists that is close to what you need, you can often extend that class to produce a new class that is closer to what you need.

In many cases, this will require much less effort than that required to start from scratch and define a new class to establish a new type. The ability to extend one class into another new class is the *essence of inheritance* .

According to the current jargon, the new class is called the *subclass* and the class that is extended is called the *superclass* .

What is inherited?

The subclass inherits all of the variables and all of the methods defined in *(or inherited into)* the superclass, almost as if you had completely defined the new class from scratch, and had reproduced all of the code already defined in the existing superclasses.

Code reuse

Therefore, inheritance often makes it possible to define a new class with a minimum requirement to write new code by formally reusing the code that was previously written into the superclasses. Sometimes you can get by with simply extending the existing class.

Sometimes, however, it is also necessary to make changes to the existing class to improve its ability to be extended in a meaningful way. *(That is the case with the sample program discussed in this module, but the next module will show you how to avoid that issue.)* It all depends on how the existing class was designed in the first place.

The Radio class

A previous program defined a class named **Radio** . Objects instantiated from the **Radio** class *(see the previous modules for a discussion of instantiating objects)* were intended to simulate car radios. *(Note that the car radios simulated by objects of the* ***Radio*** *class didn't have built-in tape players.)*

The Combo class

In this module, I will use inheritance to extend the **Radio** class into a new class named **Combo** . Objects instantiated from the **Combo** class are intended to simulate car radios with a built-in tape player.

A complete listing of the new program is shown in Listing 9 (p. 640) near the end of the module.

Will discuss in fragments

As usual, I will discuss this program in fragments. I will begin my discussion with the definition of the new class named **Combo** . Then I will come back and discuss the class named **Radio** and the driver class named **Radio02** .

The combo class

The code in Listing 1 (p. 633) shows the beginning of the class named **Combo** .

Listing 1: Beginning of the Combo class.

```
 class Combo extends Radio{

public Combo(){//constructor
  System.out.println(
         "Combo object constructed");
}//end constructor
```

5.13

Two new items

There are two new items in Listing 1 (p. 633) that you did not see in the code in the previous modules.

Combo extends Radio

First, the class named **Combo** extends the class named **Radio** . This means that an object instantiated from the **Combo** class will contain all of the variables and all the methods defined in the **Combo** class, plus all the variables and methods defined in the **Radio** class, and its superclasses. *(The variables and methods of the superclass are inherited into the subclass.)*

An explicit constructor

Second, the class named **Combo** defines an explicit constructor.

Defining a constructor is optional

When defining a new class, it is not necessary to define a constructor. If you don't define a constructor, a default constructor will be provided automatically.

Why define a constructor?

The intended purpose of a constructor is to initialize the instance variables belonging to the new object. However, constructors can do other things as well. In this case, I used an explicit constructor to display a message when the object is instantiated from the class named **Combo** .

Brief discussion of constructors

I'm not going to discuss constructors in detail at this point. However, I will give you a few rules regarding constructors.

- Constructors *(like methods)* can be overloaded. *(I will explain what overloading means in a subsequent module.)*
- The names of constructors must match the names of the classes in which they are defined.
- A constructor signature never indicates a return type *(such as void or double)* .
- The code in a constructor never contains a return statement.

Instance methods

The new class named **Combo** defines three instance methods, each of which has to do with the handling of tape in the tape player:

- insertTape
- removeTape
- playTape

(If you feel ambitious, you could upgrade this class even further to add features such as rewind, fast forward, pause, etc.).

The insertTape method

The entire method named **insertTape** is shown in Listing 2 (p. 634) . This is the method that is used to simulate the insertion of a tape by the user.

Listing 2: The insertTape method.

```
  public void insertTape(){
System.out.println("Insert Tape");
tapeIn = true
```

5.14

The most significant thing about the code in Listing 2 (p. 634) is the assignment of the **true** value to the **boolean** variable named **tapeIn** . Other than setting the value of the **tapeIn** variable to **true** , the code in Listing 2 (p. 634) simply prints some messages to indicate what is going on.

What is tapeIn used for?

As you will see shortly, the value of the variable named **tapeIn** is used to determine if it is possible to play the tape or to play the radio.

According to that logic:

- If **tapeIn** is true, it is possible to play the tape but it is not possible to play the radio.
- If **tapeIn** is false, it is possible to play the radio, but it is not possible to play the tape.

tapeIn is not declared in the Combo class

It is also worthy of note that in this version of the program, the variable named **tapeIn** is not declared in the **Combo** class *(this will change in the next module where the program uses method overriding)* . Rather, this variable is inherited from the **Radio** class that is extended by the **Combo** class.

The removeTape method

The **removeTape** method of the **Combo** class is shown in Listing 3 (p. 635) . Its behavior is pretty much the reverse of the **insertTape** method, so I won't discuss it further.

Listing 3: The removeTape method.

```
  public void removeTape(){
 System.out.println("Remove Tape");
 tapeIn = false;
 System.out.println(
                  "  Tape is out");
 System.out.println(
                  "  Radio is on");
}//end removeTape method\
```

5.15

The playTape method

Listing 4 (p. 635) shows the method named **playTape** defined in the new **Combo** class.

Listing 4: The playTape method .

```
 public void playTape(){
 System.out.println("Play Tape");
 if(!tapeIn){//tapeIn is false
   System.out.println(
         "  Insert the tape first");
 }else{//tapeIn is true
   System.out.println(
             "  Tape is playing");
 }//end if/else
}//end playTape
```

5.16

Confirm that the tape is ready

Calling the method named **playTape** can be thought of as sending a message to the **Combo** object asking it to play the tape. The code in the **playTape** method checks to confirm that the value stored in the **tapeIn** variable is **true** before executing the request to play the tape.

If **tapeIn** is **false** , an error message is displayed advising the user to insert the tape first.

If **tapeIn** is **true** , the method prints a message indicating that the tape is playing.

Modified Radio class

Listing 5 (p. 636) shows the definition of the modified version of the class named **Radio** .

Listing 5: Modified Radio class.

```
  class Radio{
protected double[] stationNumber =
                       new double[5];
protected boolean tapeIn = false
```

5.17

Tape status

The first significant change that was made to the class named **Radio** is shown in Listing 6 (p. 636) below.

Listing 6: Tape status.

```
protected boolean tapeIn = false;
```

5.18

The statement in Listing 6 (p. 636) declares and initializes a new instance variable named **tapeIn** . As explained earlier, this instance variable is used to indicate whether or not a tape is inserted. *(The Combo class inherits this variable.)*

Earlier in this module, I explained how the **playTape** method of the **Combo** class uses this value to determine whether or not to attempt to play a tape.

Change to the playStation method

The significant change that was made to the method named **playStation** of the **Radio** class is shown in Listing 7 (p. 636) below.

Listing 7: Change to the playStation method.

```
   if(!tapeIn){//tapeIn is false
  System.out.println(
      "  Playing the station at "
          + stationNumber[index]
          + " Mhz");
}else{//tapeIn is true
  System.out.println(
        "  Remove the tape first")
```

5.19

Check the tape status

The code in Listing 7 (p. 636) uses **tapeIn** to check the tape status before attempting to tune the radio station and play the radio. If a tape is inserted, this method simply displays an error message instructing the user to remove the tape first.

So, what's the big deal with inheritance?

The fact that it was necessary for me to make changes to the class named **Radio** greatly reduced the benefit of inheritance in this case. However, even in this case, the use of inheritance eliminated the need for me to define a new class that reproduces all of the code in the class named **Radio** .

(In the next module, I will explain the process of overriding methods. I will show you how to use method overriding to accomplish these same purposes by extending the ***Radio*** *class, without any requirement to modify the code in the* ***Radio*** *class. That will be a much better illustration of the benefits of inheritance.)*

The driver class

The new driver class named **Radio02** is shown in Listing 8 (p. 637) .

Listing 8: The class named Radio02.

```
 public class Radio02{
//This class simulates the
// manufacturer and the human user
public static void main(
                      String[] args){
  Combo myObjRef = new Combo()
```

5.20

New object of the Combo class

The most significant change in this class *(relative to the driver class named* ***Radio01*** *in a previous module)* is the statement that instantiates a new object of the **Combo** class *(instead of the Radio class)* .

All of the other new code in Listing 8 (p. 637) is used to send messages to the new object in order to exercise its behavior.

Program output

The **Combo** object responds to those messages by producing the screen output shown in Image 1 (p. 638) .

Image 1: Program output.

```
    Combo object constructed
Button 3 programmed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
```

5.21

An exercise for the student

As the old saying goes, I will leave it as an exercise for the student to correlate the messages in Listing 8 (p. 637) with the output shown in Image 1 (p. 638) .

5.2.3.5 Summary

Extending an existing class often provides an easy way to create a new type. This is primarily true when an existing class creates a type whose features are close to, but not identical to the features needed in the new type.

When an existing class is extended to define a new class, the existing class is often called the superclass and the new class is often called the subclass.

The subclass inherits all of the variables and all of the methods defined in the superclass and its superclasses.

Inheritance provides a formal mechanism for code reuse.

This module modifies slightly, and then extends the **Radio** class from a previous module to define a new class named **Combo** . Objects of the **Combo** class simulate car radios that contain tape players.

Objects of the **Combo** class know how to respond to messages for inserting, playing, and removing a tape, in addition to those messages appropriate for an object of the **Radio** class.

The changes that were required in the definition of the **Radio** class provide for the fact that it is not possible to play a radio station and to play a tape at the same time. This change was necessary because the original designer of the **Radio** class *(this author)* didn't design that class with the idea of extending it to include a tape player. This points out the importance of thinking ahead when defining a new class.

5.2.3.6 What's next?

In the next module, I will show you how to use *method overriding* to cause the behavior of a method inherited into a subclass to be appropriate for an object instantiated from the subclass.

I will also show you how to use method overriding to eliminate the above requirement to modify the **Radio** class before extending it.

5.2.3.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Inheritance, Part 1
- File: Java1604.htm
- Published: 01/14/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.3.8 Complete program listing

A complete listing of the program is shown in Listing 9 (p. 640) below.

The primary difference between this program and the program in the earlier module *(whose objects simulate car radios)* is the inclusion in this program of a new class named **Combo** . The class named **Combo** extends the original **Radio** class to create a new type of radio that also contains a tape player.

Listing 9: The program named Radio02.

```
    /*File Radio02.java
Copyright 2002, R.G.Baldwin
Simulates the manufacture and use of a
combination car radio and tape player.

This program produces the following
output on the computer screen:

Combo object constructed
Button 3 programmed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
**************************************/

public class Radio02{
  //This class simulates the
  // manufacturer and the human user
  public static void main(
                        String[] args){
    Combo myObjRef = new Combo();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
    myObjRef.insertTape();
    myObjRef.playStation(3);
    myObjRef.removeTape();
    myObjRef.playStation(3);
    myObjRef.playTape();
    myObjRef.insertTape();
    myObjRef.playTape();
    myObjRef.removeTape();
    myObjRef.playStation(3);
  }//end main
}//end class Radio02
```

-end-

5.2.4 Java1606: Inheritance, Part 2[22]

5.2.4.1 Table of Contents

5.2.4.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.4.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.4.2.1.1 Images

5.2.4.2.1.2 Listings

5.2.4.3 Preview

This module builds on the previous module. It is recommended that you study that module before embarking on this module.

The program discussed in this module extends a **Radio** class to produce a new class that simulates an upgraded car radio containing a tape player.

Method overriding is used to modify the behavior of a method of the **Radio** class named **playStation** , to cause that method to behave appropriately when a tape has been inserted into the tape player.

[22] This content is available online at <http://cnx.org/content/m44156/1.3/>.

5.2.4.4 Discussion and sample code

Inheriting methods and variables

When you define a class that extends another class, an object instantiated from your new class will contain all of the methods and all of the variables defined in your new class. The object will also contain all of the methods and all of the variables defined in all of the superclasses of your new class.

The behavior of the methods

The behavior of the methods defined in a superclass and inherited into your new class may, or may not, be appropriate for an object instantiated from your new class. If those methods are appropriate, you can simply leave them alone.

Overriding to change behavior

If the behavior of one or more methods defined in a superclass and inherited into your new class is not appropriate for an object of your new class, you can change that behavior by overriding the method in your new class.

How do you override a method?

To override a method in your new class, simply reproduce the name, argument list, and return type of the original method in a new method definition in your new class. Then provide a body for the new method. Write code in that body to cause the behavior of the overridden method to be appropriate for an object of your new class.

Here is a more precise description of method overriding taken from the excellent book entitled *The Complete Java 2 Certification Study Guide* , by Roberts, Heller, and Ernest:

"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."

Any method that is not declared **final** can be overridden in a subclass.

Overriding versus overloading

Don't confuse method overriding with method overloading. Here is what Roberts, Heller, and Ernest have to say about overloading methods:

"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."

Car radios with built-in tape players

This module presents a sample program that duplicates the functionality of the program named **Radio02** discussed in the previous module. A class named **Radio** is used to define the specifics of objects intended to simulate car radios.

A class named **Combo** extends the **Radio** class to define the specifics of objects intended to simulate improved car radios having built-in tape players.

Modification of the superclass

In the program named **Radio02** in the previous module, it was necessary to modify the superclass before extending it to provide the desired functionality. *(The requirement to modify the superclass before extending it seriously detracts from the benefits of inheritance.)*

No superclass modification in this module

The sample program *(named **Radio03**)* in this module uses method overriding to provide the same functionality as the previous program named **Radio02** , without any requirement to modify the superclass before extending it. *(Thus this program is more representative of the benefits available through inheritance than was the program in the previous module.)*

Overridden playStation method

In particular, a method named **playStation** , defined in the superclass named **Radio** , is overridden in the subclass named **Combo** .

The original version of **playStation** in the superclass supports only radio operations. The overridden version of **playStation** defined in the subclass supports both radio operations and tape operations.

(The behavior of the version of ***playStation*** *defined in the* ***Radio*** *class is not appropriate for an object of the* ***Combo*** *class. Therefore, the method was overridden in the* ***Combo*** *class to cause its behavior to be appropriate for objects instantiated from the* ***Combo*** *class.)*

A complete listing of the program is shown in Listing 5 near the end of this module.

The class named Radio

As usual, I will discuss the program in fragments.

Listing 1 (p. 643) shows the superclass named **Radio** . This code is shown here for easy referral. It is identical to the code for the same class used in the program named **Radio01** discussed in an earlier module.

Listing 1: The class named Radio.

```
  class Radio{
protected double[] stationNumber =
                       new double[5];

public void setStationNumber(
              int index,double freq){
  stationNumber[index] = freq;
}//end method setStationNumber

public void playStation(int index){
  System.out.println(
          "Playing the station at "
             + stationNumber[index]
             + " Mhz");
}//end method playStation
```

5.23

Will override playStation

The class named **Combo** *(discussed below)* will extend the class named **Radio** . The method named **playStation** , shown in Listing 1 (p. 643) , will be overridden in the class named **Combo** .

If you examine the code for the **playStation** method in Listing 1 (p. 643) , you will see that it assumes radio operations only and doesn't support tape operations. That is the reason that it needs to be overridden. *(For example, it doesn't know that it should refuse to play a radio station when a tape is being played.)*

The Combo class

Listing 2 (p. 644) shows the beginning of the class definition for the class named **Combo** . The **Combo** class extends the class named **Radio** .

Listing 2: Beginning of the Combo class.

```
class Combo extends Radio{

private boolean tapeIn = false;
```

5.24

The tapeIn variable

The most important thing about the code in Listing 2 (p. 644) is the declaration of the instance variable named **tapeIn** .

(In the program named Radio02 in the previous module, this variable was declared in the class named ***Radio*** *and inherited into the class named* ***Combo*** *. That was one of the undesirable changes required for the class named* ***Radio*** *in that module.)*

In this version of the program, the variable named **tapeIn** is declared in the subclass instead of in the superclass. Thus, it is not necessary to modify the superclass before extending it.

The constructor

The constructor in Listing 2 (p. 644) is the same as in the previous program named **Radio02** , so I won't discuss it further.

The overridden playStation method

The overridden version of the method named **playStation** is shown in Listing 3 (p. 644) . As you can see, this version of the method duplicates the signature of the **playStation** method in the superclass named **Radio** , but provides a different body.

Listing 3: The overridden playStation method.

```
  public void playStation(int index){
System.out.println("Play Radio");
if(!tapeIn){//tapeIn is false
  System.out.println(
      "  Playing the station at "
           + stationNumber[index]
           + " Mhz");
}else{//tapeIn is true
  System.out.println(
        "  Remove the tape first");
}//end if/else
}//end method playStation
```

5.25

Aware of the tape system

This overridden version of the **playStation** method in Listing 3 (p. 644) is aware of the existence of the tape system and behaves accordingly.

Depending on the value of the variable named **tapeIn**, this method will either

- tune and play a radio station, or
- display a message instructing the user to remove the tape.

Which version of playStation is executed?

When the **playStation** method is called on an object of the **Combo** class, the overridden version of the method *(and not the original version defined in the superclass named **Radio**)* is the version that is actually executed.

Although not particularly obvious in this example, this is one of the important characteristics of *runtime polymorphism*. When a method is called on a reference to an object, it is the type of the object *(and not the type of the variable containing the reference to the object)* that is used to determine which version of the method is actually executed.

Three other instance methods

The subclass named **Combo** defines three other instance methods:

- insertTape
- removeTape
- playTape

The code in these three methods is identical to the code in the methods having the same names in the program named **Radio02** in the previous module. I discussed that code in the previous module and won't repeat that discussion here. You can view those methods in the complete listing of the program shown in Listing 5 (p. 648) near the end of this module.

The driver class

Listing 4 (p. 645) shows the code for the driver class named Radio03.

Listing 4: The driver class.

```
 public class Radio03{
//This class simulates the
// manufacturer and the human user
public static void main(
                     String[] args){
```

The code in Listing 4 (p. 645) is also identical to the code in the program named **Radio02** discussed in the previous module. Therefore, I won't discuss it in detail here.

A new object of the Combo class

I present this code here solely to emphasize that this code instantiates a new object of the **Combo** class. This assures that the overridden version of the method named **playStation** will be executed by the statements in Listing 4 (p. 645) that call the **playStation** method.

*(Although it is not the case in Listing 4 (p. 645), even if the reference to the object of type **Combo** had been stored in a reference variable of type **Radio**, instead of a reference variable of type **Combo**, calling the **playStation** method on that reference would have caused the overridden version of the method to have been executed. That is the essence of runtime polymorphism based on overridden methods in Java.)*

Program output

This program produces the output shown in Image 1 (p. 646) on the computer screen.

Image 1: Program output.

```
    Combo object constructed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
```

5.27

I will leave it as an exercise for the student to compare this output with the messages sent to the object by the code in Listing 4 (p. 645) .

5.2.4.5 Summary

An object instantiated from a class that extends another class will contain all of the methods and all of the variables defined in the subclass, plus all of the methods and all of the variables inherited into the subclass.

The behavior of methods inherited into the subclass may not be appropriate for an object instantiated from the subclass. You can change that behavior by overriding the method in the definition of the subclass.

To override a method in the subclass, reproduce the name, argument list, and return type of the original method in a new method definition in the subclass. Make sure that the overridden method is not less accessible than the original method. Also, make sure that it doesn't throw any checked exceptions that were not declared for the original method.

Provide a body for the overridden method, causing the behavior of the overridden method to be appropriate for an object of the subclass. Any method that is not declared **final** can be overridden in a subclass.

The program discussed in this module extends a **Radio** class to produce a subclass that simulates an upgraded car radio containing a tape player.

Method overriding is used to modify the behavior of an inherited method named **playStation** to cause that method to behave appropriately when a tape has been inserted into the radio.

Method *overriding* is different from method overloading. Method *overloading* will be discussed in the next module.

5.2.4.6 What's next?

In the next module, I will explain the use of overloaded methods for the purpose of achieving compile-time polymorphism.

5.2.4.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Inheritance, Part 2
- File: Java1606.htm
- Published: 01/28/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.4.8 Complete program listing

A complete listing of the program is shown in Listing 5 (p. 648) below.

Listing 5: The program named Radio03.

```
    Copyright 2002, R.G.Baldwin
Simulates the manufacture and use of a
combination car radio and tape player.
Uses method overriding to avoid
modifying the class named Radio.

This program produces the following
output on the computer screen:

Combo object constructed
Play Radio
  Playing the station at 93.5 Mhz
Insert Tape
  Tape is in
  Radio is off
Play Radio
  Remove the tape first
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
Play Tape
  Insert the tape first
Insert Tape
  Tape is in
  Radio is off
Play Tape
  Tape is playing
Remove Tape
  Tape is out
  Radio is on
Play Radio
  Playing the station at 93.5 Mhz
**************************************/

public class Radio03{
  //This class simulates the
  // manufacturer and the human user
  public static void main(
                        String[] args){
    Combo myObjRef = new Combo();
    myObjRef.setStationNumber(3,93.5);
    myObjRef.playStation(3);
    myObjRef.insertTape();
    myObjRef.playStation(3);
    myObjRef.removeTape();
    myObjRef.playStation(3);
    myObjRef.playTape();
    myObjRef.insertTape();
    myObjRef.playTape();
    myObjRef.removeTape();
    myObjRef.playStation(3);
  }//end main
}//end class Radio03
```

-end-

5.2.5 Java1608: Polymorphism Based on Overloaded Methods[23]

5.2.5.1 Table of Contents

5.2.5.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.5.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.5.2.1.1 Listings

5.2.5.3 Preview

Previous modules introduced *overloading* and *overriding* methods. This module concentrates on the use of method overloading to achieve *compile-time polymorphism* .

Every class in Java is a direct or indirect subclass of the class named **Object** . Methods defined in the class named **Object** are inherited into all other classes. Inherited methods that are not declared **final** may be overridden to make their behavior more appropriate to objects instantiated from the new class.

Overloaded methods have the same name and different formal argument lists. They may or may not have the same return type.

Polymorphism manifests itself in Java in the form of multiple methods having the same name. This module concentrates on method overloading, sometimes referred to as *compile-time polymorphism* . Subsequent modules concentrate on method overriding, sometimes referred to as *runtime polymorphism* .

Overloaded methods may all be defined in the same class, or may be defined in different classes as long as those classes have a superclass-subclass relationship.

[23] This content is available online at <http://cnx.org/content/m44182/1.3/>.

5.2.5.4 Discussion and sample code

Three concepts

In an earlier module, I explained that most books on OOP will tell you that in order to understand OOP, you must understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Encapsulation and inheritance

Previous modules in this series have explained Encapsulation and Inheritance. This module will tackle the somewhat more complex topic of Polymorphism.

Overloading and overriding methods

In the modules on inheritance, you learned a little about overloading and overriding methods *(you will learn more about these concepts as you progress through these modules)* . This module concentrates on the use of overloaded methods to achieve compile-time polymorphism.

Real-world scenarios

The sample programs that I used in the previous modules in this series dealt with two kinds of car radios:

- Plain car radios
- Car radios having built-in tape players

I couched those programs in a real-world scenario in an attempt to convince you that encapsulation and inheritance really do have a place in the real world.

Programs were fairly long

However, even though those programs were simple in concept, they were relatively long. That made them somewhat difficult to explain due simply to the amount of code involved.

Keep it short and simple

Beginning with this module, I am going to back away from real-world scenarios and begin using sample programs that are as short and as simple as I know how to make them, while still illustrating the important points under discussion.

My objective in this and future modules is to make the polymorphic concepts as clear as possible without having those concepts clouded by other programming issues.

I will simply ask you to trust me when I tell you that polymorphism has enormous applicability in the real world.

A little more on inheritance

There is another aspect of inheritance that I didn't explain in the previous modules.

Every class extends some other class

Every class in Java extends some other class. If you don't explicitly specify the class that your new class extends, it will automatically extend the class named **Object** .

A class hierarchy

Thus, all classes in Java exist in a class hierarchy where the class named **Object** forms the root of the hierarchy.

Some classes extend **Object** directly, while other classes are subclasses of Object further down the hierarchy.

Methods in the Object class

The class named **Object** defines default versions of the following methods:

- clone()
- equals(Object obj)
- finalize()

- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- **wait()**
- **wait(long timeout)**
- **wait(long timeout, int nanos)**

As you can see, this list includes three overloaded versions of the method named **wait** . The three versions have the same name but different formal argument lists. Thus, these three methods are *overloaded* versions of the method name **wait** .

Every class inherits these eleven methods

Because every class is either a direct or indirect subclass of **Object** , every class in Java, *(including new classes that you define)* , inherit these eleven methods.

To be overridden ...

Some of these eleven methods are intended to be overridden for various purposes. However, some of them, such as **getClass** , **notify** , and the three versions of **wait** , are intended to be used directly without overriding. *(Although not shown here, these five methods are declared to be* ***final*** *, meaning that they may not be overridden.)*

What is polymorphism?

The meaning of the word polymorphism is something like one name, many forms.

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists *(overloaded methods)* . In other cases, multiple methods have the same name, same return type, and same formal argument list *(overridden methods)* .

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Method overloading

I will begin the discussion of polymorphism with method overloading, which is the simplest of the three. I will cover method overloading in this module and will cover polymorphism based on overridden methods and interfaces in subsequent modules.

Method overloading versus method overriding

Don't confuse method *overloading* with method *overriding* .

Java allows you to have two or more method definitions in the same scope with the same name, provided that they have different formal argument lists.

More specifically, here is what Roberts, Heller, and Ernest have to say about overloading methods in their excellent book titled **The Complete Java 2 Certification Study Guide** :

"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."

Similarly, as a preview of things to come, here is what they have to say about method overriding:

"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."

You should read these two descriptions carefully and make certain that you recognize the differences.

Compile-time polymorphism

Some authors refer to method overloading as a form of *compile-time polymorphism* , as distinguished from *run-time polymorphism* .

This distinction comes from the fact that, for each method call, the compiler determines which method *(from a group of overloaded methods)* will be executed, and this decision is made when the program is compiled. *(In contrast, I will tell you later that the determination of which overridden method to execute isn't made until runtime.)*

Selection based on the argument list

In practice, the compiler simply examines the types, number, and order of the parameters being passed in a method call, and selects the overloaded method having a matching formal argument list.

A sample program

I will discuss a sample program named **Poly01** to illustrate method overloading. A complete listing of the program can be viewed in Listing 4 (p. 656) near the end of the module.

Within the class and the hierarchy

Method overloading can occur both within a class definition, and vertically within the class inheritance hierarchy. *(In other words, an overloaded method can be inherited into a class that defines other overloaded versions of the method.)* The program named **Poly01** illustrates both aspects of method overloading.

Class B extends class A, which extends Object

Upon examination of the program, you will see that the class named **A** extends the class named **Object** . You will also see that the class named **B** extends the class named **A** .

The class named **Poly01** is a driver class whose **main** method exercises the methods defined in the classes named **A** and **B** .

Once again, this program is not intended to correspond to any particular real-world scenario. Rather, it is a very simple program designed specifically to illustrate method overloading.

Will discuss in fragments

As is my usual approach, I will discuss this program in fragments. The code in Listing 1 (p. 652) defines the class named **A** , which explicitly extends **Object** .

Listing 1: Definition of the class named A.

```
    class A extends Object{
  public void m(){
    System.out.println("m()");
  }//end method m()
}//end class A
```

5.29

Redundant code

Explicitly extending **Object** is not required *(but it also doesn't hurt anything to do it)* .

By default, the class named **A** would extend the class named **Object** automatically, unless the class named **A** explicitly extends some other class.

The method named m()

The code in Listing 1 (p. 652) defines a method named **m()** . Note that this version of the method has an empty argument list *(it doesn't receive any parameters when it is executed)* . The behavior of the method is simply to display a message indicating that it has been called.

The class named B

Listing 2 (p. 653) contains the definition for the class named **B** . The class named **B** extends the class named **A** , and inherits the method named **m** defined in the class named **A** .

Listing 2: Definition of the class named B.

```
    class B extends A{
  public void m(int x){
    System.out.println("m(int x)");
  }//end method m(int x)
  //---------------------------------//

  public void m(String y){
    System.out.println("m(String y)");
  }//end method m(String y)
}//end class B
```

5.30

Overloaded methods

In addition to the inherited method named **m** , the class named **B** defines two overloaded versions of the method named **m** :

- m(int x)
- m(String y)

(Note that each of these two versions of the method receives a single parameter, and the type of the parameter is different in each case.)

As with the version of the method having the same name defined in the class named **A** , the behavior of each of these two methods is to display a message indicating that it has been called.

The driver class

Listing 3 (p. 653) contains the definition of the driver class named **Poly01** .

Listing 3: Definition of the driver class named Poly01.

```
    public class Poly01{
  public static void main(String[] args){
    B var = new B();
    var.m();
    var.m(3);
    var.m("String");
  }//end main
}//end class Poly01
```

5.31

Call all three overloaded methods

The code in the **main** method

- Instantiates a new object of the class named **B** , and
- Successively calls each of the three overloaded versions of the method named **m** on the reference to that object.

One version is inherited

The overloaded version of the method named **m** , defined in the class named **A** , is inherited into the class named **B** . Therefore, it can be called on a reference to an object instantiated from the class named **B** .

Two versions defined in class B

The other two versions of the method named **m** are defined in the class named **B** . Thus, they also can be called on a reference to an object instantiated from the class named **B** .

The output

As you would expect, the output produced by sending messages to the object asking it to execute each of the three overloaded versions of the method named m is:

```
m()
m(int x)
m(String y)
```

Note that the values of the parameters passed to the methods do not appear in the output. Rather, in this simple example, the parameters are used solely to make it possible for the compiler to select the correct version of the overloaded method to execute.

This output confirms that each overloaded version of the method is properly selected for execution based on the matching of method parameters to the formal argument list of each method.

5.2.5.5 Summary

Previous modules introduced *overloading* and *overriding* methods. This module concentrates on the use of method *overloading* to achieve *compile-time polymorphism* .

All classes in Java form a hierarchy with a class named **Object** at the root of the hierarchy. Thus, every class in Java is a direct or indirect subclass of the class named **Object** .

If a new class doesn't explicitly extend some other class, it will, by default, automatically extend the class named **Object** .

The **Object** class defines default versions of eleven different methods. These methods are inherited into all other classes, and some *(those not declared **final**)* may be overridden to make their behavior more appropriate for objects instantiated from the new class.

Overloaded methods have the same name and different formal argument lists. They may or may not have the same return type.

Three of the eleven methods defined in the class named **Object** are overloaded versions of the method name **wait** . One version takes no parameters. A second version takes a single parameter of type **long** . The third version takes two parameters, one of type **long** , and one of type **int** .

The word *polymorphism* means something like *one name, many forms* . Polymorphism manifests itself in Java in the form of multiple methods having the same name.

Polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

This module concentrates on method *overloading* , sometimes referred to as *compile-time polymorphism* . This form of polymorphism is distinguished by the fact that the compiler selects a specific method from two or more overloaded methods on the basis of the types and the number of parameters passed to the method when it is called. The selection is made when the program is compiled *(rather than being made later when the program is run)* .

Overloaded methods may all be defined in the same class, or may be defined in different classes as long as those classes have a superclass-subclass relationship in the class hierarchy.

The sample program in this module illustrates three overloaded versions of the same method name with two of the versions being defined in a single class, and the other version being defined in the superclass of that class.

5.2.5.6 What's next?

The next module in this collection teaches you about assignment compatibility, type conversion, and casting for both primitive and reference types.

It also teaches you about the relationship between reference types, method calls, and the location in the class hierarchy where a method is defined.

5.2.5.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism Based on Overloaded Methods
- File: Java1608.htm
- Published: 02/11/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.5.8 Complete program listings

A complete listing of the program is shown in Listing 4 (p. 656) below.

Listing 4: Complete program listing.

```
    /*File Poly01.java
Copyright 2002, R.G.Baldwin

Program output is:
m()
m(int x)
m(String y)
************************************/

class A extends Object{
  public void m(){
    System.out.println("m()");
  }//end method m()
}//end class A
//===================================//

class B extends A{
  public void m(int x){
    System.out.println("m(int x)");
  }//end method m(int x)
  //-------------------------------//

  public void m(String y){
    System.out.println("m(String y)");
  }//end method m(String y)
}//end class B
//===================================//

public class Poly01{
  public static void main(String[] args){
    B var = new B();
    var.m();
    var.m(3);
    var.m("String");
  }//end main
}//end class Poly01
```

5.32

-end-

5.2.6 Java1610: Polymorphism, Type Conversion, Casting, etc.[24]

5.2.6.1 Table of Contents

5.2.6.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.6.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.6.2.1.1 Listings

5.2.6.3 Preview

This module discusses type conversion for both *primitive* and *reference* types.

A value of a particular type may be *assignment compatible* with variables of other types, in which case the value can be assigned directly to the variable. Otherwise, it may be possible to perform a *cast* on the value to change its type and assign it to the variable as the new type.

With regard to reference types, whether or not a cast can be successfully performed

- depends on the relationships of the classes involved in the class hierarchy.

[24] This content is available online at <http://cnx.org/content/m44168/1.3/>.

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a *downcast* .

Whether or not a method can be called on a reference to an object depends on

- the current type of the reference, and
- the location in the class hierarchy where the method is defined.

In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

A sample program is provided that illustrates much of the detail involved in type conversion, method calls, and casting with respect to reference types.

5.2.6.4 Discussion and sample code

What is polymorphism?

As a quick review, the meaning of the word *polymorphism* is something like *one name, many forms* .

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists *(overloaded methods, which were discussed in a previous module)* .

In other cases, multiple methods have the same name, same return type, and same formal argument list *(overridden methods)* .

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

I covered method overloading as one form of polymorphism in a previous module.

We need to backtrack

In this module, I will backtrack a bit and discuss the conversion of references from one type to another.

I will begin the discussion of polymorphism through method overriding and inheritance in the next module. I will cover interfaces in a future module.

Assignment compatibility and type conversion

As a background for polymorphism, you need to understand something about *assignment compatibility* and *type conversion* .

A value of a given type is assignment compatible with another type if

- a value of the first type
- can be successfully assigned to a variable of the second type.

Type conversion and the cast operator

In some cases, type conversion happens automatically. In other cases, type conversion must be forced through the use of a *cast operator* .

A cast operator is a *unary* operator, which has a single right operand. The physical representation of the cast operator is the name of a type inside a pair of matched parentheses, as in:

```
(int)
```

Applying a cast operator

Applying a cast operator to the name of a variable doesn't actually change the type of the variable. However, it does cause the contents of the variable to be treated as a different type for the evaluation of the expression in which the cast operator is contained. Thus, the application of a cast operator is a short-term event.

Primitive values and type conversion

Assignment compatibility issues come into play for both primitive types and reference types.

However, values of type **boolean** can only be assigned to variables of type boolean *(you cannot change the type of a boolean)* .

Otherwise, a primitive value can be assigned to any variable of a type

- whose range is as wide or wider
- than the range of the type of the value.

In that case, the type of the value is automatically converted to the type of the variable.

(For example, types ***byte*** *and* ***short*** *can be assigned to a variable of type* ***int*** *without the requirement for a cast because type* ***int*** *has a wider range than either type* ***byte*** *or type* ***short*** *.)*

Conversion to narrower range

On the other hand, a primitive value of a given type cannot be assigned to a variable of a type with a narrower range than the type of the value,

- unless the cast operator is used to force a type conversion.

Oftentimes, such a conversion will result in the loss of data, and that loss is the responsibility of the programmer who performs the cast.

Assignment compatibility for references

Assignment compatibility, with respect to references, doesn't involve range issues, as is the case with primitives. Instead, the reference to an object instantiated from a given class can be assigned to:

- **Any reference variable** whose type is the same as the class from which the object was instantiated.

- Any reference variable whose type is a superclass of the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated, and
- A few other cases involving the class and interface hierarchy.

Such an assignment does not require the use of a cast operator.

Type Object is completely generic

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

Converting reference types with a cast

Assignments of references, other than those listed above (p. 659) , require the use of a cast operator to purposely change the type of the reference.

Doesn't work in all cases

However, it is not possible to perform a successful cast to convert the type of a reference in all cases.

Generally, a cast can only be performed among reference types that fall on the same ancestral line of the class hierarchy, or on an ancestral line of an interface hierarchy. For example, a reference cannot be successfully cast to the type of a sibling or a cousin in the class hierarchy.

Downcasting

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a *downcast* .

While it is also possible to cast in the direction from the leaves to the root, this happens automatically, and the use of a cast operator is not required.

A sample program

The program named **Poly02** , shown in Listing 11 (p. 666) near the end of the module, illustrates the use of the cast operator with references.

When you examine that program, you will see that two classes named **A** and **C** each extend the class named **Object** . Hence, we might say that they are siblings in the class hierarchy.

Another class named **B** extends the class named **A** . Thus, we might say that **A** is a child of **Object** , and **B** is a child of **A** .

The class named A

The definition of the class named A is shown in Listing 1 (p. 660) . This class extends the class named **Object** .

*(Recall that it is not necessary to explicitly state that a class extends the class named **Object** . Any class that does not explicitly extend some other class will automatically extend **Object** by default. The class named **A** is shown to extend **Object** here simply for clarity of presentation.)*

Listing 1: Definition of the class named A.

```
 class A extends Object{
  //this class is empty
}//end class A
```

The class named **A** is empty. It was included in this example for the sole purpose of adding a layer of inheritance to the class hierarchy.

The class named B

Listing 2 (p. 660) shows the definition of the class named **B** . This class extends the class named **A** .

Listing 2: Definition of the class named B.

```
 class B extends A{
  public void m(){
   System.out.println("m in class B");
  }//end method m()
}//end class B
```

The method named m()

The class named **B** defines a method named **m()** . The behavior of the method is simply to display a message each time it is called.

The class named C

Listing 3 (p. 661) contains the definition of the class named **C**, which also extends **Object**.

Listing 3: Definition of the class named C.

```
 class C extends Object{
//this class is empty
}//end class C
```

5.35

The class named **C** is also empty. It was included in this example as a sibling class for the class named **A**. Stated differently, it was included as a class that is not in the ancestral line of the class named **B**.

The driver class

Listing 4 (p. 661) shows the beginning of the driver class named **Poly02**.

Listing 4: Beginning of the class named Poly02.

```
 public class Poly02{
public static void main(String[] args){
 Object var = new B();
```

5.36

An object of the class named B

The code in Listing 4 (p. 661) instantiates an object of the class **B** and assigns the object's reference to a reference variable of type **Object**.

*(It is important to note that the reference to the object of type **B** was not assigned to a reference variable of type **B**. Instead, it was assigned to a reference variable of type **Object**.)*

This assignment is allowable because **Object** is a superclass of **B**. In other words, the reference to the object of the class **B** is assignment compatible with a reference variable of the type **Object**.

Automatic type conversion

In this case, the reference of type **B** is automatically converted to type **Object** and assigned to the reference variable of type **Object**. *(Note that the use of a cast operator was not required in this assignment.)*

Only part of the story

However, assignment compatibility is only part of the story. The simple fact that a reference is assignment compatible with a reference variable of a given type says nothing about what can be done with the reference after it is assigned to the reference variable.

An illegal operation

For example, in this case, the reference variable that was automatically converted to type **Object** cannot be used directly to call the method named **m()** on the object of type **B**. This is indicated in Listing 5 (p. 662).

Listing 5: An illegal operation.

```
//var.m();
```

5.37

An attempt to call the method named **m()** on the reference variable of type **Object** in Listing 5 (p. 662) resulted in a compiler error. It was necessary to convert the statement into a comment in order to get the program to compile successfully.

An important rule

In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

This case violates the rule

In this case, the method named **m()** is defined in the class named **B** , which is two levels down from the class named **Object** .

When the reference to the object of the class **B** was assigned to the reference variable of type **Object** , the type of the reference was automatically converted to type **Object** .

Therefore, because the reference is of type **Object** , it cannot be used directly to call the method named **m()** .

The solution is a downcast

In this case, the solution to the problem is a downcast. The code in Listing 6 (p. 662) shows an attempt to solve the problem by casting the reference down the hierarchy to type **A** .

Listing 6: An ineffective downcast.

```
//((A)var).m();
```

5.38

Still doesn't solve the problem

However, this still doesn't solve the problem, and the result is another compiler error. Again, it was necessary to convert the statement into a comment in order to get the program to compile.

What is the problem here?

The problem is that the downcast simply didn't go far enough down the inheritance hierarchy.

The class named **A** neither defines nor inherits the method named **m()** . The method named **m()** is defined in class **B** , which is a subclass of class **A** .

Therefore, a reference of type **A** is no more useful than a reference of type **Object** insofar as calling the method named **m()** is concerned.

The real solution

The solution to the problem is shown in Listing 7 (p. 663) .

Listing 7: A downcast to type B.

```
((B)var).m();
```

5.39

The code in Listing 7 (p. 663) casts *(converts)* the reference value contained in the **Object** variable named **var** down to type **B** .

The method named **m()** is defined in the class named **B** . Therefore, a reference of type **B** can be used to call the method.

The code in Listing 7 (p. 663) compiles and executes successfully. This causes the method named **m()** to execute, producing the following output on the computer screen.

```
m in class B
```

A few odds and ends

Before leaving this topic, let's look at a couple more issues. The code in Listing 8 (p. 663) declares and populates a new variable of type **B** .

Listing 8: Declare a variable of type B.

```
B v1 = (B)var;
```

5.40

The code in Listing 8 also uses a cast to:

- Convert the contents of the **Object** variable to type **B**
- **Assign the converted reference to the new reference variable of type B.**

A legal operation

This is a legal operation. In this class hierarchy, the reference to the object of the class **B** can be assigned to a reference variable of the types **B** , **A** , or **Object** .

Cannot be assigned to type C

However, the reference to the object of the class **B** cannot be assigned to a reference variable of any other type, including the type **C** . An attempt to do so is shown in Listing 9 (p. 663) .

Listing 9: Cannot be assigned to type C.

```
//C v2 = (C)var;
```

5.41

The code in Listing 9 (p. 663) attempts to cast the reference to type **C** and assign it to a reference variable of type **C** .

A runtime error

Although the program will compile, it won't execute. An attempt to execute the statement in Listing 9 (p. 663) results in a **ClassCastException** at runtime. As a result, it was necessary to convert the statement into a comment in order to execute the program.

Another failed attempt

Similarly, an attempt to cast the reference to type B and assign it to a reference variable of type **C** , as shown in Listing 10 (p. 664) , won't compile.

Listing 10: Another failed attempt.

```
//C v3 = (B)var;
```

5.42

The problem here is that the class **C** is not a superclass of the class named **B** . Therefore, a reference of type **B** is not assignment compatible with a reference variable of type **C** .

Again, it was necessary to convert the statement into a comment in order to compile the program.

5.2.6.5 Summary

This module discusses type conversion for both primitive and reference types.

A value of a particular type may be assignment compatible with variables of other types.

If the type of a value is not assignment compatible with a variable of a given type, it may be possible to perform a cast on the value to change its type and assign it to the variable as the new type. For primitive types, this will often result in the loss of information.

Except for type **boolean** , values of primitive types can be assigned to any variable whose type represents a range that is as wide or wider than the range of the value's type. *(Values of type* ***boolean*** *can only be assigned to variables of type boolean.)*

With respect to reference types, the reference to an object instantiated from a given class can be assigned to any of the following without the use of a cast:

- Any reference variable whose type is the same as the class from which the object was instantiated.
- Any reference variable whose type is a superclass of the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated.
- A few other cases involving the class and interface hierarchy.

Assignments of references, other than those listed above, require the use of a cast to change the type of the reference.

It is not always possible to perform a successful cast to convert the type of a reference. Whether or not a cast can be successfully performed depends on the relationship of the classes involved in the class hierarchy.

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

When we cast a reference along the class hierarchy in a direction from the root class **Object** toward the leaves, we often refer to it as a downcast.

Whether or not a method can be called on a reference to an object depends on the current type of the reference and the location in the class hierarchy where the method is defined. In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

A sample program is provided that illustrates much of the detail involved in type conversion, method invocation, and casting with respect to reference types.

5.2.6.6 What's next?

I will begin the discussion of runtime polymorphism through method overriding and inheritance in the next module.

I will demonstrate that for runtime polymorphism, the selection of a method for execution is based on the actual type of object whose reference is stored in a reference variable, and not on the type of the reference variable on which the method is called.

5.2.6.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism, Type Conversion, Casting, etc.
- File: Java1610.htm
- Published: 02/26/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.6.8 Complete program listings

A complete listing of the program is shown in Listing 11 (p. 666) below.

Listing 11: Complete program listing.

```
    /*File Poly02.java
Copyright 2002, R.G.Baldwin

This program illustrates downcasting

Program output is:

m in class B
**************************************/

class A extends Object{
  //this class is empty
}//end class A
//===================================//

class B extends A{
  public void m(){
    System.out.println("m in class B");
  }//end method m()
}//end class B
//===================================//

class C extends Object{
  //this class is empty
}//end class C
//===================================//

public class Poly02{
  public static void main(String[] args){
    Object var = new B();
    //Following will not compile
    //var.m();
    //Following will not compile
    //((A)var).m();
    //Following will compile and run
    ((B)var).m();

    //Following will compile and run
    B v1 = (B)var;
    //Following will not execute
    //C v2 = (C)var;
    //Following will not compile
    //C v3 = (B)var;
  }//end main
}//end class Poly02
```

5.43

-end-

5.2.7 Java1612: Runtime Polymorphism through Inheritance[25]

5.2.7.1 Table of Contents

5.2.7.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.7.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.7.2.1.1 Listings

5.2.7.3 Preview

What is polymorphism?

The meaning of the word *polymorphism* is something like *one name, many forms* .

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists *(overloaded methods, which were discussed in a previous module)* .

In other cases, multiple methods have the same name, same return type, and same formal argument list *(overridden methods)* .

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading

[25] This content is available online at <http://cnx.org/content/m44177/1.3/>.

- Method overriding through class inheritance
- Method overriding through the Java interface

I covered method overloading as one form of polymorphism *(compile-time polymorphism)* in a previous module. I also explained automatic type conversion and the use of the cast operator for type conversion in a previous module.

In this module ...

I will begin the discussion of runtime polymorphism through method overriding and class inheritance in this module. I will cover interfaces in a future module.

The essence of runtime polymorphic behavior

With runtime polymorphism based on method overriding,

- the decision as to which version of a method will be executed is based on
- the actual type of the object whose reference is stored in the reference variable, and
- **not** on the type of the reference variable on which the method is called.

Late binding

The decision as to which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as *late binding* .

5.2.7.4 Discussion and sample code

Operational description of runtime polymorphism

Here is an operational description of runtime polymorphism as implemented in Java through class inheritance and method overriding:

- Assume that a class named **SuperClass** defines a method named **method** .
- Assume that a class named **SubClass** extends **SuperClass** and overrides the method named **method** .
- Assume that a reference to an object of the class named **SubClass** is assigned to a reference variable named **ref** of type **SuperClass** .
- Assume that the method named **method** is then called on the reference variable using the following syntax:
  - **ref.method()**
- ***Result:*** The version of the method named **method** that will actually be executed is the *overridden* version in the class named **SubClass** , and is not the version that is defined in the class named **SuperClass,** even though the reference to the object of type **SubClass** is stored in a variable of type **SuperClass** .

This is runtime polymorphism in a nutshell, which is sometimes also referred to as late-binding.

Runtime polymorphism is very powerful

As you gain more experience with Java, you will learn that much of the power of OOP using Java is centered on runtime polymorphism using class inheritance, interfaces, and method overriding. *(The use of interfaces for polymorphism will be discussed in a future module.)*

An important attribute of runtime polymorphism

The decision as to which version of the method to execute

- is based on the actual type of object whose reference is stored in the reference variable, and
- not on the type of the reference variable on which the method is called.

Why is it called runtime polymorphism?

The reason that this type of polymorphism is often referred to as runtime polymorphism is because the decision as to which version of the method to execute cannot be made until runtime. The decision cannot be made at compile time.

Why defer the decision?

The decision cannot be made at compile time because the compiler has no way of knowing *(when the program is compiled)* the actual type of the object whose reference will be stored in the reference variable .

In an extreme case, for example, the object might be de-serialized at runtime from a network connection of which the compiler has no knowledge.

Could be either type

For the situation described above, that de-serialized object could just as easily be of type **SuperClass** as of type **SubClass** . In either case, it would be valid to assign the object's reference to the same superclass reference variable.

If the object were of the **SuperClass** type, then a call to the method named **method** on the reference would cause the version of the method defined in **SuperClass** , and not the version defined in **SubClass** , to be executed. *(The version executed is determined by the type of the object and not by the type of the reference variable containing the reference to the object.)*

Sample Program

Let's take a look at a sample program that illustrates runtime polymorphism using class inheritance and overridden methods. The name of the program is **Poly03** . A complete listing of the program is shown in Listing 7 (p. 675) near the end of the module.

Listing 1 (p. 669) shows the definition of a class named **A** , which extends the class named **Object** .

(Remember that any class that doesn't extend some other class automatically extends ***Object*** *by default, and it is not necessary to show that explicitly as I did in this example.)*

Listing 1: Definition of the class named A.

```
    class A extends Object{
  public void m(){
    System.out.println("m in class A");
  }//end method m()
}//end class A
```

5.44

The class named **A** defines a method named **m()** .

Behavior of the method

The behavior of the method, as defined in the class named **A** , is to display a message indicating that it has been called, and that it is defined in the class named **A** .

This message will allow us to determine which version of the method is executed in each case discussed later.

The class named B

Listing 2 (p. 670) shows the definition of a class named **B** that extends the class named **A** .

Listing 2: Definition of the class named B.

```
    class B extends A{
  public void m(){
    System.out.println("m in class B");
  }//end method m()
}//end class B
```

5.45

The class named **B** overrides *(redefines)* the method named **m()** , which it inherits from the class named **A** .

Behavior of the overridden version of the method

Like the inherited version, the overridden version displays a message indicating that it has been called. However, the message is different from the message displayed by the inherited version discussed above. The overridden version tells us that it is defined in the class named **B** . *(The behavior of the overridden version of the method is appropriate for an object instantiated from the class named **B** .)*

Again, this message will allow us to determine which version of the method is executed in each case discussed later.

The driver class

Listing 3 (p. 670) shows the beginning of the driver class named **Poly03** .

Listing 3: Beginning of the driver class named Poly03.

```
    public class Poly03{
  public static void main(String[] args){
    Object var = new B();
    ((B)var).m();
```

5.46

A new object of the class B

The code in the **main** method begins by instantiating a new object of the class named **B** , and assigning the object's reference to a reference variable of type **Object** .

*(Recall that this is legal because an object's reference can be assigned to any reference variable whose type is a superclass of the class from which the object was instantiated. The class named **Object** is the superclass of all classes.)*

Downcast and call the method

If you read the earlier module on casting, it will come as no surprise to you that the second statement in the **main** method, which casts the reference down to type **B** and calls the method named **m()** on it, will compile and execute successfully.

Which version is executed?

The execution of the method produces the following output on the computer screen:

```
m in class B
```

By examining the output, you can confirm that the version of the method that was overridden in the class named **B** is the version that was executed.

Why was this version executed?

This should also come as no surprise to you. The cast converts the type of the reference from type **Object** to type **B** .

You can always call a public method belonging to an object using a reference to the object whose type is the same as the class from which the object was instantiated.

Not runtime polymorphic behavior

Just for the record, the above call to the method does not constitute runtime polymorphism *(in my opinion)* . I included that call to the method to serve as a backdrop for what follows.

Runtime polymorphic behavior

However, the following call to the method does constitute runtime polymorphism.

The statement in Listing 4 (p. 671) casts the reference down to type **A** and calls the method named **m()** on that reference.

It may not come as a surprise to you that the call to the method shown in Listing 4 (p. 671) also compiles and runs successfully.

Listing 4: Polymorphic behavior.

```
((A)var).m();
```

5.47

The method output

Here is the punch line. Not only does the statement in Listing 4 (p. 671) compile and run successfully, it produces the following output, *(which is exactly the same output as before)* :

```
m in class B
```

Same method executed in both cases

It is important to note that this output, *(produced by casting the reference variable to type* ***A*** *instead of type* ***B*** *)* , is exactly the same as that produced by the earlier call to the method when the reference was cast to type **B** . This means that the same version of the method was executed in both cases.

This confirms that, even though the type of the reference was converted to type **A** , *(rather than type* ***Object*** *or type* ***B*** *)* , the overridden version of the method defined in class **B** was actually executed.

This is an example of runtime polymorphic behavior.

The version of the method that was executed was based on

- the actual type of the object, **B** , and
- not on the type of the reference, **A** .

This is an extremely powerful and useful concept.

Another call to the method

Now take a look at the statement in Listing 5 (p. 672) . Will this statement compile and execute successfully? If so, which version of the method will be executed?

Listing 5: Source of a compiler error.

```
var.m();
```

5.48

Compiler error

The code in Listing 5 (p. 672) attempts, unsuccessfully, to call the method named **m()** using the reference variable named **var**, which is of type **Object**. The result is a compiler error, which, depending on your version of the JDK, will be similar to the following:

```
    Poly03.java:40: cannot resolve symbol
symbol  : method m  ()
location: class java.lang.Object
    var.m();
       ^
```

Some important rules

The **Object** class does not define a method named **m()**. Therefore, the overridden method named **m()** in the class named **B** is not an overridden version of a method that is defined in the class named **Object**.

Necessary, but not sufficient

Runtime polymorphism based on class inheritance requires that the type of the reference variable be a superclass of the class from which the object *(on which the method will be called)* is instantiated.

However, while necessary, that is not sufficient.

The type of the reference variable must also be a class that either ***defines or inherits*** the method that will ultimately be called on the object.

This method is not defined in the Object class

Since the class named **Object** neither defines nor inherits the method named **m()**, a reference of type **Object** does not qualify as a participant in runtime polymorphic behavior in this case. The attempt to use it as a participant resulted in the compiler error given above.

One additional scenario

Before leaving this topic, let's look at one additional scenario to help you distinguish what is, and what is not, runtime polymorphism. Consider the code shown in Listing 6 (p. 672).

Listing 6: A new object of type A.

```
    var = new A();
((A)var).m();
```

5.49

A new object of type A

The code in Listing 6 (p. 672) instantiates a new object of the class named **A** , and stores the object's reference in the original reference variable named **var** of type **Object** .

(As a side note, this overwrites the previous contents of the reference variable with a new reference and causes the object whose reference was previously stored there to become eligible for garbage collection.)

Downcast and call the method

Then the code in Listing 6 (p. 672) casts the reference down to type **A** , *(the type of the object to which the reference refers)* , and calls the method named **m()** on the downcast reference.

The output

As you would probably predict, this produces the following output on the computer screen:

```
m in class A
```

In this case, the version of the method defined in the class named **A** , *(not the version defined in* ***B*** *)* was executed.

Not polymorphic behavior

In my view, this is not polymorphic behavior *(at least it isn't a very useful form of polymorphic behavior)* . This code simply converts the type of the reference from type **Object** to the type of the class from which the object was instantiated, and calls one of its methods. Nothing special takes place regarding a selection among different versions of the method.

Some authors may disagree

While some authors might argue that this is technically runtime polymorphic behavior, in my view at least, it does not illustrate the real benefits of runtime polymorphic behavior. The benefits of runtime polymorphic behavior generally accrue when the actual type of the object is a subclass of the type of the reference variable containing the reference to the object.

Once again, what is runtime polymorphism?

As I have discussed in this module, runtime polymorphic behavior based on class inheritance occurs when

- The type of the reference is a superclass of the class from which the object was instantiated.
- The version of the method that is executed is the version that is either defined in, or inherited into, the class from which the object was instantiated.

More than you ever wanted to hear

And that is probably more than you ever wanted to hear about runtime polymorphism based on class inheritance.

A future module will discuss runtime polymorphism based on the Java interface. From a practical viewpoint, you will find the rules to be similar but somewhat different in the case of the Java interface.

A very important concept

For example, the entire event-driven graphical user interface structure of Java is based on runtime polymorphism involving the Java interface.

5.2.7.5 Summary

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through class inheritance
- Method overriding through the Java interface

This module discusses method overriding through class inheritance.

With runtime polymorphism based on method overriding, the decision as to which version of a method will be executed is based on the actual type of object whose reference is stored in the reference variable, and not on the type of the reference variable on which the method is called.

The decision as to which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as late binding.

This is illustrated in the sample program discussed in this module.

5.2.7.6 What's next?

In the next module, I will continue my discussion of the implementation of polymorphism using method overriding through class inheritance, and I will concentrate on a special case in that category.

Specifically, I will discuss the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and explain how that results in a very useful form of runtime polymorphism.

5.2.7.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Runtime Polymorphism through Class Inheritance
- File: Java1612.htm
- Published: 02/27/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.7.8 Complete program listing

A complete listing of the program is shown in Listing 7 (p. 675) below.

Listing 7: Complete program listing.

```
    /*File Poly03.java
Copyright 2002, R.G.Baldwin

This program illustrates downcasting
and polymorphic behavior

Program output is:

m in class B
m in class B
m in class A
************************************/

class A extends Object{
  public void m(){
    System.out.println("m in class A");
  }//end method m()
}//end class A
//===================================//

class B extends A{
  public void m(){
    System.out.println("m in class B");
  }//end method m()
}//end class B
//===================================//

public class Poly03{
  public static void main(String[] args){
    Object var = new B();
    //Following will compile and run
    ((B)var).m();
    //Following will also compile
    // and run due to polymorphic
    // behavior.
    ((A)var).m();
    //Following will not compile
    //var.m();
    //Instantiate obj of class A
    var = new A();
    //Call the method on it
    ((A)var).m();
  }//end main
}//end class Poly03
```

-end-

5.2.8 Java1614: Polymorphism and the Object Class[26]

5.2.8.1 Table of Contents

5.2.8.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.8.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.8.2.1.1 Listings

5.2.8.3 Preview

What is polymorphism?

If you have studied the earlier modules in this collection, you should already know what polymorphism is, how it is implemented in Java, the three distinct forms of polymorphism in Java, etc.

I discussed runtime polymorphism implemented through method overriding and class inheritance in a previous module. However, before leaving that topic, I need to discuss an important special case.

In this module, I will discuss the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and will explain how that results in a very useful form of runtime polymorphism.

I will briefly discuss the default versions of eleven methods defined in the **Object** class, and will explain that in many cases, those default versions are meant to be overridden.

[26] This content is available online at <http://cnx.org/content/m44190/1.3/>.

5.2.8.4 Discussion and sample code

The Java Collections Framework

Java supports a framework, known as the Java Collections Framework, which you can read about in other tutorials on my web site at http://www.dickbaldwin.com/toc.htm [27] .

Without getting into a lot of detail, the framework provides several concrete implementations of interfaces with names like **list** , **set** , and **map** .

The classes that provide the implementations have names like **LinkedList** , **TreeSet** , **ArrayList** , **Vector** , and **Hashtable** . As you might recognize, the framework satisfies the requirements for what we might refer to as classical data structures.

Not the purpose ...

However, it is not the purpose of this module to discuss either the Java Collections Framework, or classical data structures. Rather, they are mentioned here simply because the framework provides a good example of the use of the **Object** class as a generic type for runtime polymorphic behavior.

(Also beyond the scope of this module is the fact that the framework provides an outstanding example of the implementation of polymorphic behavior through the use of the Java interface. The use of the Java interface is a topic for a future module)

References of type Object

The classes mentioned above store references to objects created according to interfaces, contracts, and stipulations provided by the framework. More importantly for the purposes of this module, those references are stored as type **Object** .

(See my tutorial titled Generics in J2SE 5.0 at http://www.developer.com/java/other/article.php/3495121/Generics-in-J2SE-50.htm [28] for additional information on this topic.)

The **Object** type is a completely generic type, which can be used to store a reference to any object that can be instantiated in Java.

Methods defined in the Object class

In an earlier module, I told you that the class named **Object** defines default versions of the following methods:

- clone()
- equals(Object obj)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()
- wait(long timeout)
- wait(long timeout, int nanos)

Every class inherits these methods

Because every class is either a direct or indirect subclass of **Object** , every class in Java, *(including new classes that you define)* , inherits these eleven methods.

To be overridden ...

Some of these eleven methods are intended to be overridden for various purposes.

Calling methods of the Object class

You can store a reference to any object in a reference variable of type **Object** .

If you have studied the previous modules in this collection, you also know how runtime polymorphism based on class inheritance works.

[27] http://www.dickbaldwin.com/toc.htm
[28] http://www.developer.com/java/other/article.php/3495121/Generics-in-J2SE-50.htm

Given the above, you should know that you can call any of the methods defined in the **Object** class on any reference to any object stored in a reference variable of type **Object** *(including the references stored in the concrete implementations of the Java Collections Framework)* .

And the behavior will be ...

If the class from which that object is instantiated inherits or defines an overridden version of one of the methods in the above list, calling that method on the reference will cause the overridden version to be executed.

Otherwise, calling that method on the reference will cause the default version defined in the **Object** class to be executed.

A sample program

This is illustrated in the program named **Poly04** , which you can view in its entirety in Listing 7 (p. 684) near the end of this module.

For purposes of illustration, this program deals specifically with the method named **toString** from the above list, but it could deal just as well with other non-final methods in the list.

The class named A

Listing 1 (p. 678) defines a class named **A** , which extends the class named **Object** *(recall that it is not necessary to explicitly show that a class extends* ***Object***).

Listing 1: Definition of the class named A.

```
 class A extends Object{
  //This class is empty
}//end class A
```

5.51

Does not override the toString method

The most important thing to note about the class named **A** is that it does not override any of the methods that it inherits from the class named **Object** .

For purposes of this illustration, we will say that it inherits the default version of the method named **toString** , from the class named **Object** . *(We will see an example of the behavior of the default version of the* ***toString*** *method shortly.)*

The class named B

Listing 2 (p. 679) contains the definition of a class named **B** . This class extends the class named **A** .

Listing 2: Definition of the class named B.

```
    class B extends A{
  public String toString(){
    return "toString in class B";
  }//end overridden toString()
}//end class B
```

5.52

Overrides the toString method

Of particular interest, for purposes of this module, is the fact that the class named **B** does override the inherited t **oString** method.

*(The class named **B** inherits the default version of the method, because its superclass named **A** , which extends **Object** , does not override the **toString** method.)*

Purpose of the toString method

The purpose of the **toString** method is to return a reference to an object of the class **String** that represents an object instantiated from a class that overrides the method.

Here is part of what Sun has to say about the **toString** method:

*"Returns a string representation of the object. In general, the **toString** method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method."*

Behavior of the overridden version

As you can see, I didn't follow Sun's advice very closely in this program. To begin with, I didn't override the **toString** method in the class named **A** .

Further, the behavior of my overridden version of the **toString** method in the class named **B** doesn't provide much in the way of a textual representation of an object instantiated from class **B** .

My overridden version simply returns a reference to a **String** object, containing text that indicates that the overridden version of the method defined in the class named **B** has been executed. *(Of course, there wasn't much about an object instantiated from the class named **B** that could be represented in a textual way.)*

Will be useful later

The reference to the **String** object returned by the overridden version of the **toString** method will prove useful later when we need to determine which version of the method is actually executed.

The class named C

Listing 3 (p. 680) shows the definition of a class named **C** , which extends the class named **B** , and overrides the method named **toString** again. *(A non-final method can be overridden by every class that inherits it, resulting in potentially many different overridden versions of a method in a class hierarchy.)*

Listing 3: Definition of the class named C.

```
    class C extends B{
  public String toString(){
    return "toString in class C";
  }//end overridden toString()
}//end class B
```

5.53

Behavior of overridden version

The behavior of this overridden version of the method is similar to, but different from the overridden version in the class **B** .

In this case, the method returns a reference to a **String** object that can be used to confirm that this overridden version of the method has been executed.

The driver class

Finally, Listing 4 (p. 680) shows the beginning of the driver class named **Poly04** .

Listing 4: Beginning of the class named Poly04.

```
    public class Poly04{
  public static void main(String[] args){
    Object varA = new A();
    String v1 = varA.toString();
    System.out.println(v1);
```

5.54

A new object of the class A

The **main** method of the driver class begins by instantiating a new object of the class **A** , and saving the object's reference in a reference variable of type **Object** , named **varA** .

Call toString method on the reference

Then the code in Listing 4 (p. 680) calls the **toString** method on the reference variable named **varA** , saving the returned reference to the **String** in a reference variable of type **String** named **v1** .

Display the returned String

Finally, that reference is passed to the **println** method, causing the **String** returned by the **toString** method to be displayed on the computer screen.

(In a future module, you will learn that some of the code in Listing 4 (p. 680) *is redundant.)*

This causes the following text to be displayed on the computer screen:

```
A@111f71
```

Pretty ugly, huh?

Nowhere does our program explicitly show the creation of any text that looks anything like this. Where did it come from?

Default toString behavior

What you are seeing here is the **String** produced by the default version of the **toString** method, as defined by the class named **Object** .

Class A does not override toString

Recall that our new class named **A** does not override the **toString** method. Therefore, when the **toString** method is called on a reference to an object of the class **A** , the default version of the method is executed, producing output similar to that shown above (p. 680) .

What does Sun have to say?

Here is more of what Sun has to say about the default version of the **toString** method

*"The **toString** method for class **Object** returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object."*

You should recognize this as a description of the output produced by calling the **toString** method on the reference to the object of the class **A** . That explains the ugliness of the screen output shown above (p. 680) *(hexadecimal representations of hashcodes are usually pretty ugly)* .

A new object of the class B

Now consider the code shown in Listing 5 (p. 681) , which instantiates a new object of the class named **B** , and stores the object's reference in a reference variable of type **Object** .

Listing 5: A new object of the class named B.

```
Object varB = new B();
String v2 = varB.toString();
System.out.println(v2);
```

5.55

Call toString and display the result

The code in Listing 5 (p. 681) calls the **toString** method on the reference of type **Object** , saving the returned reference in the reference variable named **v2** . *(Recall that the **toString** method is overridden in the class named B.)*

As before, the reference is passed to the **println** method, which causes the following text to be displayed on the computer screen.

```
toString in class B
```

Do you recognize this?

You should recognize this as the text that was encapsulated in the **String** object by the overridden version of the **toString** method defined in the class named **B** .

Overridden version of toString was executed

This verifies that even though the reference to the object of the class **B** was stored in a reference variable of type **Object** , the overridden version of the **toString** method defined in the class named **B** was executed *(instead of the default version defined in the class named **Object**)* . This is a good example of *runtime polymorphic behavior* , as described in a previous module.

As you learned in the previous module, the selection of a method for execution is based on the *actual type of object* whose reference is stored in a reference variable, and **not** on the *type of the reference variable* on which the method is called.

An object of the class C

Finally, the code in Listing 6 (p. 682)

- Instantiates a new object of the class **C** .
- Stores the object's reference in a reference variable of type **Object** .
- Calls the **toString** method on the reference.
- Displays the returned string on the computer screen.

Listing 6: A new object of the class named C.

```
Object varC = new C();
String v3 = varC.toString();
System.out.println(v3);
```

5.56

What will the output look like?

By now, you should know what to expect in the way of text appearing on the computer screen. The code in Listing 6 (p. 682) causes the following text to be displayed:

```
toString in class C
```

Overridden version of toString was called

This confirms what you should already have known by now. In particular, even though the reference to the object of the class **C** is stored in a reference variable of type **Object** , the overridden version of the **toString** method defined in the class named **C** was executed. Again, this is *runtime polymorphic behavior* based on class inheritance and method overriding.

No downcasting was required

It is also very important to note that no downcasting was required in order to call the **toString** method in any of the cases shown above.

Because a default version of the **toString** method is defined in the **Object** class, the **toString** method can be called without a requirement for downcasting on a reference to any object stored in a variable of type **Object** . This holds true for any of the eleven methods defined in the class named **Object** *(although some of those methods are declared **final** and therefore may not be overridden)* .

5.2.8.5 Summary

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through class inheritance
- Method overriding through the Java interface

In this module, I have continued my discussion of the implementation of polymorphism using method overriding through class inheritance, and have concentrated on a special case in that category.

More specifically, in this module, I have discussed the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and have explained how that results in a very useful form of runtime polymorphism. .

I briefly mentioned the default version of the eleven methods defined in the **Object** class, and explained that in some cases, those default versions are meant to be overridden.

I provided a sample program that illustrates the overriding of the **toString** method, which is one of the eleven methods defined in the **Object** class.

5.2.8.6 What's next?

In the next module, I will embark on an explanation of runtime polymorphic behavior based on the Java interface and method overriding.

In my opinion, this is one of the most important concepts in Java OOP, and the one that seems to give students the greatest amount of difficulty.

5.2.8.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism and the Object Class
- File: Java1614.htm
- Published: 03/13/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.8.8 Complete program listing

A complete listing of the program is shown in Listing 7 (p. 684) below.

Listing 7: Complete program listing.

```
    /*File Poly04.java
Copyright 2002, R.G.Baldwin

This program illustrates polymorphic
behavior

Program output is:

A@111f71
toString in class B
toString in class C
************************************/

class A extends Object{
  //This class is empty
}//end class A
//===================================//

class B extends A{
  public String toString(){
    return "toString in class B";
  }//end overridden toString()
}//end class B
//===================================//

class C extends B{
  public String toString(){
    return "toString in class C";
  }//end overridden toString()
}//end class B
//===================================//

public class Poly04{
  public static void main(String[] args){
    Object varA = new A();
    String v1 = varA.toString();
    System.out.println(v1);

    Object varB = new B();
    String v2 = varB.toString();
    System.out.println(v2);

    Object varC = new C();
    String v3 = varC.toString();
    System.out.println(v3);

  }//end main
}//end class Poly04
```

-end-

5.2.9 Java1616: Polymorphism and Interfaces, Part 1[29]

5.2.9.1 Table of Contents

5.2.9.2 Preface

This module is one of a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java.

5.2.9.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.9.2.1.1 Listings

5.2.9.3 Preview

Method overloading

I covered method overloading as one form of polymorphism *(compile-time polymorphism)* in a previous module. I also explained automatic type conversion and the use of the cast operator for type conversion in a previous module.

Method overriding and class inheritance

I also discussed runtime polymorphism implemented through method overriding and class inheritance in previous modules.

Using the Java interface

In this module and the next, I will explain runtime polymorphism as implemented using method overriding and the Java interface.

A very important concept

[29] This content is available online at <http://cnx.org/content/m44195/1.3/>.

In my opinion, this is one of the most important concepts in Java OOP, and the one that seems to give students the greatest amount of difficulty. Therefore, I will try to take it slow and easy. As usual, I will illustrate the concept using sample programs.

I will also tie this concept back to the concept of polymorphism using method overriding through inheritance.

A skeleton program

In this module, I will present a simple skeleton program that illustrates many of the important aspects of polymorphic behavior based on the Java interface.

Multiple inheritance and the cardinal rule

I will explain how the implementation of interfaces in Java is similar to multiple inheritance. I will explain the cardinal rule of interface implementation.

A new relationship

I will explain that objects instantiated from classes that implement the same interface have a new relationship that goes beyond the relationship imposed by the standard class hierarchy.

One object, many types

I will explain that due to the combination of the class hierarchy and the fact that a class can implement many different interfaces, a single object in Java can be treated as many different types. However, for any given type, there are restrictions on the methods that can be called on the object.

Many classes, one type

I will explain that because different classes can implement the same interface, objects instantiated from different classes can be treated as a common interface type.

Interfaces are critical to Java programming

I will suggest that there is little if anything useful that can be done in Java without understanding and using interfaces.

In support of this suggestion, I will discuss several real-world examples of the use of the Java interface, including the Delegation Event Model, the Model View Control paradigm, and iterators in Java data structures.

5.2.9.4 Discussion and sample code

Listing 6 (p. 695) near the end of the module contains a very simple program named **Poly05** .

The purpose of this program is to illustrate polymorphic behavior using interfaces in addition to class inheritance.

Designed to illustrate structure

This is a skeleton program designed solely to illustrate the inheritance and interface implementation structure in as simple a program as possible. *(I will put some meat on this skeleton using another program in the next module.)*

Empty methods

Except for the two methods that return type **String** , all of the methods in the program are empty. *(Methods that return type **String** cannot be empty. They must contain a **return** statement in order to compile successfully.)*

Interface definitions

Listing 1 (p. 687) shows the definition of two simple interfaces named **I1** and **I2** .

Listing 1: Definition of interfaces named I1 and I2.

```
    interface I1{
  public void p();
}//end interface I1

//===================================//

interface I2 extends I1{
  public void q();
}//end interface I2
```

5.58

Similar but different

An interface definition is similar to a class definition. However, there are some very important differences.

No single hierarchy

To begin with, unlike the case with classes, there is no single interface hierarchy. Also, multiple inheritance is allowed when extending interfaces.

A new interface can extend none, one, or more existing interfaces. In Listing 1 (p. 687) , **I2** extends **I1** , but **I1** doesn't extend any other interface *(and unlike classes, an interface doesn't automatically extend another interface by default)* .

Two kinds of members allowed

Only two kinds of members are allowed in an interface definition:

- Methods, which are implicitly abstract
- Variables, which are implicitly constant *(final)*

Each of the interfaces in Listing 1 (p. 687) declares an implicitly abstract method *(an abstract method does not have a body)* .

Neither of the interfaces in Listing 1 (p. 687) declares any variables *(they aren't needed for the purpose of this module)* .

A new data type

I told you earlier that when you define a new class, you cause a new data type to become available to your program. The same is true of an interface definition. Each interface definition constitutes a new type.

The class named A

Listing 2 (p. 688) defines a very simple class named **A** , which in turn defines two methods named **toString** and **x** .

Listing 2: Definition of the class named A.

```
    class A extends Object{
  public String toString(){
    return "toString in A";
  }//end toString()
  //---------------------------------//

  public String x(){
    return "x in A";
  }//end x()
  //---------------------------------//
}//end class A
```

5.59

Overridden toString

The method named **toString** in Listing 2 (p. 688) is actually an overridden version of the method having the same name that is defined in the class named **Object** . *(Recall that a previous module made heavy use of overridden versions of the* ***toString*** *method.)*

New method

The method named **x** is newly defined in the class named **A** . *(The method named* ***x*** *is not inherited into the class named* ***A*** *, because the class named* ***Object*** *does not define a method named* ***x*** *.)*

The class named B

Listing 3 (p. 688) contains material that is new to this module.

Listing 3: Definition of the class named B.

```
    class B extends A implements I2{
  public void p(){
  }//end p()
  //---------------------------------//

  public void q(){
  }//end q();
  //---------------------------------//
}//end class B
```

5.60

Implementing an interface

Listing 3 (p. 688) defines a class named **B** , which *extends* the class named **A** , and *implements* the interface named **I2** .

As you already know, a class in Java can extend only one other class. However, a Java class can implement any number of interfaces. *(Multiple inheritance is allowed with interfaces.)*

Similar to an abstract class

An interface is similar, but not identical, to an abstract class. *(An abstract class may contain abstract methods or concrete methods, or a combination of the two while all of the methods in an interface are implicitly abstract.)*

Restrictions

An abstract class cannot be instantiated. Thus, an abstract class is only useful when it is extended by another class.

An interface also cannot be instantiated.

Implicitly abstract methods

As mentioned above, all methods declared in an interface are implicitly abstract, but that is not true for an abstract class. An abstract class can also contain fully-defined *(concrete)* methods. Regardless, an abstract class cannot be instantiated.

A totally abstract class

At the risk of offending the purists, I will coin a new term here and say that an interface is similar to a totally abstract class *(one which contains only abstract method declarations and final variables)* .

To a first degree of approximation then, we might say that the class named **B** is not only a subclass of the class named **A** , it is also a subclass of the *totally abstract* class named **I2** . *(This is pretty far out with respect to terminology, so to avoid being embarrassed, you probably shouldn't repeat it to anyone else.)*

Since **I2** extends **I1** , we might also say that the class named **B** is a subclass of the *totally abstract* class named **I1** .

A different kind of thinking

With this kind of thinking, we have suddenly make it possible for Java classes to support *multiple inheritance* , with the stipulation that all but one of the inherited classes must be *totally abstract classes* .

Be very careful with this way of thinking

However, we need to be very careful with this kind of thinking. While it may help some students to understand the role of interfaces in Java, there are probably some hidden dangers lurking here.

Back to the safety zone

The safest course of action is to simply say that the class named **B** :

- Extends the class named **A**
- Implements the interface named **I2** directly
- Implements the interface named **I1** through inheritance

Java does not support multiple inheritance, but it does allow you to extend one class and implement any number of interfaces.

The cardinal rule regarding interface implementation

The cardinal rule in implementing interfaces is:

If a class implements an interface, it must provide a concrete definition for all the methods declared by that interface, and all the methods inherited by that interface. Otherwise, the class must be declared abstract and the definitions must be provided by a class that extends the abstract class.

The cardinal rule regarding class inheritance

A similar rule exists for defining classes that inherit abstract methods from other classes:

If a class inherits one or more abstract methods from its superclasses, it must provide concrete definitions for all the inherited abstract methods. Otherwise, the class must be declared abstract and the concrete definitions must be provided by a class that extends the abstract class.

What does that mean in this case?

In this case, this means that the class named **B** must provide concrete definitions for the methods named **p** and **q** , because:

- The class named **B** implements the interface named **I2** .
- The method named **q** is declared in the interface named **I2** .
- The interface named **I2** extends the interface named **I1** .
- The method named **p** is declared in the interface named **I1** .

As in method overriding, the signature of the concrete method in the defining class must match the signature of the method as it is declared in the interface.

Class B satisfies the cardinal rule

As you can see from Listing 3 (p. 688) , the class named **B** does provide concrete *(but empty)* definitions of the methods named **p** and **q** .

(As mentioned earlier, I made the methods empty in this program for simplicity. However, it is not uncommon to define empty methods in classes that implement interfaces that declare a large number of methods, such as the ***MouseListener*** *interface. See my tutorials on event-driven programming at http://www.dickbaldwin.com/toc.htm* [30] *for examples.)*

The class named C

Listing 4 (p. 690) defines a class named **C** , which extends **Object** , and also implements **I2** . As in the case of the class named **B** , this class must, and does, provide concrete *(but empty)* definitions for the methods named **p** and **q** .

Listing 4: Definition of the class named C.

```
   class C extends Object implements I2{
  public void p(){
  }//end p()
  //---------------------------------//

  public void q(){
  }//end q();
  //---------------------------------//
}//end class B
```

5.61

A driver class

Finally, the driver class named **Poly05** shown in Listing 5 (p. 691) defines an empty **main** method.

[30] http://www.dickbaldwin.com/toc.htm

Listing 5: The driver class named Poly05.

```
   public class Poly05{
  public static void main(String[] args){
  }//end main
}//end class Poly05
```

5.62

Doesn't do anything

As mentioned earlier, the purpose of this program is solely to illustrate an inheritance and interface structure. This program can be compiled and executed, but it doesn't do anything useful.

A new relationship

At this point, it might be useful for you to sketch out the structure in a simple hierarchy diagram.

If you do, you will see that implementation of the interface named **I2** by the classes named **B** and **C** , has created a relationship between those two classes that is totally independent of the normal class hierarchical relationship.

What is the new relationship?

By declaring that both of these classes implement the same interface named **I2** , we are guaranteeing that an object of either class will contain concrete definitions of the two methods declared in the interfaces named **I2** and **I1** .

Furthermore, we are guaranteeing that objects instantiated from the two classes can be treated as the common type **I2** .

*(**Important** : references to any objects instantiated from classes that implement **I2** , can be stored in reference variables of the type **I2** , and any of the interface methods can be called on those references.)*

We know the user interface

The signatures of the interface methods in the two classes must match the signatures declared in the interfaces.

This means that if we have access to the documentation for the interfaces, we also know the signatures of the interface methods for objects instantiated from any class that implements the interfaces.

Different behavior

However, and this is extremely important, the behavior of the interface methods as defined in the class named **B** may be *(and often will be)* entirely different from the behavior of the interface methods having the same signatures as defined in the class named **C** .

Possibly the most powerful concept in Java

This is possibly the most powerful *(and most difficult)* concept embodied in the Java programming language.

If you don't understand interfaces ...

I usually tell my students several times each semester that if they don't understand interfaces, they don't really understand Java.

It is unlikely that you will ever be successful as a Java programmer without an understanding of interfaces.

There are very few worthwhile programs that can be written in Java without an understanding of interfaces.

The core aspect

So, what is the core aspect of this concept that is so powerful?

I told you earlier that each interface definition constitutes a new type. As a result, a reference to any object instantiated from any class that implements a given interface can be treated as the type of the interface.

So what!

When a reference to an object is treated as an interface type, any method declared in, or inherited into that interface can be called on the reference to the object.

However, the behavior of the method when called on references to different objects of the same interface type may be very different. In the current jargon, the behavior is *appropriate for the object on which it is called* .

One object, many types

Furthermore, because a single class can implement any number of different interfaces, a single object instantiated from a given class can be treated as any of the interface types implemented by the class from which it is instantiated. Therefore, a single object in Java can be treated as many different types.

(However, when an object is treated as an interface type, only those methods declared in that interface can be called on the object. To call other methods on the object, it necessary to cast the object's reference to a different type.)

Treating different types of objects as a common type

All of this also makes it possible to treat objects instantiated from widely differing classes as the same type, provided that all of those classes implement the same interface.

Important : When an interface method is called on one of the objects using the reference of the interface type, the behavior of the method will be as defined by the author of the specific class that implemented the interface. The behavior of the method will often be different for different objects instantiated from different classes that implement the same interface.

Receiving parameters as interface types

Methods can receive parameters that are references of interface types. In this case, the author of the code that calls interface methods on the incoming reference doesn't need to know, and often doesn't care, about the name of the class from which the object was instantiated. *(For a discussion of this capability, see my tutorials on Java Data Structures on my web site at http://www.dickbaldwin.com/toc.htm* [31] *.)*

A common example

A very common example is to store references to objects instantiated from different classes, *(which implement the same interface)* in some sort of data structure (*such as list or a set)* and then to call the same methods on each of the references in the collection.

Heart of the Delegation Event Model

For example, this methodology is at the heart of the *Delegation Event Model* , which forms the basis of Graphical User Interfaces and event-driven programming in Java.

This often entails defining classes that implement standard interfaces such as **MouseListener** , **WindowListener** , **TextListener** , etc. In this case, the programmer defines the interface methods to be appropriate for a listener object instantiated from a specific class. Then a reference to the listener object is registered on an event source *as the interface type* .

Later when an event of that type occurs, the source object calls one or more interface methods on the listener object using the reference of the interface type. The event source object doesn't know or care about the class from which the object was instantiated. In fact, it doesn't even care how the interface method behaves when it is called. The responsibility of the source object ends when it calls the appropriate interface method on the listener object.

Model View Control

This same methodology is also critical to the use of the *Model View Control* paradigm in Java using the **Observer** interface and the **Observable** class. In this case, view objects instantiated from different classes that implement the **Observer** interface can register themselves on a model object that extends the **Observable** class. Then each time the data being maintained in the model changes, each of the views will be notified so that they can update themselves.

[31] http://www.dickbaldwin.com/toc.htm

JavaBeans Components

This concept is also critical to the use of *bound* and *constrained* properties in JavaBeans Components. One bean can register itself on other beans to be notified each time the value of a bound or constrained property changes. In the case of constrained properties, the bean that is notified has the option of vetoing the change.

Java Collections Framework

The Java Collections Framework is also totally dependent on the use of interfaces. As I mentioned earlier, you can read all about this in my modules on Java Data Structures.

Iterators and Enumerators

If you appreciate data structures, you will also appreciate iterators. In Java, **Iterator** is an interface, and an object that knows how to iterate across a data structure is an object of a class that implements the **Iterator** interface.

As a result, the users of the concrete implementations in the Java Collections Framework don't need to know any of the implementation details of the collection to create and use an iterator. All of the work necessary to properly create an iterator is done by the author of the class that implements the appropriate Collection interfaces. All the user needs to understand is the behavior of the three methods declared in the Iterator interface.

5.2.9.5 Summary

Polymorphic behavior, based on the Java interface, is one of the most important concepts in Java OOP

In this module, I began my discussion of runtime polymorphism as implemented using method overriding and the Java interface.

I presented a simple skeleton program that illustrated many of the important aspects of polymorphic behavior based on the Java interface.

By using a nonstandard notation of my own design, *(a totally abstract class)* , I explained how the implementation of interfaces in Java is similar to multiple inheritance.

I explained the cardinal rule, which is:

If a class implements an interface, it must provide a concrete definition for all the methods declared by that interface, and all the methods inherited by that interface. Otherwise, the class must be declared abstract and the definitions must be provided by a class that extends the abstract class.

I explained that objects instantiated from classes that implement the same interface have a new relationship that goes beyond the relationship imposed by the standard class hierarchy.

I explained that due to the combination of the class hierarchy and the fact that a class can implement many different interfaces, a single object in Java can be treated as many different types. However, for any given type, there are restrictions on the methods that can be called on the object.

I also explained that because different classes can implement the same interface, objects instantiated from different classes can be treated as a common interface type.

I suggested that there is little if anything useful that can be done in Java without understanding and using interfaces.

Finally I discussed some real-world examples of the use of the Java interface:

- Delegation event model
- Model View Control paradigm
- Bound and constrained properties in JavaBeans Components
- Java Collections Framework Iterators and Enumerators

5.2.9.6 What's next?

In the next module, I will explain a more substantive program as I continue my discussion of polymorphic behavior using the Java interface.

5.2.9.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism and Interfaces, Part 1
- File: Java1616.htm
- Published: 03/27/02
- Revised: 01/01/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.9.8 Complete program listings

A complete listing of the sample program is shown in Listing 6 (p. 695) below.

Listing 6: Complete program listing.

```
    /*File Poly05.java
Copyright 2002, R.G.Baldwin
************************************/

interface I1{
  public void p();
}//end interface I1
//===================================//

interface I2 extends I1{
  public void q();
}//end interface I2
//===================================//

class A extends Object{
  public String toString(){
    return "toString in A";
  }//end toString()
  //---------------------------------//

  public String x(){
    return "x in A";
  }//end x()
  //---------------------------------//
}//end class A
//===================================//

class B extends A implements I2{
  public void p(){
  }//end p()
  //---------------------------------//

  public void q(){
  }//end q();
  //---------------------------------//
}//end class B
//===================================//

class C extends Object implements I2{
  public void p(){
  }//end p()
  //---------------------------------//

  public void q(){
  }//end q();
  //---------------------------------//
}//end class B
//===================================//

public class Poly05{
  public static void main(String[] args){
  }//end main
}//end class Poly05
```

-end-

5.2.10 Java1618: Polymorphism and Interfaces, Part 2[32]

5.2.10.1 Table of Contents

5.2.10.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.10.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them..

5.2.10.2.1.1 Listings

5.2.10.3 Preview

Method overloading

I covered method overloading as one form of polymorphism *(compile-time polymorphism)* in a previous module.

[32] This content is available online at <http://cnx.org/content/m44196/1.3/>.

Method overriding and class inheritance

I discussed *runtime polymorphism* implemented through method overriding and class inheritance in more than one previous module.

Using the Java interface

In this and the previous module, I am explaining runtime polymorphism as implemented using method overriding and the Java interface.

A very important concept

In my opinion, this is one of the most important concepts in Java OOP, and the one that seems to give students the greatest amount of difficulty. Therefore, I am trying to take it slow and easy. As usual, I am illustrating the concept using sample programs.

A skeleton program

In the previous module, I presented a simple skeleton program that illustrated many of the important aspects of polymorphic behavior based on the Java interface.

Multiple inheritance and the cardinal rule

I explained how the implementation of interfaces in Java is similar to multiple inheritance. I explained the cardinal rule of interface implementation.

A new relationship

I explained that objects instantiated from classes that implement the same interface have a new relationship that goes beyond the relationship imposed by the standard class hierarchy.

One object, many types

I explained that due to the combination of the class hierarchy and the fact that a class can implement many different interfaces, a single object in Java can be treated as many different types. However, for any given type, there are restrictions on the methods that can be called on the object.

Many classes, one type

I explained that because different classes can implement the same interface, objects instantiated from different classes can be treated as a common interface type.

Interfaces are critical to Java programming

I suggested that there is little if anything useful that can be done in Java without understanding and using interfaces. In support of this suggestion, I discussed several real-world examples of the use of the Java interface, including the Delegation Event Model and the Model View Control paradigm.

Another sample program

In this module, I will present another sample program that will take you deeper into the world of polymorphism as implemented using the Java interface.

The sample program that I will discuss in this module will illustrate *(in a very basic form)* some of the things that you can do with interfaces, along with some of the things that you cannot do with interfaces. In order to write programs that do something worthwhile, you will need to extend the concepts illustrated by this sample program into real-world requirements.

5.2.10.4 Discussion and sample code

Now, let's take a look at a sample program named **Poly06** that is much simpler than any of the real-world examples that you will see in future modules.

This program is designed to be very simple, while still illustrating runtime polymorphism using interfaces, class inheritance, and overridden methods.

You can view a complete listing of the program in Listing 15 (p. 708) near the end of the module.

Same structure as before

Note that this program has the same structure as **Poly05** discussed in the previous module. *(I strongly recommend that you study the previous module before continuing with this module.)* However, unlike the program in the previous module, the methods in this version of the program are not empty. When a method is called in this version, something happens. *(Admittedly not much happens. Text is displayed on the computer screen, but that is something.)*

The interface definitions

Listing 1 (p. 698) shows the definition of the two interfaces named **I1** and **I2** .

Listing 1: Definition of the interfaces named I1 and I2.

```
    interface I1{
  public void p();
}//end interface I1

//=================================//

interface I2 extends I1{
  public void q();
}//end interface I2
```

5.64

Since the methods declared in an interface are not allowed to have a body, these interface definitions are identical to those shown in the program from the previous module.

The class named A

Similarly, Listing 2 (p. 698) shows the definition of the class named **A** along with the definition of the method named **x** , and the overridden method named **toString** .

Listing 2: Definition of the class named A.

```
    class A extends Object{

  public String toString(){
    return "toString in A";
  }//end toString()
  //--------------------------------//

  public String x(){
    return "x in A";
  }//end x()
  //--------------------------------//
}//end class A
```

5.65

These two methods were also fully defined in the program from the previous module, so there is no change here either.

The method named B

Listing 3 (p. 699) defines the class named **B** , which extends **A** , and implements **I2** .

Listing 3: Definition of the class named B.

```
    class B extends A implements I2{
  public void p(){
    System.out.println("p in B");
  }//end p()
  //---------------------------------//

  public void q(){
    System.out.println("q in B");
  }//end q();
  //---------------------------------//
}//end class B
```

5.66

Actually implements two interfaces

Although it isn't obvious from an examination of Listing 3 (p. 699) alone, the class named **B** actually implements both **I2** and **I1** . This is because the interface named **I2** extends **I1** . Thus, the class named **B** implements **I2** directly, and implements **I1** through interface inheritance.

The cardinal rule

In case you have forgotten it, the cardinal rule for implementing interfaces is:

If a class implements an interface, it must provide a concrete definition for all the methods declared by that interface, and all the methods inherited by that interface. Otherwise, the class must be declared abstract and the definitions must be provided by a class that extends the abstract class.

Must define two methods

As a result, the class named **B** must provide concrete definitions for the methods **p** and **q** . *(The method named p is declared in interface I1 and the method named q is declared in interface I2.)*

As you can see from Listing 3 (p. 699) , the behavior of each of these methods is to display a message indicating that it has been executed. This will be useful later to tell us exactly which method is executed when we exercise the objects in the **main** method of the driver class.

The class named C

Listing 4 (p. 700) shows the upgraded version of the class named **C** .

Listing 4: Definition of the class named C.

```
    class C extends Object implements I2{
  public void p(){
    System.out.println("p in C");
  }//end p()
  //---------------------------------//

  public void q(){
    System.out.println("q in C");
  }//end q();
  //---------------------------------//
}//end class B
```

5.67

In this upgraded version, the methods named **p** and **q** each display a message indicating that they have been executed. Again, this will be useful later to let us know exactly which version of the methods named p and q get executed when we exercise the objects.

The driver class

Listing 5 (p. 700) shows the beginning of the class named **Poly06** . The **main** method in this class instantiates objects of the classes named **B** and **C** , and exercises them to illustrate what can, and what cannot be done with them.

Listing 5: Beginning of the class named Poly06.

```
    public class Poly06{
  public static void main(
                       String[] args){
    I1 var1 = new B();
    var1.p();//OK
```

5.68

A new data type

As explained in the previous module, when you define a new interface, you create a new data type.

You can store the reference to any object instantiated from any class that implements the interface in a reference variable of that type.

A new object of the class B

The code shown in Listing 5 (p. 700) instantiates a new object of the class **B** .

Important: stored as type I1

It is important to note that the code in Listing 5 (p. 700) stores the object's reference in a reference variable of the interface type **I1** *(not as the class type* ***B*** *)* .

Call an interface method

Following this, the code in Listing 5 (p. 700) successfully calls the method named **p** on the reference, producing the following output on the computer screen:

```
p in B
```

Why is this allowed?

This is allowable because the method named **p** is declared in the interface named **I1** .

Which version of the method was executed?

It is also important to note, *(by observing the output)* , that the version of the method defined in the class named **B** *(and not the version defined in the class named **C**)* was actually executed.

Attempt unsuccessfully to call q

Next, the code in Listing 6 (p. 701) attempts, unsuccessfully, to call the method named **q** on the same reference variable of type **I1** .

Listing 6: Try unsuccessfully to call the method named q.

```
var1.q();//won't compile
```

5.69

Why did it fail?

Even though the class named **B** , from which the object was instantiated, defines the method named **q** , that method is neither declared nor inherited into the interface named **I1** .

Therefore, a reference of type **I1** cannot be used to call the method named **q** .

The solution is a type conversion

Listing 7 (p. 701) shows the solution to the problem presented by Listing 6 (p. 701) .

Listing 7: Successfully call the method named q.

```
((I2)var1).q();//OK
```

5.70

As in the case of polymorphism involving class inheritance, the solution is to change the type of the reference to a type that either declares or inherits the method named **q** .

In this case, this takes the form of using a cast operator to convert the type of the reference from type **I1** , to type **I2** , and then calling the method named **q** on that reference of a new type.

This produces the following output:

```
q in B
```

Using type I2 directly

Listing 8 (p. 702) instantiates a new object of the class **B** and stores the object's reference in a reference variable of the interface type **I2** .

Listing 8: Instantiate a new object of the class B.

```
   I2 var2 = new B();
var2.p();//OK
 var2.q();//OK
```

5.71

Call both methods successfully

Then the code successfully calls both the methods **p** and **q** on that reference, producing the following output:

```
 p in B
q in B
```

Why does this work?

This works because:

- The interface named **I2** declares the method named **q**
- The interface named **I2** inherits the declaration of the method named **p**
- The class named **B** implements the interface named **I2** and provides concrete definitions of both the methods **p** and **q** .

Attempt, unsuccessfully, to call x on var2

Following this, the code in Listing 9 (p. 702) attempts, unsuccessfully, to call the method named **x** on the reference variable named **var2** of type **I2** . This code produces a compiler error.

Listing 9: Try unsuccessfully to call the method named x.

```
String var3 = var2.x();
```

5.72

The object of class B has a method named x

At this point, the reference variable named **var2** contains a reference to an object instantiated from the class named **B** .

Furthermore, the class named **B** inherits the method named **x** from the class named **A** .

Necessary, but not sufficient

However, the fact that the object contains the method is not sufficient to make it executable in this case.

Same song, different verse

The interface named **I2** neither declares nor inherits the method named **x** .

Therefore, the method named **x** cannot be called using the reference stored in the variable named **var2** unless the reference is converted either to type **A** *(where the method named* **x** *is defined)* or type **B** *(where the method named x is inherited)* .

Do the type conversion

The required type conversion is accomplished in Listing 10 (p. 703) where the reference is temporarily converted to type **A** using a cast operator. *(It would also work to cast it to type* ***B*** *.)*

Listing 10: Successfully call the method named x.

```
    String var3 = ((A)var2).x();//OK
System.out.println(var3);
```

5.73

The String produced by the first statement in Listing 10 (p. 703) is passed to the **println** method causing the following text to be displayed on the computer screen:

```
x in A
```

Get ready for a surprise

If you have now caught onto the general scheme of things, the next thing that I am going to show you may result in a little surprise.

Successfully call the toString method on var2

The first statement in Listing 11 (p. 703) successfully calls the **toString** method on the object of the class **B** whose reference is stored as type **I2** .

Listing 11: Call the toString method.

```
  var3 = var2.toString();//OK
System.out.println(var3);
```

5.74

How can this work?

How can this work when the interface named **I2** neither declares nor inherits a method named **toString** .

A subtle difference in behavior

I am unable to point you to any Sun documentation to verify the following. *(I also admit that I haven't spent a large amount of time searching for such documentation).*

With respect to the eleven methods declared in the **Object** *class (listed in an earlier module)* , a reference of an interface type acts like it is also of type **Object** .

And the end result is ...

This allows the methods declared in the **Object** class to be called on references held as interface types without a requirement to cast the references to type **Object** . *(Later, I will show you that the reverse is not true.)*

The output

Therefore, the two statements shown in Listing 11 (p. 703) cause the following to be displayed on the computer screen:

```
toString in A
```

Polymorphism applies

Note that the object whose reference is held in **var2** was instantiated from the class named **B**, which extends the class named **A**.

Due to polymorphism, the **toString** method that was actually executed in Listing 11 (p. 703) was the overridden version defined in class **A**, and not the default version defined in the **Object** class. The overridden version in class **A** was inherited into class **B**.

The reverse is not true

While a reference of an interface type also acts like type **Object**, a reference of type **Object** does not act like an interface type.

Store a reference as type Object

The code in Listing 12 (p. 704) instantiates a new object of type **B** and stores it in a reference of type **Object**.

Attempt unsuccessfully to call p

Then it attempts, unsuccessfully, to call the method named **p** on the reference.

Listing 12: Try unsuccessfully to call the method named p.

```
Object var4 = new B();
var4.p();//won't compile
```

5.75

Same song, an even different verse

The code in Listing 12 (p. 704) won't compile, because the **Object** class neither defines nor inherits the method named **p**.

In order to call the method named **p** on the reference of type **Object**, the type of the reference must be changed to either:

- The class in which the method is defined
- An interface that declares the method, which is implemented by the class in which the method is defined
- A couple of other possibilities involving subclasses or sub-interfaces

Convert reference to type I1

The code in Listing 13 (p. 704) uses a cast operator to convert the reference from type **Object** to type **I1**, and calls the method named **p** on the converted reference.

Listing 13: Successfully call the method named p.

```
((I1)var4).p();//OK
```

5.76

The output

The code in Listing 13 (p. 704) compiles and executes successfully, producing the following text on the computer screen:

```
p in B
```

A walk in the park

If you understand all of the above, understanding the code in Listing 14 (p. 705) should be like a walk in the park on a sunny day.

A walk in the park.

```
    var2 = new C();
var2.p();//OK
    var2.q();//OK
```

5.77

Class C implements I2

Recall that the class named **C** also implements the interface named **I2** .

The code in Listing 14 (p. 705) instantiates a new object of the class named **C** , and stores the object's reference in the existing reference variable named **var2** of type **I2** .

Then it calls the methods named **p** and **q** on that reference, causing the following text to be displayed on the computer screen:

```
    p in C
q in C
```

Which methods were executed?

This confirms that the methods that were actually executed were the versions defined in the class named **C** (*and not the versions defined in the class named* ***B***) .

Same method name, different behavior

It is important to note that the behavior of the methods named **p** and **q** , as defined in the class named **C** , is different from the behavior of the methods having the same signatures defined in the class named **B** . Therein lies much of the power of the Java interface.

The power of the Java interface

Using interface types, it is possible to collect many objects instantiated from many different classes *(provided all the classes implement a common interface)* , and store each of the references in some kind of collection as the interface type.

Appropriate behavior

Then it is possible to call any of the interface methods on any of the objects whose references are stored in the collection.

To use the current jargon, when a given interface method is called on a given reference, the behavior that results will be *appropriate* to the class from which that particular object was instantiated.

This is runtime polymorphism based on interfaces and overridden methods.

5.2.10.5 Summary

If you don't understand interfaces ...

If you don't understand interfaces, you don't understand Java, and it is highly unlikely that you will be successful as a Java programmer.

Interfaces are indispensable in Java

Beyond writing "Hello World" programs, there is little if anything that can be accomplished using Java without understanding and using interfaces.

What can you do with interfaces?

The sample program that I discussed in this module has illustrated *(in a very basic form)* some of the things that you can do with interfaces, along with some of the things that you cannot do with interfaces.

In order to write programs that do something worthwhile, you will need to extend the concepts illustrated by this sample program into real-world requirements.

5.2.10.6 What's next?

Java supports the use of **static** member variables and **static** methods in class definitions.

While **static** members can be useful in some situations, the existence of **static** members tends to complicate the overall object-oriented structure of Java.

Furthermore, the overuse of **static** members can lead to problems similar to those experienced in languages like C and C++ that support global variables and global functions.

The use of static members will be discussed in the next module.

5.2.10.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Polymorphism and Interfaces, Part 2
- File: Java1618.htm
- Published: 04/10/02
- Revised: 01/01/13

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.10.8 Complete program listing

A complete listing of the sample program is shown in Listing 15 (p. 708) below.

Listing 15: Complete program listing.

```
    /*File Poly06.java
Copyright 2002, R.G.Baldwin

This program illustrates polymorphic
behavior using interfaces in addition
to class inheritance.

The program output is:
p in B
q in B

p in B
q in B
x in A
toString in A

p in B

p in C
q in C
************************************/

interface I1{
  public void p();
}//end interface I1
//===================================//

interface I2 extends I1{
  public void q();
}//end interface I2
//===================================//

class A extends Object{

  public String toString(){
    return "toString in A";
  }//end toString()
  //---------------------------------//

  public String x(){
    return "x in A";
  }//end x()
  //---------------------------------//
}//end class A
//===================================//

class B extends A implements I2{
  public void p(){
    System.out.println("p in B");
  }//end p()
  //---------------------------------//

  public void q(){
    System.out.println("q in B");
  }//end q();
```

-end-

5.2.11 Java1620: Static Members[33]

5.2.11.1 Table of Contents

5.2.11.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.11.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.11.2.1.1 Images

5.2.11.2.1.2 Listings

[33] This content is available online at <http://cnx.org/content/m44197/1.4/>.

- Listing 12 (p. 721) . Revisiting System.out.println.
- Listing 13 (p. 726) . Complete program listing.

5.2.11.3 Preview

Static members

There is another aspect of OOP in Java that I have avoided up to this point in the discussion: ***static*** variables and ***static*** methods.

Tends to complicate ...

I have avoided this topic because, while not particularly difficult, the existence of **static** members tends to break up the simple structures that I have discussed in previous modules in this collection.

While **static** members can be useful in some situations, the existence of **static** members tends to complicate the overall object-oriented structure of Java.

Avoid overuse of static members

Furthermore, the overuse of **static** members can lead to problems similar to those experienced in languages like C and C++ that support global variables and global functions.

When to use static members

I will discuss the use of **static** members in this module, and will provide some guidelines for their use.

The class named Class

I will also introduce the class named **Class** and discuss how it enters into the use of **static** variables and methods.

Instance members versus class members

I will describe the differences between *instance* members and *class* members with particular emphasis being placed on their accessibility.

Three kinds of objects

From a conceptual viewpoint, there are at least three kinds of objects involved in a Java program:

- Ordinary objects
- Array objects
- Class objects

Ordinary objects

All *(or at least most)* of the discussion up to this point in the collection deals with what I have referred to in the above list as *ordinary objects* .

These are the objects that you instantiate in you code by applying the **new** operator to a constructor for a class in order to create a new instance *(object)* of that class. *(There are also a couple of other ways to create ordinary objects, but I'm not going to get into that at this time.)*

Array objects

I haven't discussed *array objects* thus far in this collection. *(I will discuss them in a future module.)*

Suffice it for now to say that array objects are objects whose purpose is to encapsulate a one-dimensional array structure that can contain either primitive values, or references to other objects *(including other array objects)*.

I will discuss Class objects in this module.

5.2.11.4 Discussion and sample code

Class objects

Let me emphasize at the beginning that the following discussion is ***conceptual*** in nature. In this discussion, I will describe how the Java system behaves, not necessarily how it is implemented. In other words, however it is implemented, it behaves as though it is implemented in the manner described below.

The class named Class

There is a class whose name is **Class** . The purpose of this class is to encapsulate information about some other class *(actually, it can also be used to encapsulate information about primitive types as well as class types).*

Here is part of what Sun has to say about this class:

"Instances of the class ***Class*** *represent classes and interfaces in a running Java application. ...*

Class *has no public constructor. Instead* ***Class*** *objects are constructed automatically by the Java Virtual Machine as classes are loaded ..."*

What does this mean?

As a practical matter, when one or more objects are instantiated from a given class, an extra object of the **Class** class is also instantiated automatically. This object contains information about the class from which the objects were instantiated. *(Note that it is also possible to cause a* ***Class*** *object that describes a specific class to be created in the absence of objects of that class, but that is a topic that will be reserved for more advanced modules.)*

A real-world analogy

Here is an attempt to describe a real-world analogy. Remember that a class definition contains the blueprint for objects instantiated from that class.

A certain large construction company is in the business of building condominium projects. This contractor builds condos of many different sizes, types, and price ranges. However, each different condo project contains condos of only two or three different types or price ranges.

A library of blueprints

There is a large library of blueprints at the contractor's central office. This library contains blueprints for all of the different types of condos that the contractor has built or is building. *(This library is analogous to the class libraries available to the Java programmer.)*

A subset from the blueprint library

When a condo project begins, the contractor delivers copies of several sets of blueprints to the construction site. The blueprints delivered to that site describe only the types of condos being constructed on that site.

Condo is analogous to an object

Each condo unit is analogous to an *ordinary Java object* .

Each set of blueprints delivered to the construction site is roughly analogous to an *object of the class named* ***Class*** . In other words, each set of blueprints describes one or more condo units constructed from that set of blueprints.

When construction is complete

When the construction project is complete, the contractor delivers a set of blueprints for each type of condo unit to the management firm that has been hired to manage the condo complex. Each set of blueprints continues to be analogous to an object of the class named **Class** . The blueprints remain at the site of the condo units.

RTTI

Thus, information regarding the construction, wiring, plumbing, air conditioning, etc., for each condo unit *(object)* continues to be available at the site even after the construction has been completed. *(This is somewhat analogous to something called runtime type information and often abbreviated as RTTI. A Class object contains RTTI for objects instantiated from that class.)*

What are those analogies again?

In the above scenario, each condo unit is *(roughly)* analogous to an object instantiated from a specific class *(set of blueprints).*

Each set of blueprints remaining onsite after construction is complete is roughly analogous to a **Class** object that describes the characteristics of one or more condo units.

What do you care?

Until you get involved in such advanced topics as *reflection* and *introspection* , you don't usually have much involvement or much interest in **Class** objects. They are created automatically, and are primarily used by the Java virtual machine during runtime to help it do the things that it needs to do.

An exception to that rule

However, there is one area where you will be interested in the use of these **Class** objects from early on. You will be interested whenever variables or methods in the class definition are declared to be **static** .

Class variables and class methods

According to the current jargon, declaring a member variable to be **static** causes it to be a *class variable* . *(Note that local variables cannot be declared* ***static*** *. Only member variables can be declared* ***static*** *.)* Similarly, declaring a method to be **static** causes it to be a *class method.*

Instance variables and methods

On the other hand, according to the current jargon, not declaring a variable to be **static** causes it to be an *instance variable* , and not declaring a method to be **static** causes it to be an *instance method* .

In general, we can refer to them as *class members* and *instance members* .

What is the difference?

Here are some of the differences between *class* and *instance* members insofar as this discussion is concerned.

How many copies of member variables exist?

Every object instantiated from a given class has its own copy of each *instance variable* defined in the class. *(Instance variables are not shared among objects.)* However, every object instantiated from a given class shares the same copy of each *class variable* defined in the class. *(It is as though the class variable belongs to the single* ***Class*** *object and not to the individual objects instantiated from that class.)*

Access to an instance variable

Every object has its own copy of each instance variable *(the object owns the instance variable).* Therefore, the only way that you can access an instance variable is to use that object's reference to send a message to the object requesting access to the variable *(even then, you may not be given access, depending on access modifiers).*

Why call it an instance variable?

According to the current jargon, ***an object is an instance of a class*** . *(I probably told you that somewhere before in this collection.)* Each object has its own copy of each non-static variable. Hence, they are often called instance variables. *(Every instance of the class owns one and they are not implicitly shared among instances.)*

Access to a class variable

You can also send a message to an object requesting access to a class variable that the object shares with other objects instantiated from the same class. *(Again, you may or may not gain access, depending the access modifiers).*

Access using the Class object

More importantly, you can also access a class variable without a requirement to go through an object instantiated from the class. *(In fact, a class variable can be accessed in the total absence of objects of that class.)* *(Remember, this discussion is conceptual in nature, and may not represent an actual implementation.)*

Assuming that a class variable is otherwise accessible, you can access the class variable by sending an access request message to the **Class** object to which the variable belongs.

One way to think of this

To help you keep track of things in a message-passing sense, you can pretend that there is a global reference variable whose name is the same as the name of a class.

This *(hypothetical)* reference variable contains a reference to the **Class** object that owns the class variable. Using standard Java message-passing syntax, you can access the class variable by joining the name of the reference variable to the name of the class variable with a period. Example syntax is shown below:

```
ReferenceVariableName.ClassVariableName
```

As a result of the hypothetical substitution process that I described above, this is equivalent to the following:

```
ClassName.ClassVariableName
```

We will see an example of this in the sample program that I will discuss later.

Be careful with this thought process

While this thought process may be useful when thinking about ***static*** *variables and methods, I want to point out, that the thought process breaks down very quickly when dealing with* ***Class*** *objects in a deeper sense.*

For example, when calling the ***getName*** *method on a* ***Class*** *object, an actual reference of type* ***Class*** *is required to access the members of the* ***Class*** *object. The name of the class will not suffice.*

If this discussion of a global reference variable whose name matches the name of the class is confusing to you, just forget it. Simply remember that you can access class variables by joining the name of the class to the name of the class variable using a period as the joining operator.

Characteristics of class methods

I'm not going to talk very much about instance methods and class methods in this module. However, there are a couple of characteristics of class methods that deserve a brief discussion in this context.

Cannot access instance members

First, the code in a class method has direct access only to other **static** members of the class. *(A class method does not have direct access to instance variables or instance methods of the class.)* This is sort of like saying that a class method has access to the methods and variables belonging to the **Class** object, but does not have access to the methods and variables belonging to the *ordinary objects* instantiated from the class described by the **Class** object.

Once again, be careful

Once again, this thinking breaks down very quickly once you get beyond ***static*** *members. A* ***Class*** *object also has instance methods, such as* ***getName*** *, which can only be accessed using an actual reference to the* ***Class*** *object.*

Now you are probably beginning to understand why I deferred this discussion until after I finished discussing the easy stuff.

No object required

Another important characteristic is that a class method can be accessed without a requirement for an object of the class to exist.

As with class variables, class methods can be accessed by joining the name of the class to the name of the method with a period.

I will illustrate much of this with a sample program named **MyClass01** .

Discuss in fragments

I will discuss the program in fragments. You will find a complete listing of the program in Listing 13 (p. 726) near the end of the module.

Listing 1 (p. 713) shows the beginning of the class definition.

Listing 1: Beginning of the class named MyClass01.

```
 class MyClass01{
static Date v1 = new Date();
Date v2 = new Date();
```

5.79

Two member variables

The code in Listing 1 (p. 713) declares two member variables, named **v1** and **v2** , and initializes each of those variables with a reference to a new object of the **Date** class. *(When instantiated using*

the constructor with no arguments, the new ***Date*** *object encapsulates the current date and time from the system clock.)*

Note the static keyword

The important thing to note here is the use of the **static** keyword when declaring the variable named **v1** . This causes **v1** to be a *class variable* , exhibiting the characteristics of class variables described earlier.

An instance variable

On the other hand, the variable named **v2** is not declared **static** . This causes it to be an *instance variable* , as described above.

The main method is a class method

Listing 2 (p. 714) shows the signature for the **main** method.

Listing 2: Signature of the main method.

```
public static void main(String[] args){
```

5.80

The important thing to note here is that the **main** method is declared **static** . That causes it to be a *class method* .

As a result, the **main** method can be called without a requirement for an object of the class to exist. *(Also, the main method has direct access only to other* ***static*** *members.)*

How a Java application starts running

In fact, that is how the Java Virtual Machine starts an application running.

First the JVM finds the specified file having an extension of ***.class.*** Then it examines that file to see if it has a **main** method with the correct signature. If not, an error occurs.

If the JVM finds a **main** method with the correct signature, it calls that method without instantiating an object of the class. That is how the Java Virtual Machine causes a Java application to start running.

A side note regarding applets

For those of you who are familiar with Java applets, you should know that this is not the case for an applet. An applet does not use a ***main*** *method. When an applet is started, an object of the controlling class is instantiated by the browser, by the* ***appletviewer*** *program, or by whatever program is being used to control the execution of the applet.*

A poor programming technique

Basically, this entire sample program is coded inside the **main** method. As a practical manner, this is a very poor programming technique, but it works well for this example.

Display some text

The code in Listing 3 (p. 715) , which is the first executable statement in the **main** method, causes the words **Static variable** to appear on the computer screen. I will come back and discuss the details of this and similar statements later in the module.

Listing 3: Display some text.

```
System.out.println("Static variable");
```

5.81

Display date information

Continuing with the code in the **main** method, the code in Listing 4 (p. 715) causes the current contents of the **Date** object referred to by the contents of the class variable named **v1** to be displayed on the computer screen.

Listing 4: Display date information.

```
System.out.println(MyClass01.v1);
```

5.82

No object required

For the moment, concentrate on the text inside the parentheses in the statement in Listing 4 (p. 715) .

Because the variable named **v1** is a class variable, it's value is accessed by joining the name of the class to the name of the variable with a period.

What was the output?

I will discuss the remaining portion of statements of this sort later. For now, just be aware that the code in Listing 4 (p. 715) caused the output shown in Image 1 (p. 715) to be displayed on my computer screen when I ran the program.

Image 1: Output date and time.

```
Mon Sep 17 09:52:27 CDT 2001
```

5.83

Displays date and time

Obviously, the date and time displayed will depend on when you run the program. As you can see, I first wrote this module and ran this program in 2001.

Pay particular attention to the seconds portion of the time. I will refer back to this later.

A five-second delay

The code in Listing 5 (p. 716) *(still in the* ***main*** *method)* causes the main thread of the program to go to sleep for five seconds. Don't worry about it if you don't understand this code. The only reason that I included it in the program was to force a five-second delay in the execution of the program.

Listing 5: A five-second delay.

```
    try{
  Thread.currentThread().sleep(5000);
 }catch(InterruptedException e){}
```

5.84

Instantiate a new object

Having caused the program to sleep for five seconds, the code in Listing 6 (p. 716) instantiates a new object of the class named **MyClass01** . This code stores the new object's reference in the reference variable named **ref1** .

Listing 6: Instantiate a new object .

```
MyClass01 ref1 = new MyClass01();
```

5.85

A new Date object also

Recall from Listing 1 (p. 713) above that the class declares an instance variable named **v2** of the type **Date** .

When the new object is instantiated by the code in Listing 6 (p. 716) , a new **Date** object is also instantiated. A reference to that object is stored in the instance variable named **v2** . *(In other words, the new object of the class **MyClass01** owns a reference to a new object of the class **Date** . That reference is stored in an instance variable named **v2** in the new **MyClass01** object.)*

Display the new Date object

The code in Listing 7 (p. 716) causes a textual representation of the new **Date** object referred to by the reference variable named **v2** belonging to the object referred to by the reference variable named **ref1** , to be displayed on the standard output device.

Listing 7: Display the new Date object .

```
System.out.println(ref1.v2);
```

5.86

Five seconds later

This code caused the date and time shown in Image 2 (p. 717) to appear on the computer screen when I ran the program:

Image 2: Five seconds later.

```
Mon Sep 17 09:52:32 CDT 2001

                    5.87
```

The date and time shown in Image 2 (p. 717) is five seconds later than the time reflected in the **Date** object referred to by the *class variable* named **v1** *(see Image 1 (p. 715))* . That time was displayed by the code in Listing 4 (p. 715) earlier.

So, what does this mean?

It means that the **Date** object referred to by the **static** reference variable named **v1** was created five seconds earlier than the **Date** object referred to by the instance variable named **v2** .

When is a class variable created?

I can't tell you precisely when a class variable comes into existence. All I can say is that the virtual machine brings it into existence as soon as it is needed.

My guess is that it comes into existence at the first mention *(in the program)* of the class to which it belongs.

When is an instance variable created?

An instance variable doesn't come into existence until the object to which it belongs is created. *(An instance variable cannot exist until the object to which it belongs exists.)*

If the instance variable is initialized with a reference to a new object *(such as a new **Date** object in this sample program)*, that new object comes into existence when the object to which it belongs comes into existence.

A five-second delay

In this program, I purposely inserted a five-second delay between the first mention of the class named **MyClass01** in Listing 4 (p. 715) , and the instantiation of the object of the class named **MyClass01** in Listing 6 (p. 716) .

As a result, the **Date** object referred to by the instance variable named **v2** was created about five seconds later than the **Date** object referred to by the class variable named **v1** .

This is reflected in the date and time values displayed and discussed above.

Accessing class variable via an object

While it is possible to access a class variable using the name of the class joined to the name of the variable, it is also possible to access a class variable using a reference to any object instantiated from the class.

(As mentioned earlier, if two or more objects are instantiated from the same class, they share the same class variable.)

The code in parentheses in Listing 8 (p. 718) uses the reference variable named **ref1** to access the class variable named **v1** , and to cause the contents of the **Date** object referred to by the class variable to be displayed.

Listing 8: Accessing class variable via an object.

```
System.out.println(ref1.v1);
```

5.88

The output

This caused the date and time shown in Image 3 (p. 718) to be displayed on my computer screen.

Image 3: Same date and time as before.

```
Mon Sep 17 09:52:27 CDT 2001
```

5.89

Same date and time as before

As you have probably already surmised, this is the same date and time shown earlier in Image 1 (p. 715) . This is because the code in Listing 8 (p. 718) refers to the same class variable as the code in Listing 4 (p. 715) .

Nothing has caused the contents of that class variable to change, so both Image 1 (p. 715) and Image 3 (p. 718) display the contents of the same **Date** object.

(Only one class variable exists and it doesn't matter how you access it. Either way, you gain access to the same ***Date*** *object whose reference is stored in the class variable. Thus, the same date and time is shown in both cases.)*

Another new object

If you examine the code in Listing 13 (p. 726) near the end of the program, you will see that an additional five-second delay is introduced at this point in the program.

Following that delay, the code in Listing 9 (p. 718) instantiates another new object of the class named **MyClass01** , and stores the object's reference in a new reference variable named **ref2** .

(The object referred to by ***ref1*** *is a different object than the object referred to by* ***ref2*** *. Each object has its own instance variable named* ***v2*** *, and in this case, each instance variable is initialized to instantiate and refer to a new* ***Date*** *object when the new* ***MyClass01*** *object is instantiated.)*

Listing 9: Another new object .

```
MyClass01 ref2 = new MyClass01();
```

5.90

Display the date and time

Then, the code in Listing 10 (p. 719) causes the contents of the **Date** object referred to by the instance variable named **v2** in the second object of the class named **MyClass01** to be displayed.

Listing 10: Display the date and time .

```
System.out.println(ref2.v2);
```

5.91

This caused the output shown in Image 4 (p. 719) to be displayed on my computer screen when I ran the program.

(Once again, you will get different results if you compile and run the program because the date and time shown is the date and time that you run the program.)

Image 4: A new date and time.

```
Mon Sep 17 09:52:37 CDT 2001
```

5.92

Five seconds later

As you have probably figured out by now, the time encapsulated in this **Date** object is five seconds later than the time encapsulated in the **Date** object displayed in Image 2 (p. 717) . This is because the program was put to sleep for five seconds between the instantiation of the two objects referred to by **ref1** and **ref2** .

Every object has one

Every object instantiated from a given class has its own copy of each instance variable declared in the class definition. There is no sharing of instance variables among objects.

Each instance variable comes into existence when the object to which it belongs comes into existence, and ceases to exist when the object to which it belongs ceases to exist.

Eligible for garbage collection

If the instance variables are reference variables holding references to other objects, as is the case here, and if there are no other reference variables holding references to those same objects, the secondary objects cease to exist when the primary objects cease to exist.

Technically, the objects may not actually cease to exist. Technically they become eligible for garbage collection, which means that the memory that they occupy becomes eligible for reuse. However, as a practical matter, they cease to exist insofar as the program is concerned because they are no longer accessible.

A five-second difference in the time of creation

Since the two objects referred to by **ref1** and **ref2** came into existence with a five-second delay, the **Date** objects belonging to those two object reflect a five-second difference in the time encapsulated in the objects.

Only one copy of class variable exists

Also remember that if a variable is a class variable, only one copy of the variable exists, and all objects instantiated from the class share that one copy.

This is illustrated by the code in Listing 11 (p. 720) , which uses the reference to the second object instantiated from the class named **MyClass01** , to cause the contents of the class variable named **v1** to be displayed.

Listing 11: Display date information.

```
    System.out.println(ref2.v1);
}//end main
```

5.93

The output produced by the code in Listing 11 (p. 720) is shown in Image 5 (p. 720) .

Image 5: Same date and time as before.

```
Mon Sep 17 09:52:27 CDT 2001
```

5.94

Same output as before

As you can see, this is the same as the output shown in Image 1 (p. 715) and Image 3 (p. 718) earlier.

Accessing the same physical class variable

Since only one class variable named **v1** exists, and all objects instantiated from the class named **MyClass01** share that single copy, it doesn't matter whether you access the class variable using the name of the class, or access it using a reference to either of the objects instantiated from the class. In all three cases, you are accessing the same physical class variable.

Since nothing was done to cause the contents of the class variable to change after it came into existence and was initialized, Image 1 (p. 715) , Image 3 (p. 718) , and Image 5 (p. 720) are simply three different displays of the date and time encapsulated in the same **Date** object whose reference is stored in the class variable.

Let's revisit System.out.println...

Now, I want to revisit the statement originally shown in Listing 8 (p. 718) and repeated in Listing 12 (p. 721) for viewing convenience.

Listing 12: Revisiting System.out.println.

```
System.out.println(ref1.v1);
```

5.95

Java programmer wanted

I sometimes tell my students that if I were out in industry interviewing prospective Java programmers, my first question would be to ask the prospective employee to tell me everything that she knows about the statement in Listing 12 (p. 721) .

Covers a lot of Java OOP technology

This is not because there is a great demand for the use of this statement in real-world problems. *(In fact, in a GUI-driven software product world, there is probably very little demand for the use of this statement.)* Rather, it is because a lot of Java object-oriented technology is embodied in this single statement.

In that scenario, I would expect to receive a verbal dissertation of fifteen to twenty minutes in length to cover all the important points.

The short version

Let me give you the short version. There is a class named **System** . The **System** class declares three **static** *(class)* variables having the following types, names, and modifiers:

- public static final PrintStream **out**
- public static final InputStream **in**
- public static final PrintStream **err**

(Note that these class variables are also declared ***final*** *, causing them to behave as constants.)*

A **ccess the out variable without an object**

Because **out** is a class variable, **System.out** returns the contents of the class variable named **out** *(an object of the* ***System*** *class is not required in order to access a class variable of the* ***System*** *class).*

In general, *(ignoring the possibility of subclasses and interfaces)* because **out** is a reference variable of type **PrintStream** , the returned value must either be **null** *(no object reference)* or a reference to a valid **PrintStream** object.

Object of the PrintStream class

When the Java Virtual Machine starts an application running, it automatically instantiates an object of the **PrintStream** class and connects it to the **standard output device** . *(By default, the standard output device is typically the computer screen, but it can be redirected at the operating system level to be some other device. The following discussion assumes that the screen is the standard output device.)*

Assign object's reference to out variable

When the **PrintStream** object is instantiated by the virtual machine, the object's reference is assigned to the class variable of the **System** class named **out** . *(Because the variable named* ***out*** *is final, the contents of the variable cannot be modified later.)*

Reference to a PrintStream object

Therefore, the expression **System.out** returns a reference to the **PrintStream** object, which is connected to the standard output device.

Many instance methods

An object of the **PrintStream** class contains many instance methods. This includes numerous overloaded versions of a method named **println** . The signature of one of those overloaded **versions of the println method follows** :

```
public void println(Object x)
```

Textual representation of an object

The purpose of this overloaded version of the **println** method is to:

- Create a textual representation of the object referred to by the incoming parameter of type **Object** *(because **Object** is a totally generic type, this version of the **println** method can accept an incoming parameter that is a reference to any type of object)*
- Send that textual representation to the output device

In general...

A new **PrintStream** object can be connected to a variety of output devices when it is instantiated. However, in the special case of the **PrintStream** object instantiated by the virtual machine when the program starts, whose reference is stored in the class variable named **out** of the **System** class, the purpose of the object is to provide a display path to the standard output device.

Our old friend, the toString method

To accomplish this, the code in the version of the **println** method shown above (p. 721) calls the **toString** method on the incoming reference. *(I discussed the **toString** method in detail in earlier modules in this collection.)* The **toString** method may, or may not, have been overridden in the definition of the class from which the object was instantiated, or in some superclass of the class from which the object was instantiated.

Default version of toString

If not overridden, the default version of the **toString** method defined in the **Object** class is used to produce a textual representation of the object. As we learned in an earlier module, that textual representation looks something like the following:

```
ClassName@HexHashCode
```

Overridden version of toString method

If the class from which the object was instantiated *(or some superclass of that class)* contains an overridden version of the **toString** method, runtime polymorphism kicks in and the overridden version of the method is executed to produce the textual representation of the object.

The Date class overrides toString

In the case of this sample program, the object was instantiated from the **Date** class. The **Date** class does override the **toString** method.

When the overridden **toString** method is called on a **Date** object's reference, the **String** returned by the method looks something like that shown in Image 6 (p. 722) .

Image 6: Output from overridden toString method in Date class.

```
Mon Sep 17 09:52:27 CDT 2001
```

5.96

You will recall that this is the output that was produced by the code shown in Listing 8 (p. 718) and Listing 12 (p. 721) .

More than you ever wanted to know ...

And that is probably more than you ever wanted to know about the expression **System.out.println...**

.

It is also probably more than you ever wanted to know about class variables, class methods, instance variables, and instance methods.

Some cautions

Before leaving this topic, I do want to express some cautions. Basically, I want to suggest that you use ***static*** members very sparingly, if at all.

Static variables

To begin with, don't ever use **static** variables without declaring them **final** unless you understand exactly why you are declaring them **static** .

*(**static** **final** variables, or constants, are often very appropriate. See the fields in the **Color** class for example.)*

I can only think of only a very a few situations in which the use of a non-final **static** variable might be appropriate.

(One appropriate use might be to count the number of objects instantiated from a specific class.)

Static methods

Don't declare methods **static** if there is any requirement for the method to remember anything from one call to the next.

There are many appropriate uses for **static** methods, but in most cases, the purpose of the method will be to completely perform some action with no requirement to remember anything from that call to the next.

The method should probably also be self-contained. By this I mean that all information that the method needs to do its job should either come from incoming parameters or from **final** **static** member variables *(constants)*. The method probably should not depend on the values stored in non-final **static** member variables, which are subject to change over time.

*(A **static** method only has access to other **static** members of the class, so it cannot depend on instance variables defined in the class.)*

An appropriate example of a **static** method is the **sqrt** method of the **Math** class. This method computes and *"Returns the correctly rounded positive square root of a double"* where the double value is provided as a parameter to the method. Each time the method is called, it completes its task and doesn't attempt to save any values from that call to the next. Furthermore, it gets all the information that it needs to do its job from an incoming parameter.

5.2.11.5 Summary

Added complexity

The existence of **static** members tends to break up the simple OOP structures that I have discussed in previous modules in this collection.

While **static** members can be useful in some situations, the existence of **static** members tends to complicate the overall object-oriented structure of Java.

Furthermore, the overuse of **static** members can lead to problems similar to those experienced in languages like C and C++ that support global variables and global functions.

The class named Class

I discussed the class named **Class** and how a *conceptual* object of type **Class** exists in memory following a reference to a specific class in the program code.

The **Class** object represents the referenced class in memory, and contains the **static** variables and **static** methods belonging to that class. *(It contains some other information as well, such as the name of the superclass.)*

Class members and instance members

Class variables and *class methods* are declared **static** *(declaring a member **static** in the class definition causes to be called a class member)* .

Instance variables and instance methods are not declared **static** .

Each object has its own copy ...

Every object instantiated from a given class has its own copy of each instance variable declared in the class definition. *(Instance variables are not shared among objects.)*

Every object instantiated from a given class acts like it has its own copy of every instance method declared in the class definition. *(Although instance methods are actually shared among objects in order to reduce the amount of memory required, they are shared in such a way that they don't appear to be shared.)*

Every object shares ...

Every object instantiated from a given class shares the same single copy of each class variable declared in the class definition. Similarly, every object instantiated from a given class shares the same copy of each class method.

Accessing an instance member

An instance variable or an instance method can only be accessed by using a reference to the object that owns it. Even then, it may or may not be accessible depending on the access modifier assigned by the programmer.

Accessing a class member

The single shared copy of a class variable or a class method can be accessed in either of two ways:

- Via a reference to any object instantiated from the class.
- By simply joining the name of the class to the name of the class variable or the class method.

Again, the variable or method may or may not be accessible, depending on the access modifiers assigned by the programmer.

When to use class variables

It is very often appropriate to use **final** **static** variables, as constants in your programs. It is rarely, if ever, appropriate to use non-final **static** variables in your programs. The use of non-final **static** variables should definitely be minimized.

When to use static methods

It is often appropriate to use **static** methods in your programs, provided there is no requirement for the method to remember anything from one call to the next. **Static** methods should be self-contained.

5.2.11.6 What's next?

The next module in this collection will address the special case of Array Objects.

5.2.11.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Static Members
- File: Java1620.htm
- Published: 04/24/02
- Revised: 01/01/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.11.8 Complete program listing

A complete listing of the sample program is shown in Listing 13 (p. 726) .

Listing 13: Complete program listing.

```
    /*File MyClass01.java
Copyright 2002, R.G.Baldwin

This program illustrates static
members of a class. Output is:

Static variable
Mon Sep 17 09:52:27 CDT 2001

Instance variable
Mon Sep 17 09:52:32 CDT 2001
Static variable
Mon Sep 17 09:52:27 CDT 2001


Instance variable
Mon Sep 17 09:52:37 CDT 2001
Static variable
Mon Sep 17 09:52:27 CDT 2001
************************************/
import java.util.Date;
class MyClass01{
 static Date v1 = new Date();
 Date v2 = new Date();

 public static void main(
 String[] args){
 //Display static variable
 System.out.println(
 "Static variable");
 System.out.println(MyClass01.v1);

 //Delay for five seconds
 try{
 Thread.currentThread().sleep(5000);
 }catch(InterruptedException e){}

 //Instantiate an object and
 // display instance variable
 MyClass01 ref1 = new MyClass01();
 System.out.println();//blank line
 System.out.println(
 "Instance variable");
 System.out.println(ref1.v2);

 //Now, display the static
 // variable using object reference
 System.out.println(
 "Static variable");
 System.out.println(ref1.v1);
```



```
 System.out.println();//blank line

 //Delay for five seconds
 try{
```

-end-

5.2.12 Java1622: Array Objects, Part 1[34]

5.2.12.1 Table of Contents

5.2.12.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.12.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.12.2.1.1 Listings

5.2.12.3 Preview

This module explains how array objects fit into the grand scheme of things in Object-Oriented Programming *(OOP)* using Java.

A different syntax is required to create array objects than the syntax normally used to create ordinary objects.

Array objects are accessed via references. Any of the methods of the **Object** class can be called on a reference to an array object.

Array objects encapsulate a group of variables. The variables don't have individual names. They are accessed using positive integer index values. The integer indices of a Java array object always extend from **0** to **(n-1)** where **n** is the **length** of the array encapsulated in the object.

[34] This content is available online at <http://cnx.org/content/m44198/1.3/>.

All array objects in Java encapsulate one-dimensional arrays. The component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects. This is the mechanism for creating *multi-dimensional* or *ragged* arrays in Java.

5.2.12.4 Discussion and sample code

Three kinds of objects

In an earlier module, I told you that from a conceptual viewpoint, there are at least three kinds of objects involved in a Java program:

- Ordinary objects
- Class objects
- Array objects

Ordinary objects

Most of the discussion up to that point in the collection dealt with what I have referred to in the above list as *ordinary objects* .

These are the objects that you instantiate in you code by applying the **new** operator to a constructor for a class in order to create a new instance *(object)* of that class.

Class objects

In that module, I emphasized that my discussion of **Class** objects was conceptual in nature and did not necessarily represent an actual implementation. I went on to discuss the class named **Class** , and discussed how the use of that class fits into the grand scheme of OOP in Java. I explained how the existence of *class variables* and *class methods* tends to complicate the rather simple OOP structure consisting only of ordinary objects.

Array objects

I haven't discussed *array objects* up to this point in this collection. That is the purpose of this module.

Also tends to complicate

The existence of array objects also tends to complicate the OOP structure of a Java program consisting only of ordinary objects. Even if you don't consider array objects to be a different kind of object, you must at least consider them to be a *special* kind of object. A completely different syntax is required to create array objects than the syntax normally used to instantiate ordinary objects.

References to array objects

Arrays are objects in Java *(at least, arrays are always encapsulated in objects).* Array objects are dynamically created. Like ordinary objects, array objects are accessed via references. The reference to an array object may be assigned to a reference variable whose type is specified as:

```
TypeName[]
```

For example, Listing 1 (p. 729) shows some unrelated declarations for variables that are capable of storing references to array objects.

Listing 1: Sample variable declarations for array objects.

```
    int[] x1;

Button[] x2;

Object[] x3;
```

5.98

Note the empty square brackets that are required in the variable declarations in Listing 1 (p. 729) .

The special case of type Object

In addition, a reference to an array object may be assigned to a reference variable of type **Object** as shown in Listing 2 (p. 729) .

Listing 2: The special case of type Object.

```
    Object x4;
```

5.99

Note that there are *no square brackets* in the statement in Listing 2.

What does this mean?

This means that like ordinary objects, a reference to an array object can be treated as type **Object** *(with no square brackets).*

This further means that any of the methods defined in the **Object** class *(such as the* ***toString*** *and* ***getClass*** *methods)* can be called on a reference to an array object.

The String representation of an array object's reference

For example, when the **toString** method is called on a reference to an array object containing data of type **int** , the resulting string will be similar to the following:

```
    [I@73d6a5
```

Pretty ugly, huh?

You may recognize this as being similar to the default **String** returned by calling the **toString** method on an ordinary object with the name of the class for the ordinary object being replaced by **[I** .

For example, the **String** returned by calling the **toString** method on an object of the class named **Array04** , *(with no overridden toString method),* looks something like the following.

```
    Array04@73d6a5
```

(Note that the hexadecimal numeric values following the @ in both of the above examples will change from one case to the next.)

Calling the getClass method

Similarly, calling the **getClass** method on references to arrays containing data of the types **Array04** , **Button** , and **int** , respectively, and then calling the **toString** method on the **Class** objects returned by the **getClass** method, produces the following:

```
    class [LArray04;
class [Ljava.awt.Button;
class [I
```

Complicating the OOP structure

I made the following statement in an earlier paragraph:

"The existence of array objects also tends to complicate the OOP structure of a Java program consisting only of ordinary objects."

Array object is not a subclass of class Object

An array object can be treated as type **Object** for purposes of calling the methods of the **Object** class on the reference to the array object. However, it would probably be misleading to say that an array object is instantiated from a subclass of the **Object** class.

The new operator and the constructor name

Ordinary objects are created by applying the **new** operator to the constructor for a class, where the name of the constructor is always the same as the name of the class. That is not the case with array objects. Array objects are created by applying the **new** operator to the name of the type of data to be encapsulated in the array object.

Passing parameters versus square-bracket notation

In addition, whereas the instantiation of ordinary objects involves parameters passed in parentheses, a square-bracket notation is used instead of parentheses to create an array object. The value in the square brackets specifies the **length** of the array.

Creating an array object

Array objects *(with default initialization values)* are created by applying the **new** operator to the name of the data type to be stored in the array, using a square-bracket notation. An example is shown by the right-hand portion of the first statement in Listing 3 (p. 730) .

Listing 3: Creating array objects.

```
    int[] x1 = new int[5];

int[] x2 = {1,2,3,4,5};
```

5.100

A five-element array object

The first statement in Listing 3 (p. 730) creates an array object capable of storing five values of type **int** . The statement also assigns the array object's reference to the newly-declared reference variable named **x1** .

Default initial values

Each element in the array is initialized to the default value zero.

(All array elements created in this manner receive a default initial value. Numeric primitive types receive an initial value of zero. Elements of type ***boolean*** *receive an initial value of* ***false*** *. Elements whose type is the name of a class or the name of an interface receive an initial value of* ***null*** *.)*

Explicit initialization

The second statement in Listing 3 (p. 730) also creates an array object capable of storing five values of type **int** , but in this case, the values in the elements are explicitly initialized to the values shown.

(Note that the ***new*** *operator is not used in the second statement in Listing 3. This is also a significant departure from the syntax used to instantiate ordinary objects.)*

This array object's reference is assigned to the reference variable named **x2** .

Note the empty square brackets in the variable declarations

The syntax of the type specification for the reference variable in each statement in Listing 3 (p. 730) is different from the syntax used in the type specification for either a primitive variable or an ordinary class type reference variable *(note the square brackets on the left in Listing 3)* . In Listing 3 (p. 730) , the type specifications indicate that each variable is capable of holding a reference to an array object.

The size of the array

Furthermore, the empty square brackets *(in the declaration of the reference variable)* indicate that the reference variable doesn't know *(and doesn't care)* about the size of the array to which it may refer. Each of the reference variables declared in Listing 3 (p. 730) can refer to a one-dimensional array object of any size. Also, each of the reference variables can refer to different array objects at different points in time during the execution of the program.

> NOTE: **The Array class** As an aside, let me mention that there is a class named **Array** , which provides **static** methods to dynamically create and access Java arrays. The use of the methods of this class makes it possible to handle arrays with a programming style similar to the programming style typically used with ordinary objects. However, the use of the methods of the **Array** class tends to require more programming effort than the square-bracket notation discussed in this module. I will discuss a sample program that illustrates the methods of the **Array** class in a future module.

Encapsulating a group of variables

As is the case with other languages that support arrays, array objects in Java encapsulate a group of variables.

Zero or more variables may be encapsulated in an array object. If the number is zero, the array object is said to be empty.

(An example of an empty array object is the ***String[]*** *array passed to the* ***main*** *method in a Java application when the user doesn't enter any arguments at the command line.)*

No individual names

Also, as with other languages that support arrays, the variables encapsulated in an array object don't have individual names. Rather, they are referenced using positive integer index values.

(Typically, in Java, the index is placed in square brackets, which are applied to the name of the reference variable holding a reference to the array object.)

Elements or components?

It is common in the literature to refer to the variables that make up an array as its *elements* . However, the Java specification refers to them as *components.* The specification ascribes a different meaning to the word element, as shown in the following quotation from the specification:

"The value of an array component of type ***float*** *is always an element of the* ***float*** *value set ...; similarly, the value of an array component of type* ***double*** *is always an element of the* ***double*** *value set."*

Another quotation from Sun *(shown later in this module)* provides a somewhat clearer distinction between the words *component* and *element* .

(However, from force of habit, I will probability use component and element interchangeably in this module.)

The length of an array

If an array has **n** components, the **length** of the array is **n** . The components of the array are referenced using integer indices from 0 to (n - 1), inclusive.

Another quotation from Sun

Here is another quotation from the Java specification that explains the type specifications for the variable declarations in Listing 1 (p. 729) and Listing 3 (p. 730) .

"All the components of an array have the same type, called the component type of the array. If the component type of an array is T, then the type of the array itself is written T[]."

Components may be of an array type

As of the time that this object was originally written, all array objects in Java encapsulate one-dimensional arrays *(I have read that this may change in the future).*

The component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects.

Multi-dimensional or ragged arrays

One way to think of this is to think of the second level of array objects as being sub-arrays of the original array object. This construct can be used to create multi-dimensional array structures.

(The geometry of such multi-dimensional array structures is not constrained to be rectangles, cubes, etc., as is the requirement in many other languages. Some authors may refer to this as ragged arrays.)

Tree structures

This process of having the components of an array contain references to sub-arrays can be continued indefinitely *(well, maybe not indefinitely, but further than I care to contemplate).*

(This can be thought of as a tree structure where each array object containing references to other array objects is a node in the tree.)

The leaves of the tree

Eventually, the components *(the leaves of the tree structure)* must refer to a component type that is not an array type. According to Sun:

"... this is called the element type of the original array, and the ***components*** *at this level of the data structure are called the* ***elements*** *of the original array."*

Component versus element

Hopefully, the above quotation provides a somewhat clearer distinction between the use of the words *component* and *element* than was presented earlier.

Generic references

The reference to any array object can also be assigned to reference variables of the types **Object** , **Cloneable** , or **Serializable** .

(***Object*** *is the class at the top of the inheritance hierarchy.* ***Cloneable*** *and* ***Serializable*** *are interfaces, which are implemented by all array objects. Thus, a reference to an array object can be treated as any of these three types.)*

Generic array objects

Therefore, if the element type of an array object is one of these types, the elements in the array can refer to:

- Other array objects
- Ordinary objects
- A mixture of the two

This is illustrated in the sample program named **Array05** shown in Listing 9 (p. 738) near the end of the module.

Will explain in fragments

I will explain this program in fragments. Listing 4 (p. 733) shows the beginning of the controlling class and the beginning of the **main** method for the program named **Array05** ..

Listing 4: The beginning of the class named Array05.

```
public class Array05{
  public static void main(String[] args){
    int[] v1 = {1,2,3,4,5};
    Object[] v2 = new Object[2];
```

5.101

Listing 4 (p. 733) creates two array objects.

An array of type int

The first array object is a five-element array of element type **int**, with the element values initialized as shown by the values within the curly brackets. The reference to this array object is assigned to the reference variable named **v1**.

An array of element type Object

The second array object is a two-element array of element type **Object**, with each of the element values initialized to their default value of **null**. The reference to the array object is assigned to the reference variable named **v2**.

(Note that unlike the previous discussion of ***Object****, the declaration of the reference variable in this case does include empty square brackets. I will have more to say about this later.)*

A new object of this class

Listing 5 (p. 733) creates a new *ordinary object* of class **Array05**. The code assigns the object's reference to the first element in the array object of element type **Object**, referred to by the reference variable named **v2**.

Listing 5: A new ordinary object of class Array05.

```
    v2[0] = new Array05();
```

5.102

This is allowable because the reference to an object of any class can be assigned to a reference variable of type **Object**.

(The array object referred to by ***v2*** *contains two elements, each of which is a reference variable of type* ***Object****.)*

Populate the second element

The code in Listing 6 (p. 734) assigns the reference to the existing array object of the element type **int** to the second element in the array object of element type **Object**.

Listing 6: Populate the second element.

```
v2[1] = v1;
```

5.103

This is allowable because a reference to any array object can be assigned to a reference variable of type **Object** .

Array contains two references

At this point, the array object of element type **Object** contains two references.

(Each of the elements in an array of the declared type ***Object[]*** *is a reference of type* ***Object*** *.)*

The first element refers to an ordinary object of the class **Array05** .

The second element refers to an array object of type **int** , having five elements, populated with the integer values of 1 through 5 inclusive.

(Note that this is not a multi-dimensional array in the traditional sense. I will discuss the Java approach to such multi-dimensional arrays in the next module. This is simply a generic array of element type ***Object*** *, one element of which happens to contain a reference to an array object of type* ***int*** *.)*

Print some data

The code in Listing 7 (p. 734) passes each of the references to the **println** method of the **PrintStream** class.

Listing 7: Print some data.

```
System.out.println(v2[0]);
System.out.println(v2[1]);
```

5.104

The **println** method causes the **toString** method to be called on each reference. The **String** returned by the **toString** method is displayed on the computer screen in each case.

This is allowable because any method defined in the **Object** class *(including the* ***toString*** *method)* can be called on any reference stored in a reference variable of type **Object** .

This is true regardless of whether that reference is a reference to an ordinary object or a reference to an array object.

The output

Listing 7 (p. 734) causes the following two lines of text to be displayed:

```
Array05@73d6a5
[I@111f71
```

Pretty ugly, huh?

In both cases, this is the value of the **String** returned by the default version of the **toString** method defined in the **Object** class. Here is what Sun has to say about that default behavior:

"Returns a string representation of the object. In general, the ***toString*** *method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.*

The ***toString*** *method for class* ***Object*** *returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object."*

Doesn't address array objects

Obviously, this description of behavior doesn't address the case where the object is an array object, unless the characters **[I** are considered to be the name of a class. *(I will have a little more to say about this later.)*

Produce some more output

Finally, Listing 8 (p. 735) shows the last statement in this simple program.

Listing 8: Produce some more output.

```
    System.out.println( ( (int[])v2[1] )[4] );
  }//end main
}//end class Array05
```

5.105

What does this mean?

As you can see, the syntax of this statement is pretty ugly.

Values are accessed from an array object by following the array's reference with a pair of square brackets containing an integer index value as follows:

```
v2[1]
```

Get the value at index 1 as type Object

This code begins by accessing the component at index value 1 of the array object referred to by the reference variable named **v2** .

The value retrieved is a reference, and is retrieved as type **Object** , *(because the variable named* ***v2*** *was declared to be of type* ***Object[]*** *).*

A cast is required

A cast is used to convert from type **Object[]** to type **int[]** using the following code:

```
(int[])
```

This produces a reference to an array object capable of containing values of type **int** .

Apply index to the int array

After the type of the reference has been converted, the accessor **[4]** is applied to the reference. This causes the **int** value stored in the array object of type **int** *(at index value 4)* to be returned.

(If you refer back to Listing 4 (p. 733) , you will see that the integer value 5 was stored in the element at index value 4 of this array object.)

You should try to remember this syntax and compare it with the syntax used in the Java approach to traditional multi-dimensional arrays, which I will discuss in the next module.

The output

Thus, the code in Listing 8 (p. 735) causes the number 5 to be displayed on the computer screen.

Let's recap

To recap, the program named **Array05** creates a two-element array object capable of storing references of type **Object** .

Object is generic

Because **Object** is a completely generic type, each of the elements in the array is capable of storing a reference to any ordinary object, or storing a reference to any array object.

Store reference to ordinary object in generic array

The first element in the array is populated with a reference to an ordinary object instantiated from the class named **Array05** .

(Important: The actual object does not occupy the array element. Rather, the actual object exists someplace else in memory, and a reference to the object occupies the array element.)

Store a reference to an array object in the generic array

The second element in the array of element type **Object** is populated with a reference to another array object capable of containing elements of type **int** .

As above, the actual array object of type **int** does not occupy the second element. Rather, that array object exists someplace else in memory, and a reference to the array object occupies the second element in the array of element type **Object** .

Display some data

After the array object of element type **Object** is created and populated, three print statements are executed to display information about the array object and its contents *(those print statements are shown in Listing 7 (p. 734) and* Listing 8 (p. 735) *)*.

The print statements produce the following output on the computer screen:

```
Array05@73d6a5
[I@111f71
5
```

Default textual representation of ordinary object

The first line of output is the default textual representation of the ordinary object, achieved by calling the default **toString** method on the reference to the ordinary object.

Default textual representation of array object

The second line of output is the textual representation of the array object of type **int[]** , achieved by calling the default **toString** method on the reference to the array object.

Primitive value stored in array object

The third line of text is the value stored in element index 4 of the **int[]** array object whose reference is stored in element index 1 of the array object of element type **Object** .

Primitive versus non-primitive array element contents

References to objects are stored in the elements of non-primitive array objects. The objects themselves exist somewhere else in memory.

Actual primitive values are stored in the elements of a primitive array object.

Thus, the elements of an array object contain actual primitive values, null references, or actual references to ordinary or array objects, depending on the type of the elements of the array object.

5.2.12.5 Summary

This module begins the discussion of array objects in Java.

The existence of array objects tends to complicate the OOP structure of a Java program otherwise consisting only of ordinary objects.

A completely different syntax is required to create array objects than the syntax normally used to instantiate ordinary objects. Ordinary objects are normally instantiated by applying the **new** operator to the constructor for the target class passing parameters between a pair of matching parentheses.

Array objects *(with default initialization)* are created using the **new** operator, the type of data to be encapsulated in the array, and a square-bracket notation to specify the **length** of the array encapsulated in the object.

Array objects with explicit initialization are created using a comma-separated list of expressions enclosed in curly brackets.

Arrays in Java are objects, which are dynamically created and allocated to dynamic memory.

Like ordinary objects, array objects are accessed via references. The type of such a reference is considered to be **TypeName[]** *(note the empty square brackets in the type specification).*

A reference to an array object can also be assigned to a reference variable of type **Object** *(note the absence of square brackets).* Thus, any of the methods of the **Object** class can be called on a reference to an array object.

As is the case with other languages that support arrays, array objects in Java encapsulate a group of zero or more variables. The variables encapsulated in an array object don't have individual names. Rather, they are accessed using positive integer index values.

The integer indices of a Java array object always extend from **0** to **(n-1)** where **n** is the **length** of the array object.

As of the time of this writing, all array objects in Java encapsulate one-dimensional arrays. However, the component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects. This is the mechanism for creating *multi-dimensional* or *ragged* arrays in Java.

The reference to any array object can be assigned to reference variables of the types **Object** , **Cloneable** , or **Serializable** . If the element type of an array object is one of these types, the elements in the array can refer to:

- Other array objects
- Ordinary objects
- A mixture of the two

5.2.12.6 What's next?

This module has barely scratched the surface in explaining how array objects fit into the grand scheme of things in OOP using Java. In the next module, I will continue the discussion, showing you some of the *(often complex)* aspects of using Java array objects to emulate traditional *multi-dimensional* arrays. I will also show you how to create *ragged* arrays in Java.

5.2.12.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 1
- File: Java1622.htm
- Published: 05/15/02
- Revised: 01/01/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.12.8 Complete program listing

A complete listing of the program is shown in Listing 9 (p. 738) below.

Listing 9: Complete program listing.

```
/*File Array05.java
Copyright 2002, R.G.Baldwin

This program illustrates storage of
references to ordinary objects and
references to array objects in the
same array object of type Object.

Program output is:

Array05@73d6a5
[I@111f71
5
**************************************/

public class Array05{
  public static void main(
                        String[] args){

    int[] v1 = {1,2,3,4,5};
    Object[] v2 = new Object[2];
    v2[0] = new Array05();
    v2[1] = v1;

    System.out.println(v2[0]);
    System.out.println(v2[1]);
    System.out.println(
                    ((int[])v2[1])[4]);
  }//end main
}//end class Array05
```

-end-

5.2.13 Java1624: Array Objects, Part 2[35]

5.2.13.1 Table of Contents

5.2.13.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.13.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.13.2.1.1 Listings

[35] This content is available online at <http://cnx.org/content/m44199/1.3/>.

- Listing 23 (p. 757) . Beginning of a more general case.
- Listing 24 (p. 757) . Populate the leaf array elements.
- Listing 25 (p. 758) . Display the output.
- Listing 26 (p. 761) . Complete program listing.

5.2.13.3 Preview

This module explains various details regarding the use of array objects in Java, and illustrates many of those details through the use of sample code.

A sample program shows you three ways to emulate traditional two-dimensional rectangular arrays, and also shows you how to create and use ragged arrays.

5.2.13.4 Discussion and sample code

Array objects

A different syntax is required to create array objects than the syntax normally used to create ordinary objects.

Array objects are accessed via references.

Any of the methods of the **Object** class can be called on a reference to an array object.

The indices of a Java array object

Array objects encapsulate a group of variables, which don't have individual names. They are accessed using positive integer index values. The integer indices of a Java array object always extend from **0** to **(n-1)** where **n** is the **length** of the array encapsulated in the object.

Multidimensional arrays

Array objects in Java encapsulate one-dimensional arrays. However, the component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects. This is the mechanism for creating *multi-dimensional* or *ragged* arrays in Java.

Such a structure of array objects can be thought of as a tree of array objects, with the data being stored in the array objects that make up the leaves of the tree.

Array types

When declaring a reference variable capable of referring to an array object, the array type is declared by writing the name of an element type followed by some number of empty pairs of square brackets []. This is illustrated in Listing 1 (p. 740) , which declares a reference variable named **v1** , capable of storing a reference to a *two-dimensional* array of type **int** .

Listing 1: Reference variable declaration.

```
int[][] v1;
```

5.107

---

*(Note that Listing 1 doesn't really declare a two-dimensional array in the traditional sense of other programming languages. Rather, it declares a reference variable capable of storing a reference to a one-dimensional array object, which in turn is capable of storing references to one-dimensional array objects of type **int** .)*

Multiple pairs of square brackets are allowed

The components in an array object may refer to other array objects. The number of bracket pairs used in the declaration of the reference variable indicates the depth of array nesting *(in the sense that array elements can refer to other array objects).* This is one of the ways that Java implements the concept of traditional multi-dimensional arrays *(I will show you some other ways later in this module).*

The code in Listing 1 shows two levels of nesting for the reference variable of type

```
int[][]
```

Length not part of variable declaration

Note that an array's length is not part of its type or reference variable declaration.

Ragged arrays

Note also that multi-dimensional arrays, when implemented in this fashion, are not required to represent rectangles, cubes, etc. For example, the number of elements in each row of a Java two-dimensional array can be different. Some authors refer to this as a *ragged array* .

Allowable types

The specified element type of an array may be any primitive or reference type. Note, however, that all elements of the array must be of the same type *(consistent with the type-conversion rules discussed below)* .

Listing 2 (p. 741) shows the declaration of a reference variable capable of referring to a three *-dimensional* array object of element type **Button** (***Button*** *is one of the classes in the standard class library).*

Listing 2: A three-dimensional array object of element type Button.

```
Button[][][] v2;
```

Rules of type conversion and assignment compatibility apply

The normal rules of *type conversion* and *assignment compatibility* apply when creating and populating array objects. For example, if the specified type is the name of a non-abstract class, a null reference or a reference to any object instantiated from that class or any subclass of that class may be stored in the array element.

The generic class Object

For example, Listing 3 (p. 741) shows the declaration of a reference variable capable of referring to a one-dimensional array object of element type **Object** .

Since **Object** is the superclass of all other classes, this array object is capable of storing references to objects instantiated from any other class. *(As we saw in the previous module, it is also capable of storing a reference to any other array object as well.)*

Listing 3: The generic class Object.

```
Object[] v3;
```

Primitive type conversions

Similarly, if the declared element type for the array object is one of the primitive types, the elements of the array can be used to store values of any primitive type that is *assignment compatible* with the declared type *(without the requirement for a cast).*

For example, the code in Listing 4 (p. 742) shows the creation of a one-dimensional array object capable of storing values of type **int** . The array object has a length of 3 elements, and the object's reference is stored in a reference variable named **v1** .

Listing 4: Primitive type conversions.

```
int[] v1;
v1 = new int[3];
byte x1 = 127;
short x2 = 16384;
int x3 = 32000;
v1[0] = x1;
v1[1] = x2;
v1[2] = x3;
```

5.110

Assignment-compatible assignments

Values of the types **byte** , **short** , and **int** , are stored in the elements of the array object in Listing 4 (p. 742) .

Actual type is lost in the process

It should be noted that the **byte** and **short** values are converted to type **int** as they are stored. When retrieved later, they will be retrieved as type **int** . Any indication that these values were ever of any type other than **int** is lost in the process of storing and retrieving the values.

What about class types?

If the declared element type is the name of a class, *(which may or may not be abstract),* a null reference or a reference to any object instantiated from the class or any subclass of the class may be stored in the array element.

(Obviously you can't store a reference to an object instantiated from an abstract class, because you can't instantiate an abstract class.)

What about an interface type?

If the declared element type is an interface type, a null reference or a reference to any object instantiated from any class that implements the interface can be stored in the array element.

(This is an extremely powerful concept, allowing references to objects instantiated from many different classes to be collected into an array as the interface type.)

Array reference variables

All array objects are accessed via references. A reference variable whose declared type is an array type *does not contain an array.* Rather, it contains either null, or a reference to an array object.

Allocation of memory

Declaring the reference variable does not create an array, nor does it allocate any space for the array components. It simply causes memory to be allocated for the reference variable itself, which may later contain a reference to an array object.

Initialization

In the same sense that it is possible to declare a reference variable for an ordinary object, and initialize it with a reference to an object when it is declared, it is also possible to declare a reference to an array object and initialize it with a reference to an array object when it is declared. This is illustrated in Listing 5 (p. 743) , which shows the following operations combined into a single statement:

- Declaration of a variable to contain a reference to an array object
- Creation of the array object
- Storage of the array object's reference in the reference variable

---

Listing 5: Initialization.

```
int[] v1 = new int[3];
```

5.111

Can refer to different array objects

The **length** of an array is not established when the reference variable is declared. As with references to ordinary objects, a reference to an array object can refer to different array objects at different points in the execution of a program.

For example, a reference variable that is capable of referring to an array of type **int[]** can refer to an array object of a given **length** at one point in the program and can refer to a different array object of the same type but a different **length** later in the program.

Placement of square brackets

When declaring an array reference variable, the square brackets [] may appear as part of the type, or following the variable name, or both. This is illustrated in Listing 6 (p. 743) .

Listing 6: Placement of square brackets.

```
int[][] v1;
int[] v2[];
int v3[][];
```

5.112

Type and length

Once an array object is created, its type and length never changes. A reference to a different array object must be assigned to the reference variable to cause the reference variable to refer to an array of different length.

Creating the actual array object

An array object is created by an array creation expression or an array initializer.

An array creation expression *(or an array initializer)* specifies:

- The element type
- The number of levels of nested arrays
- The length of the array for at least one of the levels of nesting

Two valid array creation expressions are illustrated by the statements in Listing 7 (p. 744) .

Listing 7: Creating the actual array object.

```
int[][] v1;
int[] v2[];

v1 = new int[2][3];
v2 = new int[10][];
```

5.113

A two-dimensional rectangular array

The third statement in Listing 7 (p. 744) creates an array object of element type **int** with two levels of nesting. This array object can be thought of as a traditional two-dimensional rectangular array having two rows and three columns. *(This is a somewhat arbitrary choice as to which dimension specifies the number of rows and which dimension specifies the number of columns. You may prefer to reverse the two.)*

A ragged array

The fourth statement also creates an array object of element type **int** with two levels of nesting. However, the number of elements in each column is not specified at this point, and it is not appropriate to think of this as a two-dimensional rectangular array. In fact, once the number of elements in each column has been specified, it may not describe a rectangle at all. Some authors refer to an array of this type as a *ragged array.*

The length of the array

The **length** of the array is always available as a final instance variable named **length** . I will show you how to use the value of **length** in a sample program later in this module.

Accessing array elements

An array element is accessed by an *array access expression* . The access expression consists of an expression whose value is an array reference followed by an indexing expression enclosed by matching square brackets.

The expression in parentheses in Listing 8 (p. 745) illustrates an array access expression *(or perhaps two concatenated array access expressions).*

Listing 8: An array access expression.

```
int[][] v1 = new int[2][3];
System.out.println(v1[0][1]);
```

5.114

First-level access

This *array access expression* first accesses the contents of the element at index 0 in the array object referred to by the reference variable named **v1** . This element contains a reference to a second array object *(note the double matching square brackets, [[[in the declaration of the variable named* ***v1*** *).*

Second-level access

The *array access expression* in Listing 8 (p. 745) uses that reference to access the value stored in the element at index value 1 in the second array object. That value is then passed to the **println** method for display on the standard output device.

(In this case, the value 0 is displayed, because array elements are automatically initialized to default values when the array object is created. The default value for all primitive numeric values is zero.)

Zero-based indexing

All array indexes in Java begin with **0** . An array with length **n** can be indexed by the integers **0** to **(n-1)** . Array accesses are checked at runtime. If an attempt is made to access the array with any other index value, an **ArrayIndexOutOfBoundsException** will be thrown.

Index value types

Arrays must be indexed by integer values of the following types: **int** , **short** , **byte** , or **char** . For any of these types other than **int** , the value will be promoted to an **int** and used as the index.

An array cannot be accessed using an index of type **long** . Attempting to do so results in a compiler error.

Default initialization

If the elements in an array are not purposely initialized when the array is created, the array elements will be automatically initialized with default values. The default values are:

- All reference types: null
- Primitive numeric types: 0
- Primitive boolean type: false
- Primitive char type: the Unicode character with 16 zero-valued bits

Explicit initialization of array elements

The values in the array elements may be purposely initialized when the array object is created using a comma-separated list of expressions enclosed by matching curly brackets. This is illustrated in Listing 9 (p. 746) .

Listing 9: Explicit initialization of array elements.

```
int[] v1 = {1,2,3,4,5};
```

5.115

No new operator

Note that this format does not use the **new** operator. Also note that the expressions in the list may be much more complex than the simple literal values shown in Listing 9 (p. 746) .

Length and order

When this format is used, the length of the constructed array will equal the number of expressions in the list.

The expressions in an array initializer are executed from left to right in the order that they occur in the source code. The first expression specifies the value at index value zero, and the last expression specifies the value at index value n-1 *(where n is the length of the array).*

Each expression must be assignment-compatible with the array's component type, or a compiler error will occur.

A sample program

The previous paragraphs in this module have explained some of the rules and characteristics regarding array objects. They have also illustrated some of the syntax involved in the use of array objects in Java.

More powerful and complex

Many aspiring Java programmers find the use of array objects to be something less than straightforward, and that is understandable. In fact, Java array objects are somewhat more powerful than array structures in many other programming languages, and this power often manifests itself in additional complexity.

A traditional two-dimensional rectangular array

Some of that complexity is illustrated by the program named **Array07** , shown in Listing 26 (p. 761) near the end of this module. This program illustrates three different ways to accomplish essentially the same task using array objects in Java. That task is to emulate a traditional two-dimensional rectangular array as found in other programming languages. Two of the ways that are illustrated are essentially ragged arrays with sub-arrays having equal length.

Ragged arrays

The program also illustrates two different ways to work with array objects and ragged arrays.

Will discuss in fragments

As is my practice, I will discuss and explain the program in fragments.

All of the interesting code in this program is contained in the **main** method, so I will begin my discussion with the first statement in the **main** method.

Create a two-dimensional rectangular array structure

Listing 10 (p. 747) creates an array structure that emulates a traditional rectangular array with two rows and three columns.

Listing 10: Create a two-dimensional rectangular array structure.

```
Object[][] v1 = new Object[2][3];
```

5.116

(Note that unlike the ragged array structures to be discussed later, this approach requires all rows to be the same length and all columns to be the same length.)

Reference variable declaration

The code to the left of the equal sign *(=)* in Listing 10 (p. 747) declares a reference variable named **v1** . This reference variable is capable of holding a reference to an array object whose elements are of the type **Object[]** ..

In other words, this reference variable is capable of

- holding a reference to an array object,
- whose elements are capable of holding references to other array objects,
- whose elements are of type **Object** .

Two levels of nesting

The existence of double matching square brackets in the variable declaration in Listing 10 (p. 747) indicates two levels of nesting.

Restrictions

The elements in the array object referred to by **v1** can only hold references to other array objects whose element type is **Object** *(or references to array objects whose element type is a subclass of* ***Object*** *).*

The elements in the array object referred to by **v1** cannot hold references to ordinary objects instantiated from classes, or array objects whose element type is a primitive type.

In other words, the elements in the array object referred to by **v1** can only hold references to other array objects. The element types of those array objects must be *assignment compatible* with the type **Object** *(this includes interface types and class types but not primitive types).*

A tree of empty array objects

The code to the right of the equal sign *(=)* in Listing 10 (p. 747) creates a *tree structure* of array objects. The object at the root of the tree is an array object of type **Object[]** , having two elements *(a* ***length*** *of two).*

The reference variable named **v1** refers to the array object that forms the root of the tree.

Each of the two elements in the root array object is initialized with a reference to another array object.

(These two objects might be viewed as sub-arrays, or as child nodes in the tree structure).

Each of the child nodes is an array object of type **Object** „ and has a **length** of three.

Each element in each of the two child node array objects is initialized to the value **null** *(this is the default initialization for array elements of reference types that don't yet refer to an object).*

Recap

To recap, the reference variable named **v1** contains a reference to a two-element, one-dimensional array object. Each element in that array object is capable of storing a reference of type **Object[]** *(a reference to another one-dimensional array object of type* ***Object*** *).*

Two sub-array objects

Two such one-dimensional sub-array *(or child node)* objects, of element type **Object** , are created. References to the two sub-array objects are stored in the elements of the two-element array object at the root of the tree.

Each of the sub-array objects has three elements. Each element is capable of storing a reference to an object as type **Object** .

The leaves of the tree

These two sub-array objects might be viewed as the leaves of the tree structure.

Initialize elements to null

However, the objects of type **Object** don't exist yet. Therefore, each element in each of the sub-array objects is automatically initialized to **null** .

Arrays versus sub-arrays

Note that there is no essential difference between an array object and a sub-array object in the above discussion. The use of the sub prefix is used to indicate that an ordinary array object belongs to another array object, because the reference to the sub-array object is stored in an element of the owner object.

Many dimensions are possible

Multi-dimensional arrays of any *(reasonable)* depth can be emulated in this manner. An array object may contain references to other array objects, which may contain references to other array objects, and so on.

The leaves of the tree structure

Eventually, however, the elements of the leaves in the tree structure must be specified to contain either primitive values or references to ordinary objects. This is where the data is actually stored.

(Note however, that if the leaves are specified to contain references of type ***Object*** *, they may contain references to other array objects of any type, and the actual data could be stored in those array objects.)*

The length of an array

Every array object contains a public final instance variable named **length** , which contains an integer value specifying the number of elements in the array.

Once created, the length of the array encapsulated in an array object cannot change. Therefore, the value of **length** specifies the length of the array throughout the lifetime of the array object.

Using length to populate the leaves of the tree structure

The value of **length** is very handy when processing array objects. This is illustrated in Listing 11 (p. 748) , which uses a nested **for** loop to populate the elements in the leaves of the tree structure referred to by **v1** . *(The elements in the leaf objects are populated with references to objects of type* ***Integer*** *.)*

Listing 11: Using length to populate the leaves of the tree structure.

```
for(int i=0;i<v1.length;i++){
  for(int j=0;j<v1[i].length;j++){
    v1[i][j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

5.117

Using length in loop's conditional expressions

Hopefully by now you can read and understand this code without a lot of help from me. I will point out, however, that the value returned by **v1.length** *(in the conditional expression for the outer loop)* is the number of leaves in the tree structure *(this tree structure has two leaves).*

I will also point out that the value returned by **v1[i].length** *(in the conditional expression for the inner loop)* is the number of elements in each leaf array object *(each leaf object in this tree structure has three elements).*

Finally, I will point out that the expression **v1[i][j]** accesses the ***jth*** element in the ***ith*** leaf, or sub-array. In the traditional sense of a rectangular array, this could be thought of as the ***jth*** column of the ***ith*** row. This mechanism is used to store object references in each element of each of the leaf array objects.

Populate with references to Integer objects

Thus, each element in each leaf array object is populated with a reference to an object of the type **Integer** . Each object of the type **Integer** encapsulates an **int** value calculated from the current values of the two loop counters.

Display leaf object contents

In a similar manner, the code in Listing 12 (p. 749) uses the **length** values in the conditional expressions of nested **for** loops to access the references stored in the elements of the leaf array objects, and to use those references to access and display the values encapsulated in the **Integer** objects whose references are stored in those elements.

Listing 12: Display leaf object contents.

```
    for(int i=0;i<v1.length;i++){
  for(int j=0;j<v1[i].length;j++){
    System.out.print(
                    v1[i][j] + " ");
  }//end inner loop
  System.out.println();//new line
 }//end outer loop
```

5.118

The rectangular output

The code in Listing 12 (p. 749) produces the following output on the screen.

```
 1 2 3
2 4 6
```

As you can see, this emulates a traditional two-dimensional array having two rows and three columns.

A ragged array with two rows and three columns

The second approach to emulating a traditional two-dimensional rectangular array will create a *ragged array* where each row is the same length.

(It is very important to note that, unlike this case, with a ragged array, the number of elements in each row or the number of elements in each column can be different.)

The most significant thing about this approach is the manner in which the tree of array objects is created *(see Listing 13 (p. 750)).*

Listing 13: Beginning of a ragged array with two rows and three columns.

```
Object[][] v2 = new Object[2][];
```

5.119

Three statements required

With this approach, three statements are required to replace one statement from the previous approach. *(Two additional statements are shown in Listing 14 (p. 750) .)*

A single statement in the previous approach *(Listing 10 (p. 747))* created all three array objects required to construct the tree of array objects, and initialized the elements in the leaf array objects with **null** values.

Create only the root array object

However, the boldface code in Listing 13 (p. 750) creates only the array object at the root of the tree. That array object is an array object having two elements capable of storing references of type **Object[]** .

Empty square brackets

If you compare this statement with the statement in Listing 10 (p. 747) , you will notice that the rightmost pair of square brackets in Listing 13 (p. 750) is empty. Thus, Listing 13 (p. 750) creates only the array object at the root of the tree, and initializes the elements in that array object with **null** values.

Leaf array objects don't exist yet

The leaf array objects don't exist at the completion of execution of the statement in Listing 13 (p. 750) .

Create the leaf array objects

The statements in Listing 14 (p. 750) create two array objects of element type **Object** .

Listing 14: Create the leaf array objects.

```
v2[0] = new Object[3];
v2[1] = new Object[3];
```

5.120

Save the references to the leaves

References to these two leaf objects are stored in the elements of the array object at the root of the tree, *(which was created in* Listing 13 (p. 750) *).* Thus, these two array objects become the leaves of the tree structure of array objects.

This completes the construction of the tree structure. Each element in each leaf object is initialized with **null** .

Why bother?

You might ask why I would bother to use this approach, which requires three statements in place of only one statement in the previous approach.

The answer is that I wouldn't normally use this approach if my objective were to emulate a traditional rectangular array. However, this approach is somewhat more powerful than the previous approach.

The lengths of the leaf objects can differ

With this approach, the **length** values of the two leaf array objects need not be the same. Although I caused the **length** value of the leaf objects to be the same in this case, I could just as easily have caused them to be different lengths *(I will illustrate this capability later in the program).*

Populate and display the data

If you examine the complete program in Listing 26 (p. 761) near the end of the module, you will see that nested **for** loops, along with the value of **length** was used to populate and display the contents of the leaf array objects. Since that portion of the code is the same as with the previous approach, I won't show and discuss it here.

The rectangular output

This approach produced the following output on the screen, *(which is the same as before):*

```
 1 2 3
2 4 6
```

Now for something really different

The next approach that I am going to show you for emulating a two-dimensional rectangular array is significantly different from either of the previous two approaches.

Not element type Object[]

In this approach, I will create a one-dimensional array object of element type **Object** *(not element type* ***Object[]*** *)* . I will populate the elements of that array object with references to other array objects of element type **Object** . In doing so, I will create a tree structure similar to those discussed above.

The length of the leaf objects

As with the second approach above, the array objects that make up the leaves of the tree can be any length, but I will make them the same length in order to emulate a traditional rectangular two-dimensional array.

Create the array object

First consider the statement shown in Listing 15 (p. 751) . Compare this statement with the statements shown earlier in Listing 10 (p. 747) and Listing 13 (p. 750) .

Listing 15: Create the array object.

```
Object[] v3 = new Object[2];
```

No double square brackets

Note in particular that the statement in Listing 15 (p. 751) does not make use of double square brackets, as was the case in Listing 10 (p. 747) and Listing 13 (p. 750) . Thus, the statement show in Listing 15 (p. 751) is entirely different from the statements shown in Listing 10 (p. 747) and Listing 13 (p. 750) .

Declare a reference variable

That portion of the statement to the left of the equal sign *(=)* declares a reference variable capable of storing a reference to an array object whose elements are capable of storing references of the type **Object** *(not type* ***Object[]*** *as in the previous examples).*

Refer to the root object

This reference variable will refer to an array object that forms the root of the tree structure. However, the root object in this case will be considerably different from the root objects in the previous two cases.

In the previous two cases, the elements of the root object were required to store references of type **Object[]** *(note the square brackets).* In other words, an array object whose elements are of type **Object[]** can only store references to other array objects whose elements are of type **Object** .

A more general approach

However, an array object whose elements are of type **Object** *(as is the case here),* can store:

- References to any object instantiated from any class
- References to array objects whose elements are of any type *(primitive or reference)*
- A mixture of the two kinds of references .

Therefore, this is a much more general, and much more powerful approach.

A price to pay

However, there is a price to pay for the increased generality and power. In particular, the programmer who uses this approach must have a much better understanding of Java object-oriented programming concepts than the programmer who uses the two approaches discussed earlier in this module.

Particularly true relative to first approach

This is particularly true relative to the first approach discussed earlier. That approach is sufficiently similar to the use of multi-dimensional arrays in other languages that a programmer with little understanding of Java object-oriented programming concepts can probably muddle through the syntax based on prior knowledge. However, it is unlikely that a programmer could muddle through this approach without really understanding what she is doing.

Won't illustrate true power

Although this approach is very general and very powerful, this sample program won't attempt to illustrate that power and generality. Rather, this sample program will use this approach to emulate a traditional two-dimensional rectangular array just like the first two approaches discussed earlier. *(Later, I will also use this approach for a couple of ragged arrays.)*

Populate the root object

The two statements in Listing 16 (p. 752) create two array objects, each having three elements. Each element is capable of storing a reference to any object that is *assignment compatible* with the **Object** type.

(Assignment compatibility includes a reference to any object instantiated from any class, or a reference to any array object of any type (including primitive types), or a mixture of the two.)

Listing 16: Populate the root object.

```
v3[0] = new Object[3];
v3[1] = new Object[3];
```

5.122

References to the two new array objects are stored in the elements of the array object that forms the root of the tree structure. The two new array objects form the leaves of the tree structure.

Populate the leaf array objects

As in the previous two cases, the code in Listing 17 (p. 753) uses nested **for** loops to populate the array elements in the leaf objects with references to new **Integer** objects. *(The **Integer** objects encapsulate **int** values based on the loop counter values for the outer and inner loops.)*

Listing 17: Populate the leaf array objects.

```
for(int i=0;i<v3.length;i++){
  for(int j=0;j<((Object[])v3[i]).length;j++){
    ((Object[])v3[i])[j] = new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

5.123

Added complexity

The added complexity of this approach manifests itself in

- The *cast operators* shown in boldface Italics in Listing 17 (p. 753)
- The attendant required grouping of terms within parentheses

Inside and outside the parentheses

Note that within the inner loop, one of the square-bracket accessor expressions is inside the parentheses and the other is outside the parentheses.

Why are the casts necessary?

The casts are necessary to convert the references retrieved from the array elements from type **Object** to type **Object[]** . For example, the reference stored in **v3[i]** is stored as type **Object** .

Get length of leaf array object

The cast in the following expression converts that reference to type **Object[]** before attempting to get the value of **length** belonging to the array object whose reference is stored there.

```
((Object[])v3[i]).length
```

Assign a value to an element in the leaf array object

Similarly, the following expression converts the reference stored in **v3[i]** from type **Object** to type **Object[]** . Having made the conversion, it then accesses the **jth** element of the array object whose reference is stored there *(in order to assign a value to that element).*

```
((Object[])v3[i])[j]=
```

Display data in leaf array objects

Listing 18 (p. 754) uses similar casts to get and display the values encapsulated in the **Integer** objects whose references are stored in the elements of the leaf array objects.

Listing 18: Display data in leaf array objects.

```
    for(int i=0;i<v3.length;i++){
  for(int j=0;j<((Object[])v3[i]).length;j++){
    System.out.print(((Object[])v3[i])[j] + " ");
  }//end inner loop
  System.out.println();//new line
}//end outer loop
```

5.124

The rectangular output

This approach produced the following output on the screen, *(which is the same as the previous two approaches):*

```
 1 2 3
2 4 6
```

Ragged arrays

All the code in the previous three cases has been used to emulate a traditional rectangular two-dimensional array. In the first case, each row was required to have the same number of elements by the syntax used to create the tree of array objects.

In the second and third cases, each row was not required to have the same number of elements, but they were programmed to have the same number of elements in order to emulate a rectangular two-dimensional array.

A triangular array, sort of ...

Now I am going to show you some cases that take advantage of the *ragged-array* capability of Java array objects. In the next case, *(beginning with Listing 19 (p. 754))*, I will create a ragged array having two rows. The first row will have two elements and the second row will have three elements. *(This array object might be thought of as being sort of triangular.)*

Listing 19: A triangular array.

```
Object[][] v4 = new Object[2][];
v4[0] = new Object[2];
v4[1] = new Object[3];
```

5.125

You have seen this before

You saw code like this in the second case discussed earlier. However, in that case, the second and third statements created new array objects having the same length. In this case, the second and third statements create array objects having different lengths. This is one of the ways to create a ragged array in Java *(you will see another way in the next case that I will discuss).*

Populate the leaf array objects

Listing 20 (p. 755) populates the elements of the leaf array objects with references to objects of the class **Integer** .

Listing 20: Populate the leaf array objects.

```
    for(int i=0;i<v4.length;i++){
  for(int j=0;j<v4[i].length;j++){
    v4[i][j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
 }//end outer loop
```

5.126

You have seen this before also

You have also seen the code in Listing 20 (p. 755) before. I repeated it here because this case clearly emphasizes the value of the **length** constant that is available in all Java array objects. In the earlier case, the **length** of the two leaf array objects was the same, so it would have been feasible to simply hard-code that value into the conditional expression of the inner **for** loop.

The length is not the same now

However, in this case, the **length** of the two leaf array objects is not the same. Therefore, it wouldn't work to hard-code a limit into the conditional expression of the inner **for** loop. However, because the **length** of each leaf array object is available as a public member of the array object, that value can be used to control the number of iterations of the inner loop for each separate leaf array object.

The triangular output

The next section of code in the program shown in Listing 26 (p. 761) near the end of the module uses the same code as before to display the **int** values encapsulated in the **Integer** objects whose references are stored in the leaf array objects. Since it is the same code as before, I won't repeat it here.

The output produced by this case is shown below:

```
 1 2
2 4 6
```

Note that this is not the same as before, and this output does not describe a rectangular array. Rather, it describes a ragged array where the rows are of different lengths.

(As I indicated earlier, it is sort of triangular. However, it could be any shape that you might want it to be.)

A more general approach

The next case, shown in Listing 21 (p. 756) , is the same as the third case discussed earlier, except that the lengths of the leaf array objects are not the same.

As before, this case creates a one-dimensional array object of type **Object** *(having two elements)* that forms the root of a tree. Each element in the root object contains a reference to another array object of type **Object** .

One of those leaf objects has two elements and the other has three elements, thus producing a ragged array *(you could make the lengths of those objects anything that you want them to be).*

Listing 21: A more general approach.

```
Object[] v5 = new Object[2];
v5[0] = new Object[2];
v5[1] = new Object[3];
```

5.127

Populate the leaf objects

As before, the elements in the leaf array objects are populated with references to objects of the class **Integer** , which encapsulate **int** values based on the current value of the loop counters. This is shown in Listing 22 (p. 756) .

Listing 22: Populate the leaf objects.

```
  for(int i=0;i<v5.length;i++){
 for(int j=0;
      j<((Object[])v5[i]).length;
                          j++){
  ((Object[])v5[i])[j] =
        new Integer((i+1)*(j+1));
 }//end inner loop
}//end outer loop
```

5.128

Same code as before

This is the same code that you saw in Listing 17 (p. 753) . I repeated it here to emphasize the requirement for casting .

Display the data

This case uses the same code as Listing 18 (p. 754) to display the **int** values encapsulated by the **Integer** objects whose references are stored in the elements of the leaf array objects. I won't repeat that code here.

The triangular output

The output produced by this case is shown below:

```
 1 2
2 4 6
```

Note that this is the same as the case immediately prior to this one. Again, it does not describe a rectangular array. Rather, it describes a ragged array where the rows are of different lengths.

A more general case

I'm going to show you one more general case for a ragged array. This case illustrates a more general approach. In this case, I will create a one-dimensional array object of element type **Object** . I will populate the elements of that array object with references to other array objects. These array objects will be the leaves of the tree.

Leaf array objects are type int

In this case, the leaves won't be of element type **Object** . Rather, the elements in the leaf objects will be designed to store primitive **int** values.

(An even more general case would be to populate the elements of the root object with references to a mixture of objects of class types, interface types, and array objects where the elements of the array objects are designed to store primitives of different types, and references of different types. Note, however, each leaf array object must be designed to store a single type, but will accept for storage any type that is assignment-compatible with the specified type for the array object.)

This case begins in Listing 23 (p. 757) , which creates the root array object, and populates its elements with references to leaf array objects of type **int** .

Listing 23: Beginning of a more general case .

```
Object[] v6 = new Object[2];
v6[0] = new int[7];
v6[1] = new int[3];
```

5.129

Leaf objects are different lengths

One of the leaf array objects has a length of 7. The other has a length of 3.

Populate the leaf array elements

Listing 24 (p. 757) populates the elements in the leaf array objects with values of type **int** .

Listing 24: Populate the leaf array elements.

```
    for(int i=0;i<v6.length;i++){
  for(int j=0;j<((int[])v6[i]).length;j++){
    ((int[])v6[i])[j] = (i+2)*(j+2);
  }//end inner loop
 }//end outer loop
```

5.130

Similar to previous code

The code in Listing 24 (p. 757) is similar to code that you saw earlier. The differences are:

- Cast is to type **int[]** instead of **object[]**
- Values assigned are type **int** instead of references to **Integer** objects

Display the output

Finally, Listing 25 (p. 758) displays the **int** values stored in the elements of the leaf array objects.

Listing 25: Display the output.

```
  for(int i=0;i<v6.length;i++){
for(int j=0;j<((int[])v6[i]).length;j++){
  System.out.print(((int[]

                                    5.131
```

The code in Listing 25 (p. 758) is very similar to what you have seen before, and there should be no requirement for an explanation of this code.

The code in Listing 25 (p. 758) produces the following output:

```
 4 6 8 10 12 14 16
6 9 12
```

I will leave it as an exercise for the student to correlate the output with the code.

5.2.13.5 Summary

When declaring a reference variable capable of referring to an array object, the array type is declared by writing the name of an element type followed by some number of empty pairs of square brackets [].

The components in an array object may refer to other array objects. The number of bracket pairs used in the declaration of the reference variable indicates the depth of array nesting *(in the sense that array elements can refer to other array objects).*

An array's length is not part of its type or reference variable declaration.

Multi-dimensional arrays are not required to represent rectangles, cubes, etc. They can be *ragged.*

The normal rules of *type conversion* and *assignment compatibility* apply when creating and populating array objects.

Object is the superclass of all other classes. Therefore, an array of element type **Object** is capable of storing references to objects instantiated from any other class. The type declaration for such an array object would be **Object[]** .

An array of element type **Object** is also capable of storing references to any other array object.

If the declared element type for the array object is one of the primitive types, the elements of the array can be used to store values of any primitive type that is *assignment compatible* with the declared type *(without the requirement for a cast).*

If the declared element type is the name of a class, *(which may or may not be abstract),* a null reference or a reference to any object instantiated from the class or any subclass of the class may be stored in the array element.

If the declared element type is an interface type, a null reference or a reference to any object instantiated from any class that implements the interface can be stored in the array element.

A reference variable whose declared type is an array type *does not contain an array.* Rather, it contains either null, or a reference to an array object. Declaring the reference variable does not create an array, nor does it allocate any space for the array components.

It is possible to declare a reference to an array object and initialize it with a reference to an array object when it is declared.

A reference to an array object can refer to different array objects *(of the same element type and different lengths)* at different points in the execution of a program.

When declaring an array reference variable, the square brackets [] may appear as part of the type, or following the variable name, or both.

Once an array object is created, its type and length never changes.

An array object is created by an array creation expression or an array initializer.

An array creation expression *(or an array initializer)* specifies:

- The element type
- The number of levels of nested arrays
- The length of the array for at least one of the levels of nesting

The length of the array is always available as a final instance variable named **length** .

An array element is accessed by an expression whose value is an array reference followed by an indexing expression enclosed by matching square brackets.

If an attempt is made to access the array with an invalid index value, an **ArrayIndexOutOfBoundsException** will be thrown.

Arrays must be indexed by integer values of the types **int** , **short** , **byte** , or **char** . An array cannot be accessed using an index of type **long** .

If the elements in an array are not purposely initialized when the array is created, the array elements will be automatically initialized with default values.

The values in the array elements may be purposely initialized when the array object is created using a comma-separated list of expressions enclosed by matching curly brackets.

The program in this module illustrated three different ways to emulate traditional rectangular two-dimensional arrays.

The program also illustrated two different ways to create and work with ragged arrays.

5.2.13.6 What's next?

In the next module, I will provide so additional information about array objects, and then illustrate the use of the classes named **Array** and **Arrays** for the creation and manipulation of array objects.

5.2.13.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 2
- File: Java1624.htm
- Published: 05/22/02
- Revised: 01/01/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.13.8 Complete program listing

A complete listing of the program is shown in Listing 26 (p. 761) below.

Listing 26: Complete program listing.

```
    /*File Array07.java
Copyright 2002, R.G.Baldwin

This program illustrates three
different ways to emulate a traditional
rectangular array in Java.  Two of
those ways are essentially ragged
arrays with equal-length sub arrays.

The program also illustrates two ways
to create ragged arrays in Java.

Tested using JDK 1.3 under Win 2000.
**************************************/

public class Array07{
  public static void main(
                        String[] args){

    //Create an array structure that
    // emulates a traditional
    // rectangular array with two rows
    // and three columns.  This
    // approach requires all rows to
    // be the same length.
    Object[][] v1 = new Object[2][3];
    //Populate the array elements with
    // references to objects of type
    // Integer.
    for(int i=0;i<v1.length;i++){
      for(int j=0;j<v1[i].length;j++){
        v1[i][j] =
              new Integer((i+1)*(j+1));
      }//end inner loop
    }//end outer loop
    //Display the array elements
    for(int i=0;i<v1.length;i++){
      for(int j=0;j<v1[i].length;j++){
        System.out.print(
                       v1[i][j] + " ");
      }//end inner loop
      System.out.println();//new line
    }//end outer loop
    System.out.println();//new line

    //Create a ragged array with two
    // rows.  The first row has three
    // columns and the second row has
    // three columns.  The length of
    // each row could be anything, but
    // was set to three to match the
    // above array structure.
    Object[][] v2 = new Object[2][];
    v2[0] = new Object[3];
    v2[1] = new Object[3];
```

-end-

5.2.14 Java1626: Array Objects, Part 3[36]

5.2.14.1 Table of Contents

5.2.14.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.14.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.2.14.2.1.1 Listings

5.2.14.3 Preview

This module discusses various details regarding the use of array objects in Java, including:

- The members of an array object
- The interfaces implemented by array objects
- **Class** objects and array objects
- The classes named **Array** and **Arrays**

[36] This content is available online at <http://cnx.org/content/m44200/1.4/>.

5.2.14.4 Discussion and sample code

Members of an array object

An array object has the following members *(in addition to the data stored in the object):*

- A public final variable named **length** , which contains the number of components of the array *(length may be positive or zero)*
- A public method named **clone** . This method overrides the method of the same name in **Object** class.
- Default versions of all the methods inherited from the class named **Object** , *(other than **clone** , which is overridden as described above).*

Implements Cloneable and Serializable

Also, every array object implements the **Cloneable** and **Serializable** interfaces. *(Note that neither of these interfaces declares any methods.)*

What is the Cloneable interface?

Here is what Sun has to say about the **Cloneable** interface:

*"A class implements the **Cloneable** interface to indicate to the **Object.clone()** method that it is legal for that method to make a field-for-field copy of instances of that class. Attempts to clone instances that do not implement the **Cloneable** interface result in the exception **CloneNotSupportedException** being thrown."*

Thus, the fact than an array object implements the **Cloneable** interface makes it possible to clone array objects.

A cloned array is shallow

While it is possible to clone arrays, care must be exercised when cloning multidimensional arrays. That is because a clone of a multidimensional array is shallow.

What does shallow mean?

Shallow means that the cloning process creates only a single new array.

Subarrays are shared between the original array and the clone.

(Although I'm not certain, I suspect that this may also be the case for cloning array objects containing references to ordinary objects. I will leave that determination as an exercise for the student. In any event, be careful if you clone array objects.) **Serialization**

Serialization of an object is the process of decomposing the object into a stream of bytes, which can later be recomposed into a copy of the object. Here is what Sun has to say about the **Serializable** interface:

*"Serializability of a class is enabled by the class implementing the **java.io.Serializable** interface. Classes that do not implement this interface will not have any of their state serialized or deserialized.*

All subtypes of a serializable class are themselves serializable.

The serialization interface has no methods or fields and serves only to identify the semantics of being serializable."

Even though this quotation from Sun doesn't address array objects, because array objects implement the **Serializable** interface, they can be serialized and later reconstructed.

Class objects representing array objects

An object of the class named **Class** can be obtained *(by calling the **getClass** method of the **Object** class)* to represent the class from which an ordinary object was instantiated.

The **Class** object is able to answer certain questions about the class that it represents *(such as the name of the superclass),* and has other uses as well.

*(One of the other uses is to specify the type as a parameter to the methods of the **Array** class, which I will illustrate later in this module.)*

Every array also has an associated **Class** object.

That **Class** object is shared with all other arrays with the same component type.

The superclass of an array type is **Object** . *(Think about this!)*

An array of characters is not a string

For the benefit of the C/C++ programmers in the audience, an array of **char** is not a **String** .

(In Java, a string is an object of the ***String*** *class or the* ***StringBuffer*** *class).*

Not terminated by null

Also, neither a **String** object nor an array of type **char** is terminated by '\u0000' *(the NUL character)* .

(This information is provided for the benefit of C programmers who are accustomed to working with so-called null-terminated strings. If you're not a C programmer, don't worry about this.)

A String object in Java is immutable

Once initialized, the contents of a Java **String** object never change.

On the other hand, an array of type **char** has mutable elements. The **String** class provides a method named **toCharArray** , which returns an array of characters containing the same character sequence as a **String** .

StringBuffer objects

The class named **StringBuffer** also provides a variety of methods that work with arrays of characters. The contents of a **StringBuffer** object are mutable.

The Array and Arrays classes

The classes named **Array** and **Arrays** provide methods that you can use to work with array objects.

The **Array** class provides static methods to dynamically create and access Java arrays.

The **Arrays** class contains various methods for manipulating arrays *(such as sorting and searching).* It also contains a static factory method that allows arrays to be viewed as lists.

A sample program named Array08

The sample program named **Array08** *(shown in Listing 12 (p. 774) near the end of the module)* illustrates the use of some of these methods.

Will discuss in fragments

As usual, I will discuss this program in fragments. Essentially all of the interesting code is in the method named **main** , so I will begin my discussion there. The first few fragments will illustrate the creation, population, and display of a one-dimensional array object whose elements contain references to objects of type **String** .

The newInstance method of the Array class

The code in Listing 1 (p. 764) calls the **static** method of the **Array** class named **newInstance** to create the array object and to store the object's reference in a reference variable of type **Object** named **v1** .

(Note that there are two overloaded versions of the ***newInstance*** *method in the* ***Array*** *class. I will discuss the other one later.)*

Listing 1: Using the newInstance method.

```
Object v1 = Array.newInstance(Class.forName("java.lang.String"),
                              3);
```

5.133

Two parameters required

This version of the **newInstance** method requires two parameters. The first parameter specifies the component type. This must be a reference to a **Class** object representing the component type of the new array object.

The second parameter, of type **int** , specifies the length of the new array object.

The Class object

The second parameter that specifies the array length is fairly obvious. However, you may need some help with the first parameter. Here is part of what Sun has to say about a **Class** object.

*"Instances of the class **Class** represent classes and interfaces in a running Java application. Every array also belongs to a class that is reflected as a **Class** object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as **Class** objects."*

Getting a reference to a Class object

I know of three ways to get *(or refer to)* a **Class** object.

- **Class** objects for primitive types
- The **getClass** method
- The **forName** method

Class objects for primitive types

There are nine predefined **Class** objects that represent the eight primitive types and void. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent: **boolean** , **byte** , **char** , **short** , **int** , **long** , **float** , and **double** . You can refer to these class objects using the following syntax:

- boolean.class,
- int.class,
- float.class, etc.

I will illustrate this later in this module.

The getClass method

If you have a reference to a target object *(ordinary object or array object)*, you can gain access to a **Class** object representing the class from which that object was instantiated by calling the **getClass** method of the **Object** class, on that object.

The **getClass** method returns a reference of type **Class** that refers to a **Class** object representing the class from which the target object was instantiated.

The forName method

The static **forName** method of the **Class** class accepts the name of a class or interface as an incoming **String** parameter, and returns the **Class** object associated with the class or interface having the given string name.

*(The **forName** method cannot be used with primitive types as a parameter.)*

Class object for the String class

Referring back to Listing 1 (p. 764) , you will see that the first parameter passed to the **newInstance** method was a reference to a **Class** object representing the **String** class.

Thus, the statement in Listing 1 (p. 764) creates a one-dimensional array object, of component type **String** , three elements in length.

The reference to the array object is saved in the generic reference variable of type **Object** .

(In case you haven't recognized it already, this is an alternative to syntax such as

new String[3] *.*

Note that there are no square brackets in this alternative approach. Thus, it might be said that this approach is more mainstream OOP than the approach that requires the use of square brackets.)

Populate the array object

The code in Listing 2 (p. 766) uses two **static** methods of the **Array** class to populate the three elements of the array object with references to objects of type **String** .

Listing 2: Populate the array object.

```
    for(int i = 0; i < Array.getLength(v1);i++){
  Array.set(v1, i, "a"+i);
}//end for loop
```

5.134

The getLength method

The **getLength** method of the **Array** class is used in Listing 2 (p. 766) to get the **length** of the array for use in the conditional expression of a **for** loop.

Note that unlike the sample programs in the previous module *(that stored the array object's reference as type **Object**)*, it was not necessary to cast the reference to type **String[]** in order to get the **length** .

The set method

The **set** method of the **Array** class is used in Listing 2 (p. 766) to store references to **String** objects in the elements of the array object.

Again, unlike the programs in the previous module, it was not necessary to cast the array reference to type **String[]** to access the elements. In fact, there are no square brackets anywhere in Listing 2 (p. 766) .

Display the data

Listing 3 (p. 766) uses a similar **for** loop to display the contents of the **String** objects whose references are stored in the elements of the array object.

Listing 3: Display the data.

```
    for(int i = 0; i < Array.getLength(v1); i++){
  System.out.print(Array.get(v1, i)  + " ");
}//end for loop
```

5.135

No square brackets

Once again, note that no casts, and no square brackets were required in Listing 3 (p. 766) . In fact, this approach makes it possible to deal with one-dimensional array objects using a syntax that is completely devoid of square brackets.

Rather than using square brackets to access array elements, this is a *method-oriented* approach to the use of array objects. This makes it possible to treat array objects much the same as we treat ordinary objects in Java.

A two-dimensional rectangular array object tree

Next, I will use the methods of the **Array** class to create, populate, and display a rectangular two-dimensional array object tree, whose elements contain references to objects of the class **String** .

Another overloaded version of newInstance

To accomplish this, I will use the other overloaded version of the **newInstance** method. This version requires a reference to an array object of type **int** as the second parameter.

(Note that the Sun documentation describes two different behaviors for this method, depending on the whether the first parameter represents a non-array class or interface, or represents an array type. This sample program illustrates the first possibility.)

The second parameter

As mentioned above, the version of the **newInstance** method that I am going to use requires a reference to an array object of type **int** as the second parameter.

*(The length of the array object of type **int** specifies the number of dimensions of the multi-dimensional array object. The contents of the elements of the array object of type **int** specify the sizes of those dimensions.)*

Thus, my first task is to create and populate an array object of type **int** .

An array object of type int

Listing 4 (p. 767) shows the code required to create and populate the array object of type **int** . This is a one-dimensional array object having two elements *(length equals 2)*. The first element is populated with the **int** value 2 and the second element is populated with the **int** value 3.

Listing 4: An array object of type int.

```
    Object v2 = Array.newInstance(int.class,2);
Array.setInt(v2, 0, 2);
Array.setInt(v2, 1, 3);
```

5.136

Why do we need this array object?

When this array object is used later, in conjunction with the version of the **newInstance** method that requires a reference to an array object of type **int** as the second parameter, this array object will specify an array object having two dimensions *(a rectangular array)*. The rectangular array will have two rows and three columns.

Same newInstance method as before

Note that Listing 4 (p. 767) uses the same version of the **newInstance** method that was used to create the one-dimensional array object in Listing l (p. 764) .

Class object representing int

Note the syntax of the first parameter passed to the **newInstance** method in Listing 4 (p. 767) . As mentioned earlier, this is a reference to the predefined **Class** object that represents the primitive type **int** . This causes the component type of the array object to be type **int** .

The setInt method

You should also note the use of the **setInt** method of the **Array** class to populate each of the two elements in the array in Listing 4 (p. 767) *(with **int** values of 2 and 3 respectively)*.

The two-dimensional array object tree

Listing 5 (p. 768) uses the other overloaded version of the **newInstance** method to create a two-dimensional array object tree, having two rows and three columns.

Listing 5: The two-dimensional array object tree.

```
Object v3 = Array.newInstance(Class.forName("java.lang.String"),
                              (int[])v2);
```

5.137

A reference to the array object at the root of the tree is stored in the reference variable of type **Object** named **v3** . Note that the tree is designed to store references to objects of type **String** .

(The number of dimensions and the size of each dimension are specified by the reference to the array object of type ***int*** *passed as the second parameter.)*

Square-bracket cast is required here

The required type of the second parameter for this version of the **newInstance** method is **int[]** . Therefore, there was no way for me to avoid the use of square brackets. I could either store the reference to the array object as type **Object** and cast it before passing it to the method, *(which I did),* or save it originally as type **int[]** , *(which I didn't).* Either way, I would have to know about the type **int[]** .

Populate the leaf elements

The nested **for** loop in Listing 6 (p. 768) uses the various methods of the **Array** class to populate the elements in the leaf array objects with references to objects of the class **String** .

Listing 6: Populate the leaf elements.

```
    for(int i=0;i < Array.getLength(v3);i++){
  for(int j=0;j < Array.getLength(Array.get(v3,i));j++){
    Array.set(Array.get(v3,i),j,"b" + (i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

5.138

Admittedly, the code in Listing 6 (p. 768) is a little complex. However, there is really nothing new there, so I won't discuss it further.

Display the data

Similarly, the code in Listing 7 (p. 769) uses the methods of the **Array** class in a nested **for** loop to get and display the contents of the **String** objects whose references are stored in the elements of the leaf array objects. Again, there is nothing new here, so I won't discuss this code further.

Listing 7: Display the data.

```
    for(int i=0;i < Array.getLength(v3);i++){
  for(int j=0;j < Array.getLength(Array.get(v3,i));j++){
    System.out.print(Array.get(Array.get(v3,i),j) + " ");
  }//end inner loop
  System.out.println();
}//end outer loop
System.out.println();
```

5.139

Very few square brackets

I will point out that with the exception of the requirement to create and pass an array object as type **int[]** , it was possible to write this entire example without the use of square brackets. This further illustrates the fact that the **Array** class makes it possible to create and work with array objects in a *method-oriented* manner, almost devoid of the use of square-bracket notation.

Sorting and Searching

Many college professors require their students to spend large amounts of time reinventing algorithms for sorting and searching *(and for various collections and data structures as well)*. There was probably a time in history when that was an appropriate use of a student's time. However, in my opinion, that time has passed.

Reuse, don't reinvent

Through a combination of the **Arrays** class, and the **Java Collections Framework** , most of the sorting, searching, data structures, and collection needs that you might have are readily available without a requirement for you to reinvent them.

(One of the most important concepts in OOP is ***reuse, don't reinvent*** *.)*

I will now illustrate sorting and searching using **static** methods of the **Arrays** class.

(Note that the ***Arrays*** *class is different from the* ***Array*** *class discussed earlier.)*

Create, populate, and display an array object

To give us something to work with, Listing 8 (p. 770) creates, populates, and displays the contents of an array object. Note that the array object is populated with references to **String** objects. There is nothing new here, so I won't discuss the code in Listing 8 (p. 770) in detail.

Listing 8: Create, populate, and display an array object.

```
    Object v4 = Array.newInstance(Class.forName("java.lang.String"),
                                  8);
//Populate the array object.
// Create a gap in the data.
for(int i = 0; i < Array.getLength(v4); i++){
  if(i < 4){
    Array.set(v4,i,"c"+(8-i));}
  else{
    Array.set(v4,i,"c"+(18-i));}
}//end for loop

//Display the raw data
for(int i = 0; i < Array.getLength(v4); i++){
  System.out.print(Array.get(v4,i)+ " ");
}//end for loop
```

5.140

The output

The code in Listing 8 (p. 770) produces the following output on the screen:

```
c8 c7 c6 c5 c14 c13 c12 c11
```

Note that the order of this data is generally descending, and there is no string encapsulating the characters **c4** .

Sort and display the data

The code in Listing 9 (p. 770) uses the **sort** method of the **Arrays** class to sort the array data into ascending order.

Listing 9: Sort and display the data.

```
    Arrays.sort((Object[])v4);

//Display the sorted data
for(int i = 0; i < Array.getLength(v4); i++){
  System.out.print(Array.get(v4, i) + " ");
}//end for loop
```

5.141

The output

The code in Listing 9 (p. 770) displays the sorted contents of the array object, producing the following **output on the computer screen** :

```
c11 c12 c13 c14 c5 c6 c7 c8
```

Note that the order of the data in the array object has been modified, and the array data is now in ascending order.

(This order is based on the natural ordering of the ***String*** *data. I discuss other ways to order sorted data in conjunction with the* ***Comparable*** *and* ***Comparator*** *interfaces in my modules on the Java Collections Framework.)*

Binary search

A binary search is a search algorithm that can very quickly find an item stored in a sorted collection of items. In this case, the collection of items is stored in an array object, and the data is sorted in ascending order.

Search for an existing string

Listing 10 (p. 771) uses the **binarySearch** method of the **Arrays** class to perform a search for an existing **String** object whose reference is stored in the sorted array. The code searches for the reference to the **String** object encapsulating the characters **c5** .

Listing 10: Search for an existing string.

```
System.out.println(Arrays.binarySearch((Object[])v4,"c5"));
```

5.142

The result of the search

The code in Listing 10 (p. 771) displays the numeral 4 on the screen.

When the **binarySearch** method finds a match, it returns the index value of the matching element. If you go back and look at the sorted contents (p. 770) of the array shown earlier, you will see that this is the index of the element containing a reference to a **String** object that encapsulates the characters **c5** .

Search for a non-existing string

The code in Listing 11 (p. 771) uses the **binarySearch** method to search for a reference to a **String** object that encapsulates the characters **c4** . As I indicated earlier, a **String** object that encapsulates these characters is not represented in the sorted array object.

Listing 11: Search for a non-existing string.

```
System.out.println(Arrays.binarySearch((Object[])v4,"c4"));
```

5.143

The result of the search

The code in Listing 11 (p. 771) produces the following negative numeral on the screen: **-5** .

Here is Sun's explanation for the value returned by the **binarySearch** method:

"Returns: index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first

element greater than the key, or list.size(), if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found."

Thus, the negative return value indicates that the method didn't find a match. The absolute value of the return value can be used to determine the index of the reference to the target object if it did exist in the sorted list. I will leave it as an exercise for the student to interpret Sun's explanation beyond this simple explanation.

Other capabilities

In addition to sorting and searching, the **Arrays** class provides several other methods that can be used to manipulate the contents of array objects in Java.

5.2.14.5 Summary

An array object has the following members *(in addition to the data stored in the object):*

- A public final variable named **length**
- An overridden version of the public method named **clone**
- Default versions of all the other methods inherited from the class named **Object**

Every array object implements the **Cloneable** and **Serializable** interfaces.

A clone of a multidimensional array is shallow. Therefore, you should exercise caution when cloning array objects.

Because array objects implement the **Serializable** interface, they can be serialized and later reconstructed.

Every array also has an associated **Class** object.

The classes named **Array** and **Arrays** provide methods that you can use to work with array objects.

The **Array** class provides static methods to dynamically create and access Java array objects.

The **Arrays** class contains various methods for manipulating arrays *(such as sorting and searching).* It also contains a static factory method that allows arrays to be viewed as lists.

Class objects are required when using the methods of the **Array** class to dynamically create Java array objects.

There are nine predefined **Class** objects that represent the eight primitive types and void. They are accessed using the following syntax: boolean.class, int.class, etc.

Three ways to get a **Class** object are:

- Class objects for primitive types: **int.class** , etc.
- The **getClass** method
- The **forName** method

The methods of the **Array** class make it possible to deal with one-dimensional array objects using a syntax that is completely devoid of square brackets. This is a *method-oriented* approach to the use of array objects. This makes it possible to treat array objects much the same as we treat ordinary objects in Java. The required syntax for multi-dimensional array objects is mostly devoid of square brackets.

The **Arrays** class provides methods for sorting and searching array objects as well as performing other operations on array objects as well.

Through a combination of the **Arrays** class and the Java Collections Framework, most of the sorting, searching, data structures, and collection needs that you might have are readily available without a requirement for you to reinvent them.

One of the most important concepts in OOP is *reuse, don't reinvent.*

5.2.14.6 What's next?

The next module will explain the use of the ***this*** and ***super*** keywords.

5.2.14.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 3
- File: Java1626.htm
- Published: 08/08/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.2.14.8 Complete program listing

A complete listing of the program is shown in Listing 12 (p. 774) below.

Listing 12: Complete program listing.

```
    /*File Array08.java
Copyright 2002, R.G.Baldwin
Rev 2/10/02

This program illustrates the use of
static methods of the Array class to
dynamically create and access Java
arrays.

It also illustrates the use of static
methods of the Arrays class to sort
and search array objects.

Tested using JDK 1.3 under Win 2000.
**************************************/
import java.lang.reflect.Array;
import java.util.Arrays;

public class Array08{
  public static void main(
                        String[] args){

  try{
    //Create, populate, and display a
    // one-dimensional array object
    // whose elements contain
    // references to objects of type
    // String.

    //Create the array object
    Object v1 = Array.newInstance(
              Class.forName(
               "java.lang.String"), 3);
    //Populate the array object
    for(int i = 0; i <
             Array.getLength(v1); i++){
      Array.set(v1, i, "a"+i);
    }//end for loop
    //Display the data
    for(int i = 0; i <
             Array.getLength(v1); i++){
      System.out.print(
                Array.get(v1, i) + " ");
    }//end for loop
    System.out.println();
    System.out.println();

    //Create, populate, and display a
    // rectangular two-dimensional
    // array object tree whose
    // elements contain references
    // to objects of type String.

    //First create an array object of
    // type int required as a
```

-end-

5.2.15 Java1628: The this and super Keywords[37]

5.2.15.1 Table of Contents

5.2.15.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.15.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.15.2.1.1 Images

5.2.15.2.1.2 Listings

5.2.15.3 Preview

This module explains the use of the keywords ***this*** and ***super*** . Short sample programs illustrate how you can use these keywords for several purposes.

I will discuss and illustrate the use of the **this** keyword in the following situations:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.

[37] This content is available online at <http://cnx.org/content/m44201/1.4/>.

- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example).*

I will also discuss and illustrate the use of the **super** keyword in the following situations:

- To bypass the overridden version of a method in a subclass and execute the version in the superclass.
- To bypass a member variable in a subclass in order to access a member variable having the same name in a superclass.
- To cause a constructor in a subclass to call a parameterized constructor in the immediate superclass.

5.2.15.4 Discussion and sample code

You already know quite a lot about OOP

By now you know that an *object* is an *instance of a class* . You know that all variables and methods in Java must be contained in a class or an object. You know that the three primary characteristics of an object-oriented programming language are:

- *encapsulation*
- *inheritance*
- *polymorphism* .

If you have been studying this series of modules on the Essence of OOP in Java, you already know quite a lot about OOP in general, and the implementation of OOP in Java in particular.

A few more important OOP/Java concepts

However, there are a few more important concepts that I haven't previously discussed in this series of modules. In this module, I will explain the use of the keywords ***this*** and ***super*** .

Data and methods

The class provides the plan from which objects are built. This plan defines the *data* that is to be stored in an object, and the *methods* for manipulating that data. The data is variously referred to as *data members*, *fields* , and *variables* , depending on which book you are reading.

Non-static and static

The data can be further sub-divided into *non-static* and *static* , often referred to as i *nstance variables* and *class variables* respectively.

The methods are also often referred to as *member methods* , and they can also be *static* or *non-static* . Static methods are often referred to as *class methods* while non-static methods are often referred to as *instance methods* .

Instance variables and instance methods

The class body contains the *declarations* for, and possibly the *initialization* of all data members *(both class variables and instance variables)* as well as the full definition of all *methods* .

In this module, we will be particularly interested in instance variables and instance methods.

Every class is a subclass of Object

By default, every class in Java extends *(either directly or indirectly)* the class named **Object** . A new class may either extend **Object** , or extend another class that extends **Object** , or extend another class further down the inheritance hierarchy.

The immediate parent class of a new class is known as its *superclass* , and the new class is known as the *subclass* .

(Sometimes we use the word superclass to indicate the collection of classes in the inheritance hierarchy from which a specific class is derived.)

If you do not specify the *superclass* for a new class, it will extend **Object** by default.

The *extends* keyword

The keyword **extends** is used in the class declaration to specify the immediate *superclass* of the new class using the syntax shown in Image 1 (p. 777) .

Image 1: The extends keyword.

```
    class NewClass extends SuperClassName{
  //body of class
}//end class definition
```

5.145

Inheritance

A class inherits the variables and methods of its superclass, and of the superclass of that class, etc., all the way back up the family tree to the single class **Object**, which is the root of all inheritance.

Thus, an object that is instantiated from a class contains all the instance variables and all the instance methods defined by that that class and defined by all its ancestors.

However, the methods may have been *overridden* one or more times along the way. Also, access to those variables and methods may have been restricted through the use of the **public**, **private**, and **protected** keywords.

(There is another access level, often referred to as ***package private****, which is what you get when you don't use an access keyword.)*

The this keyword

Every instance method in every object in Java receives a reference named **this** when the method is called. The reference named **this** is a reference to the object on which the method was called. It can be used for any purpose for which such a reference is needed.

Three common situations

There are at least three common situations where such a reference is needed:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.
- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example).*

Normally, instance methods belonging to an object have direct access to the instance variables belonging to that object, and to the class variables belonging to the class from which that object was instantiated.

(Class methods never have access to instance variables or instance methods.)

Name can be duplicated

However, the name of a method parameter or constructor parameter can be the same as the name of an instance variable belonging to the object or a class variable belonging to the class. It is also allowable for the name of a local variable to be the same as the name of an instance variable or a class variable. In this case, the local variable or the parameter is said to *hide* the member variable having the same name.

Reference named *this* is passed to instance methods

As mentioned above, whenever an instance method is called on an object, a hidden reference named **this** is always passed to the method. The **this** reference always refers to the object on which the method was called. This makes it possible for the code in the method to refer back to the object on which the method was called.

The reference named **this** can be used to access the member variables hidden by the local variables or parameters having of the same name.

The sample program named This01

The sample program shown in Listing 1 (p. 779) illustrates the use of the **this** reference to access a hidden instance variable named **myVar** and a hidden class variable named **yourVar** .

Listing 1: The program named This01.

```
    /*File This01.java
Copyright 2002, R.G.Baldwin
Illustrates use of this keyword to
access hidden member variables.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

myVar parameter = 20
local yourVar variable = 1
Instance variable myVar = 5
Class variable yourVar = 10
************************************/

class This01 {
  int myVar = 0;
  static int yourVar = 0;

  //Constructor with parameters named
  // myVar and yourVar
  public This01(int myVar,int yourVar){
    this.myVar = myVar;
    this.yourVar = yourVar;
  }//end constructor
  //---------------------------------//

  //Method with parameter named myVar
  // and local variable named yourVar
  void myMethod(int myVar){
    int yourVar = 1;
    System.out.println(
         "myVar parameter = " + myVar);
    System.out.println(
            "local yourVar variable = "
                           + yourVar);
    System.out.println(
           "Instance variable myVar = "
                       + this.myVar);
    System.out.println(
            "Class variable yourVar = "
                     + this.yourVar);
  }//end myMethod
  //---------------------------------//

  public static void main(
                        String[] args){
    This01 obj = new This01(5,10);
    obj.myMethod(20);
  }//end main method
}//End This01 class definition.
```

The key points

The key points to observe in the program is Listing 1 (p. 779) are:

- When the code refers to **myVar** or **yourVar** , the reference resolves to either an incoming parameter or to a local variable having that name.
- When the code refers to **this.myVar** or **this.yourVar** , the reference resolves to the corresponding instance variable and class variable having that name.

To summarize this situation, every time an instance method is called, it receives a hidden reference named **this** . That is a reference to the object on which the method was called.

The code in the method can use that reference to access any instance member of the object on which it was called, or any class member of the class from which the object was instantiated.

However, when class methods are called, they do not receive such a hidden reference, and therefore, they cannot refer to any instance members of any object instantiated from the class. They can only access class members of the same class.

Calling other constructors of the same class

Now I am going to discuss and illustrate the second common situation listed earlier.

A class can define two or more overloaded constructors having the same name and different argument lists. Sometimes it is useful for one overloaded constructor to call another overloaded constructor in the same class. When this is done, the constructor being called is referred to as though it were a method whose name is **this** , and whose argument list matches the argument list of the constructor being called.

The sample program named This02

This situation is illustrated in the program named **This02** shown in Listing 2 (p. 781) .

Listing 2: The program named This02.

```
    /*File This02.java
Copyright 2002, R.G.Baldwin
Illustrates use of this keyword for one
overloaded constructor to access
another overloaded constructor of the
same class.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

Instance variable myVar = 15
************************************/

class This02 {
  int myVar = 0;

  public static void main(
                        String[] args){
    This02 obj = new This02();
    obj.myMethod();
  }//end main method
  //---------------------------------//

  //Constructor with no parameters
  public This02(){
    //Call parameterized constructor
    this(15);
  }//end constructor
  //---------------------------------//

  //Constructor with one parameter
  public This02(int var){
    myVar = var;
  }//end constructor
  //---------------------------------//

  //Method to display member variable
  // named myVar
  void myMethod(){
    System.out.println(
           "Instance variable myVar = "
                              + myVar);
  }//end myMethod

}//End This02 class definition.
```

Calling a *noarg* constructor

The **main** method in Listing 2 (p. 781) instantiates a new object by applying the **new** operator to the *noarg* constructor for the class named **This02** .

(The common jargon for a constructor that doesn't take any parameters is a noarg constructor.)

The *noarg* constructor calls a parameterized constructor

The code in the *noarg* constructor uses the **this** keyword to call the other overloaded constructor, passing an **int** value of 15 as a parameter.

That constructor stores the value of the incoming parameter (15) in the instance variable named **myVar** . Then control returns to the *noarg* constructor, which in turn returns control to the **main** method. When control returns to the **main** method, the new object has been constructed, and the instance variable named **myVar** belonging to that object contains the value 15.

Display the value of the instance variable

The next statement in the **main** method calls the method named **myMethod** on the object, which causes the value stored in the instance variable (15) to be displayed on the screen.

The most important statement

For purposes of this discussion, the most important statement in the program is the statement that reads:

```
this(15);
```

This is the statement used by one overloaded constructor to call another overloaded constructor.

Callbacks

An extremely important concept in programming is the third situation mentioned in the earlier list (p. 777) . This is a situation where a method in one object calls a method in another object and passes a reference to itself as a parameter.

(This is sometimes referred to as registration. That is to say, one object registers itself on another object.)

The method in the second object saves the reference that it receives as an incoming parameter. This makes it possible for a method in the second object to make a callback to the first object sometime later. This is illustrated in the program named **This03** , shown in Listing 3 (p. 783) .

Listing 3: The program named This03 .

```
    /*File This03.java
Copyright 2002, R.G.Baldwin
Illustrates using the this keyword in
a callback scenario.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

Instance variable myVar = 15
************************************/

class This03 {
  public static void main(
                        String[] args){
    ClassA objA = new ClassA();
    ClassB objB = new ClassB();
    objA.goRegister(objB);
    objB.callHimBack();
    objA.showData();
  }//end main method
}//End This03 class definition.
//===================================//

class ClassA{
  int myVar;

  void goRegister(ClassB refToObj){
    refToObj.registerMe(this);

  }//end goRegister
  //---------------------------------//

  void callMeBack(int var){
    myVar = var;
  }//end callMeBack
  //---------------------------------//

  void showData(){
    System.out.println(
           "Instance variable myVar = "
                             + myVar);
  }//end showData
}//end ClassA
//===================================//

class ClassB{
  ClassA ref;

  void registerMe(ClassA var){
    ref = var;
  }//end registerMe
  //---------------------------------//

  void callHimBack(){
```

Not intended to be useful

Note that the program in Listing 3 (p. 783) is intended solely to illustrate the concept of a callback, and is not intended to do anything useful. This is a rather long and convoluted explanation, so please bear with me.

The **main** method begins by instantiating two objects, one each from the classes named **ClassA** and **ClassB** .

Go register yourself

Then the **main** method sends a message to **objA** telling it to go register itself on **objB** . A reference to **objB** is passed as a parameter to the method named **goRegister** belonging to **objA** .

The code in **objA** uses this reference to call the method named **registerMe** on **objB** , passing **this** as a parameter. In other words, the code in **objA** calls a method belonging to **objB** passing a reference to itself as a parameter. The code in **objB** saves that reference in an instance variable for later use.

Make a callback

Then the **main** method sends a message to **objB** asking it to use the saved reference to make a callback to **objA** . The code in the method named **callHimBack** uses the reference to **objA** saved earlier to call the method named **callMeBack** on **objA** , passing 15 as a parameter. The method named **callMeBack** belonging to **objA** saves that value in an instance variable.

Show the data

Finally, the **main** method calls the **showData** method on **objA** to cause the value stored in the instance variable belonging to **objA** to be displayed on the computer screen.

Callbacks are important

Again, this program is provided solely to illustrate the concept of a callback using the **this** keyword. In practice, callbacks are used throughout Java, but they are implemented in a somewhat more elegant way, making use of interfaces.

For example, interfaces with names like **Observer** and **MouseListener** are commonly used to register *observer* objects on *observable* objects *(sometimes referred to as listeners and sources).* Then later in the program, when something of interest happens on the *observable* object *(the source),* all registered *observer* objects *(the listeners),* are notified of the event.

The main point regarding the *this* reference

The main point of this discussion is that the **this** reference is available to all instance methods belonging to an object, and can be used whenever there is a need for a reference to the object on which the method is called.

To disambiguate something

At least one prominent author uses the word *disambiguate* to describe the process described by the first item in the earlier list (p. 777) , where the **this** keyword is used to bypass one variable in favor of a different variable having the same name. I will also use that terminology in the following discussion.

Three uses of the *super* keyword

Here are three common uses of the **super** keyword:

- If your class *overrides* a method in a superclass, you can use the **super** keyword to bypass the overridden version in the class and execute the version in the superclass.
- If a local variable in your method or a member variable in your class hides a member variable in the superclass *(having the same name),* you can use the **super** keyword to access the member variable in the superclass.
- You can also use **super** in a constructor of your class to call a parameterized constructor in the superclass.

The program named Super3

The program in Listing 4 (p. 786) uses **super** to call a parameterized constructor in the superclass from the subclass constructor. This is an important use of **super** .

The program also uses **this** and **super** to disambiguate a local variable, an instance variable of the subclass, and an instance variable of the superclass. All three variables have the same name.

Listing 4: The program named Super3.

```
    /*File Super3.java
Copyright 2002, R.G.Baldwin
Illustrates use of super reference to
access constructor in superclass.  Also
illustrates use of super to
disambiguate instance variable in
subclass from instance variable in
superclass.  Illustrates use of this
to disambiguate local variable from
instance variable in subclass.

Tested using JDK 1.4.0 under Win2000

The output from this program is:

In SuperClass constructor.
Setting superclass instance var to 500

In subclass constructor.
Setting subclass instance var to 400

In main
Subclass instance var = 400

In method myMeth
Local var = 300
Subclass instance var = 400
SuperClass instance var = 500
************************************/
class SuperClass{
  int data;

  //Parameterized superclass
  // constructor
  public SuperClass(int val){
    System.out.println(
        "In SuperClass constructor. ");
    System.out.println(
         "Setting superclass instance "
                   + "var to " + val);
    data = val;
    System.out.println();//blank line
  }//end SuperClass constructor
}//end SuperClass class definition
//===================================//

class Super3 extends SuperClass{
  //Instance var in subclass has same
  // name as instance var in superclass
  int data;
```



```
  //Subclass constructor
  public Super3(){
    //Call parameterized SuperClass
    // constructor
```

The keyword **super** is used twice in the program in Listing 4 (p. 786) .

Call a parameterized constructor

The first usage of the keyword **super** appears as the first executable statement in the *noarg* constructor for the class named **Super3** . This statement reads as follows:

```
super(500);
```

This statement causes the parameterized constructor for the immediate superclass *(the class named **SuperClass**)* of the class named **Super3** , to be executed before the remaining code in the constructor for **Super3** is executed.

This is the mechanism by which you can cause a parameterized constructor in the immediate superclass to be executed.

What if you don't do this?

If you don't do this, an attempt will always be made to call a *noarg* constructor on the superclass before executing the remaining code in the constructor for your class.

(That is why you should almost always make certain that the classes that you define have a noarg constructor in addition to any parameterized constructors that you may define.)

First executable statement in constructor

When **super(parameters)** is used to call the superclass constructor, it must always be the first executable statement in the constructor.

Whenever you call the constructor of a class to instantiate an object, if your constructor doesn't have a call to **super** as the first executable statement in the constructor, the call to the *noarg* constructor in the superclass is made automatically.

In other words, in order to construct an object of a class, it is necessary to first construct that part of the object attributable to the superclass. That normally happens automatically, making use of the superclass constructor that doesn't take any parameters.

Calling a parameterized constructor

If you want to use a version of the superclass constructor that takes parameters, you can make your own call to **super(parameters)** as the first executable statement in your constructor *(as was done in this program)*.

Accessing a superclass member variable

The second use of the **super** keyword in the program shown in Listing 4 (p. 786) uses the keyword to bypass an instance variable named **data** in the class named **Super3** , to access and display the value of an instance variable named **data** in the superclass named **SuperClass** .

Note that in that same section of code, the **this** keyword is used to bypass a local variable named **data** in order to display the value of an instance variable named **data** in the class named **Super3** .

Similarly, a statement without the use of either **this** or **super** is used to display the value of a local variable named **data** .

To disambiguate

Therefore, as stated earlier, the program uses **this** and **super** to disambiguate a local variable, an instance variable of the subclass, and an instance variable of the superclass, where all three variables have the same name.

Accessing overridden superclass method

As mentioned earlier (p. 784) , if your method *overrides* a method in its superclass, you can use the keyword **super** to call the overridden version in the superclass, possibly completely bypassing the overridden version in the subclass.

The program named Super4

This is illustrated by the program in Listing 5 (p. 788) . This program contains an overridden version of a superclass method named **meth** . The subclass version uses the value of an incoming parameter to decide whether to call the superclass version and then to call some of its own code, or to execute its own code exclusively.

Listing 5: The program named Super4.

```
    /*File Super4.java
Copyright 2002, R.G.Baldwin
Illustrates calling the superclass
version of an overridden method from
code in the subclass version.

Tested using JDK 1.4.0 under Win 2000.

The output from this program is:

In main
Entering overridden method in subclass
Incoming parameter is false
Subclass version only is called
Back in or still in subclass version
Goodbye from subclass version

Entering overridden method in subclass
Incoming parameter is true
SuperClass method called
Back in or still in subclass version
Goodbye from subclass version

Back in main
************************************/
class SuperClass{
  //Following method is overridden in
  // the subclass.
  void meth(boolean par){
    System.out.println(
       "Incoming parameter is " + par);
    System.out.println(
          "SuperClass method called");
  }//end meth
}//end SuperClass class definition
//===================================//

class Super4 extends SuperClass{
  //Following method overrides method
  // in the superclass
  void meth(boolean par){
    System.out.println(
          "Entering overridden method "
                      + "in subclass");
    //Decide whether to call
    // superclass version
    if(par)
      //Call superclass version
      super.meth(par);
    else{
      //Don't call superclass version
      System.out.println(
       "Incoming parameter is " + par);
      System.out.println(
            "Subclass version only is "
```

Only one statement contains super

The **super** keyword is used in only one statement in the program in Listing 5 (p. 788) . That statement appears in the subclass version of an overridden method, and is as follows:

```
super.meth(par);
```

This statement is inside the body of an **if** statement. If the value of **par** is true, then this statement is executed, causing the superclass version of the method named **meth** to be executed *(passing the value of* **par** *as a parameter to the superclass method).* When the method returns, the remaining code in the subclass version of the method is executed.

If the value of **par** is false, the above statement is bypassed, and the superclass version of the method doesn't get executed. In this case, only the code in the subclass version is executed.

5.2.15.5 Summary

I have discussed and illustrated the use of the **this** keyword in the following common situations:

- To bypass local variables or parameters that hide member variables having the same name, in order to access the member variable.
- To make it possible for one overloaded constructor to call another overloaded constructor in the same class.
- To pass a reference to the current object to a method belonging to a different object *(as in implementing callbacks, for example).*

I have also discussed and illustrated the use of the **super** keyword in the following situations:

- To bypass the overridden version of a method in a subclass and execute the version in the superclass.
- To bypass a member variable in a subclass in order to access a member variable having the same name in a superclass.
- To cause a constructor in a subclass to call a parameterized constructor in the immediate superclass.

5.2.15.6 What's next?

The next module in this collection will teach you how to use exception handling in Java.

5.2.15.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: The this and super Keywords
- File: Java1628.htm
- Published: 08/08/13
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.2.16 Java1630: Exception Handling[38]

5.2.16.1 Table of Contents

5.2.16.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.2.16.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.2.16.2.1.1 Images

[38] This content is available online at <http://cnx.org/content/m44202/1.4/>.

5.2.16.2.1.2 Listings

5.2.16.3 Preview

This module explains Exception Handling in Java. The discussion includes the following topics:

- What is an exception?
- How do you throw and catch exceptions?
- What do you do with an exception once you have caught it?
- How do you make use of the exception class hierarchy provided by the Java development environment?

This module will cover many of the details having to do with exception handling in Java. By the end of the module, you should know that the use of exception handling is not optional in Java, and you should have a pretty good idea how to use exception handling in a beneficial way.

5.2.16.4 Discussion and sample code

Introduction

Stated simply, the exception-handling capability of Java makes it possible for you to:

- Monitor for exceptional conditions within your program
- Transfer control to special exception-handling code *(which you design)* if an exceptional condition occurs

The basic concept

This is accomplished using the keywords: **try** , **catch** , **throw** , **throws** , and **finally** . The basic concept is as follows:

- You **try** to execute the statements contained within a block of code. *(A block of code is a group of one or more statements surrounded by curly brackets.)*
- If you detect an exceptional condition within that block, you **throw** an exception object of a specific type.
- You **catch** and process the exception object using code that you have designed.
- You optionally execute a block of code, designated by **finally** , which needs to be executed whether or not an exception occurs. *(Code in the **finally** block is normally used to perform some type of cleanup.)*

Exceptions in code written by others

There are also situations where you don't write the code to **throw** the exception object, but an exceptional condition that occurs in code written by someone else transfers control to exception-handling code that you write.

For example, the **read** method of the **InputStream** class throws an exception of type **IOException** if an exception occurs while the **read** method is executing. In this case, you are responsible only for the code in the **catch** block and optionally for the code in the **finally** block.

(This is the reason that you must surround the call to ***System.in.read()*** *with a* ***try*** *block followed by a* ***catch*** *block, or optionally declare that your method* ***throws*** *an exception of type* ***IOException*** *.)*

Exception hierarchy, an overview

When an exceptional condition causes an exception to be *thrown* , that exception is represented by an object instantiated from the class named **Throwable** or one of its subclasses.

Here is part of what Sun has to say about the **Throwable** class:

"The ***Throwable*** *class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java* ***throw*** *statement. Similarly, only this class or one of its subclasses can be the argument type in a* ***catch*** *clause."*

Sun goes on to say:

"Instances of two subclasses, ***Error*** *and* ***Exception*** *, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data)."*

The Error and Exception classes

The virtual machine and many different methods in many different classes throw *exceptions* and *errors* . I will have quite a lot more to say about the classes named **Error** and **Exception** later in this module.

Defining your own exception types

You may have concluded from the Sun quotation given above that you can define and **throw** exception objects of your own design, and if you did, that is a correct conclusion. *(Your new class must extend* ***Throwable*** *or one of its subclasses.)*

The difference between Error and Exception

As mentioned above, the **Throwable** class has two subclasses:

- **Error**
- **Exception**

What is an error?

What is the difference between an **Error** and an **Exception** ? Paraphrasing David Flanagan and his excellent series of books entitled Java in a Nutshell , an **Error** indicates that a non-recoverable error has occurred that should not be caught. Errors usually cause the Java virtual machine to display a message and exit.

Sun says the same thing in a slightly different way:

"An ***Error*** *is a subclass of* ***Throwable*** *that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions."*

For example, one of the subclasses of **Error** is named **VirtualMachineError** . This error is *"Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating. "*

What is an exception?

Paraphrasing Flanagan again, an **Exception** indicates an abnormal condition that must be properly handled to prevent program termination.

Sun explains it this way:

"The class ***Exception*** *and its subclasses are a form of* ***Throwable*** *that indicates conditions that a reasonable application might want to catch."*

As of JDK 1.4.0, there are more than fifty known subclasses of the **Exception** class. Many of these subclasses themselves have numerous subclasses, so there is quite a lot of material that you need to become familiar with.

The RuntimeException class

One subclass of **Exception** is the class named **RuntimeException** As of JDK 1.4.0, this class has about 30 subclasses, many which are further subclassed. The class named **RuntimeException** is a very important class.

Unchecked exceptions

The **RuntimeException** class, and its subclasses, are important not so much for what they do, but for what they don't do. I will refer to exceptions instantiated from **RuntimeException** and its subclasses as *unchecked* exceptions.

Basically, an unchecked exception is a type of exception that you can optionally handle, or ignore. If you elect to ignore the possibility of an unchecked exception, and one occurs, your program will terminate as a result. If you elect to handle an unchecked exception and one occurs, the result will depend on the code that you have written to handle the exception.

Checked exceptions

All exceptions instantiated from the **Exception** class, or from subclasses of **Exception** other than **RuntimeException** and its subclasses must either be:

- Handled with a **try** block followed by a **catch** block, or
- Declared in a **throws** clause of any method that can throw them

In other words, checked exceptions *cannot be ignored* when you write the code in your methods. According to Flanagan, the exception classes in this category represent routine abnormal conditions that should be anticipated and caught to prevent program termination.

Checked by the compiler

Your code must anticipate and either handle or declare checked exceptions. Otherwise, your program won't compile. *(These are exception types that are checked by the compiler.)*

Throwable constructors and methods

As mentioned above, all errors and exceptions are subclasses of the **Throwable** class. As of JDK 1.4.0, the **Throwable** class provides four constructors and about a dozen methods. The four constructors are shown in Image 1 (p. 793) .

Image 1: Throwable constructors.

```
Throwable()
Throwable(String message)
Throwable(String message,Throwable cause)
Throwable(Throwable cause)
```

5.151

The first two constructors have been in Java for a very long time. Basically, these two constructors allow you to construct an exception object with, or without a **String** message encapsulated in the object.

New to JDK 1.4

The last two constructors are new in JDK 1.4.0. These two constructors are provided to support the *cause facility.* The *cause facility* is new in release 1.4. It is also known as the *chained exception* [39] facility. *(I won't cover this facility in this module. Rather, I plan to cover it in a series of future modules.)*

Methods of the Throwable class

[39] http://softwaredev.earthweb.com/java/article/0,,12082_1431531_1,00.html

Image 2 (p. 794) shows some of the methods of the **Throwable** class. *(I omitted some of the methods introduced in JDK 1.4 for the reasons given above.)*

Image 2: Methods of the Throwable class.

```
fillInStackTrace()
getStackTrace()
printStackTrace().
setStackTrace(StackTraceElement[] stackTrace)

getLocalizedMessage()
getMessage()
toString()
```

5.152

The StackTrace

The first four methods in Image 2 (p. 794) deal with the *StackTrace* . In case you are unfamiliar with the term *StackTrace,* this is a list of the methods executed in sequence that led to the exception. *(This is what you typically see on the screen when your program aborts with a runtime error that hasn't been handled.)*

Messages

The two methods dealing with messages provide access to a **String** message that may be encapsulated in the exception object. The **getMessage** class simply returns the message that was encapsulated when the object was instantiated. *(If no message was encapsulated, this method returns null.)*

The **getLocalizedMessage** method is a little more complicated to use. According to Sun, *"Subclasses may override this method in order to produce a locale-specific message."*

The toString method

The **toString** method is inherited from the **Object** class and overridden in the exception subclass to *"return a short description of the* ***Throwable*** *".*

Inherited methods

All exception objects inherit the methods of the **Throwable** class, which are listed in Image 2 (p. 794) . Thus, any of these methods may be called by the code in the **catch** block in its attempt to successfully handle the exception.

For example, exceptions may have a message encapsulated in the exception object, which can be accessed using the **getMessage** method. You can use this to display a message describing the error or exception.

You can also use other methods of the **Throwable** class to:

- Display a stack trace showing where the exception or error occurred
- Produce a **String** representation of the exception object

So, what is an exception?

According to the online book entitled The Java Tutorial [40] by Campione and Walrath:

"The term exception is shorthand for the phrase "exceptional event". It can be defined as follows:

Definition: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions."

[40] http://java.sun.com/docs/books/tutorial/

When an exceptional condition occurs within a method, the method may instantiate an exception object and hand it off to the runtime system to deal with it. This is accomplished using the **throw** keyword. *(This is called throwing an exception.)*

To be useful, the exception object should probably contain information about the exception, including its type and the state of the program when the exception occurred.

Handling the exception

At that point, the runtime system becomes responsible for finding a block of code designed to handle the exception.

The runtime system begins its search with the method in which the exception occurred and searches backwards through the call stack until it finds a method that contains an *appropriate* exception handler *(catch block)*.

An exception handler is *appropriate* if the type of the exception thrown is the same as the type of exception handled by the handler, or is a subclass of the type of exception handled by the handler.

Thus, the requirement to handle an exception that has been thrown progresses up through the call stack until an appropriate handler is found to handle the exception. If no appropriate handler is found, the runtime system and the program terminate.

(If you have ever had a program terminate with a ***NullPointerException*** *, then you know how program termination works).*

According to the jargon, the exception handler that is chosen is said to *catch the exception.*

Advantages of using exception handling

According to Campione and Walrath, exception handling provides the following advantages over "traditional" error management techniques:

- Separating Error Handling Code from "Regular" Code
- Propagating Errors Up the Call Stack
- Grouping Error Types and Error Differentiation

Separating error handling code from regular code

I don't plan to discuss these advantages in detail. Rather, I will simply refer you to The Java Tutorial [41] and other good books where you can read their discussions. However, I will comment briefly.

Campione and Walrath provide a good illustration where they show how a simple program having about six lines of code get "bloated" into about 29 lines of very confusing code through the use of traditional error management techniques. Not only does the program suffer bloat, the logical flow of the original program gets lost in the clutter of the modified program.

They then show how to accomplish the same error management using exception handling. Although the version with exception handling contains about seventeen lines of code, it is orderly and easy to understand. The additional lines of code do not cause the original logic of the program to get lost.

You must still do the hard work

However, the use of exception handling does not spare you from the hard work of detecting, reporting, and handling errors. What it does is provide a means to separate the details of what to do when something out-of-the-ordinary happens from the normal logical flow of the program code.

Propagating exceptions up the call stack

Sometimes it is desirable to propagate exception handling up the call stack and let the corrective action be taken at a higher level.

For example, you might provide a class with methods that implement a *stack* . One of the methods of your class might be to *pop* an element off the stack.

What should your program do if a using program attempts to pop an element off an empty stack? That decision might best be left to the user of your stack class, and you might simply propagate the notification up to the calling method and let that method take the corrective action.

Grouping exception types

[41] http://java.sun.com/docs/books/tutorial/

When an exception is thrown, an object of one of the exception classes is passed as a parameter. Objects are instances of classes, and classes fall into an inheritance hierarchy in Java. Therefore, a natural hierarchy can be created, which causes exceptions to be grouped in logical ways.

For example, going back to the stack example, you might create an exception class that applies to all exceptional conditions associated with an object of your stack class. Then you might extend that class into other classes that pertain to specific exceptional conditions, such as *push* exceptions, *pop* exceptions, and *initialization* exceptions.

When your code throws an exception object of a specific type, that object can be caught by an exception handler designed either to:

- Catch on the basis of a group of exceptions, or
- Catch on the basis of a subgroup of that group, or
- Catch on the basis of one of the specialized exceptions.

In other words, an exception handler can catch exceptions of the class specified by the type of its parameter, or can catch exceptions of any subclass of the class specified by the type of its parameter.

More detailed information on exception handling

As explained earlier, except for **Throwable** objects of type **Error** and for **Throwable.Exception** objects of type **RuntimeException** , Java programs must either *handle* or *declare* all **Exception** objects that are thrown. Otherwise, the compiler will refuse to compile the program.

In other words, all exceptions other than those specified above are *checked* by the compiler, and the compiler will refuse to compile the program if the exceptions aren't handled or declared. As a result, exceptions other than those specified above are often referred to as *checked* exceptions.

Catching an exception

Just to make certain that we are using the same terminology, a method *catches* an exception by providing an exception handler whose parameter type is appropriate for that type of exception object. *(I will more or less use the terms* ***catch*** *block and exception handler interchangeably.)*

The type of the parameter in the **catch** block must be the class from which the exception was instantiated, or a superclass of that class that resides somewhere between that class and the **Throwable** class in the inheritance hierarchy.

Declaring an exception

If the code in a method can throw a checked exception, and the method does not provide an exception handler for the type of exception object thrown, the method must *declare* that it can throw that exception. The **throws** keyword is used in the method declaration to declare that it **throws** an exception of a particular type.

Any checked exception that can be thrown by a method is part of the method's programming interface *(see the* ***read*** *method of the InputStream class, which throws* ***IOException*** *, for example).* Users of a method must know about the exceptions that a method can throw in order to be able to handle them. Thus, you must declare the exceptions that the method can throw in the method signature.

Checked exceptions

Checked exceptions are all exception objects instantiated from subclasses of the **Exception** class other than those of the **RuntimeException** class.

Exceptions of all **Exception** subclasses other than **RuntimeException** are checked by the compiler and will result in compiler errors if they are neither *caught* nor *declared* .

You will learn how you can create your own exception classes later. Whether your exception objects become checked or not depends on the class that you extend when you define your exception class.

(If you extend a checked exception class, your new exception type will be a checked exception. Otherwise, it will not be a checked exception.)

Exceptions that can be thrown within the scope of a method

The exceptions that can be thrown within the scope of a method include not only exceptions which are thrown by code written into the method, but also includes exceptions thrown by methods called by that method, or methods called by those methods, etc.

According to Campione and Walrath,

"This ... includes any exception that can be thrown while the flow of control remains within the method. Thus, this ... includes both exceptions that are thrown directly by the method with Java's throw statement, and exceptions that are thrown indirectly by the method through calls to other methods."

Sample programs

Now it's time to take a look at some sample code designed to deal with exceptions of the types delivered with the JDK. Initially I won't include exception classes that are designed for custom purposes. However, I will deal with exceptions of those types later in the module.

The first three sample programs will illustrate the successive stages of dealing with checked exceptions by either catching or declaring those exceptions.

Sample program with no exception handling code

The first sample program shown in Listing 1 (p. 797) neither catches nor declares the **InterruptedException** which can be thrown by the **sleep** method of the **Thread** class.

Listing 1: Sample program with no exception handling code.

```
    /*File Excep11.java
Copyright 2002, R.G.Baldwin
Tested using JDK 1.4.0 under Win2000
**************************************/
import java.lang.Thread;

class Excep11{
  public static void main(
                        String[] args){
    Excep11 obj = new Excep11();
    obj.myMethod();
  }//end main
  //---------------------------------//

  void myMethod(){
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep11
```

A possible InterruptedException

The code in the **main** method of Listing 1 (p. 797) calls the method named **myMethod** . The method named **myMethod** calls the method named **sleep** of the **Thread** class. The method named **sleep** declares that it throws **InterruptedException** .

InterruptedException is a checked exception. The program illustrates the failure to either catch or declare **InterruptedException** in the method named **myMethod** .

As a result, this program won't compile. The compiler error is similar to that shown in Image 3 (p. 798) . Note the caret in the last line that points to the point where the compiler detected the problem.

Image 3: Compiler error from an unhandled checked exception.

```
    unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown
    Thread.currentThread().sleep(1000);
                          ^
```

5.154

As you can see, the compiler detected a problem where the **sleep** method was called, because the method named **myMethod** failed to deal properly with an exception that can be thrown by the **sleep** method.

Sample program that fixes one compiler error

The next version of the program, shown in Listing 2 (p. 798) , fixes the problem identified with the call to the **sleep** method, by declaring the exception in the signature for the method named **myMethod** .

Listing 2: Sample program that fixes one compiler error.

```
    /*File Excep12.java
Copyright 2002, R.G.Baldwin
Tested using JDK 1.4.0 under Win2000
**************************************/
import java.lang.Thread;

class Excep12{
  public static void main(
                        String[] args){
    Excep12 obj = new Excep12();
    obj.myMethod();
  }//end main
  //---------------------------------//

  void myMethod()
           throws InterruptedException{
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep12
```

5.155

Another possible InterruptedException

As was the case in the previous program, this program also illustrates a failure to catch or declare an

InterruptedException . However, in this case, the problem has moved up one level in the call stack relative to the problem with the program in Listing 1 (p. 797) .

This program also fails to compile, producing a compiler error similar to that shown in Image 4 (p. 799) . Note that the caret indicates that the problem is associated with the call to **myMethod** .

Image 4: Another compiler error.

```
    unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown
    obj.myMethod();
       ^
```

5.156

Didn't solve the problem

Simply declaring a checked exception doesn't solve the problem. Ultimately, the exception must be handled if the compiler problem is to be solved.

(Note, however, that it is possible to declare that the ***main*** *method throws a checked exception, which will cause the compiler to ignore it and allow your program to compile.)*

The program in Listing 2 (p. 798) eliminated the compiler error identified with the call to the method named **sleep** . This was accomplished by declaring that the method named **myMethod** *throws InterruptedException* . However, this simply passed the exception up the call stack to the next higher-level method in the stack. This didn't solve the problem, it simply handed it off to another method to solve.

The problem still exists, and is now identified with the call to **myMethod** where it will have to be handled in order to make the compiler error go away.

Sample program that fixes the remaining compiler error

The version of the program shown in Listing 3 (p. 800) fixes the remaining compiler error. This program illustrates both declaring and handling a checked exception. This program compiles and runs successfully.

Listing 3: Sample program that fixes the remaining compiler error.

```
    /*File Excep13.java
Copyright 2002, R.G.Baldwin

Tested using JDK 1.4.0 under Win2000
**************************************/
import java.lang.Thread;

class Excep13{
  public static void main(
                        String[] args){
    Excep13 obj = new Excep13();
    try{//begin try block
      obj.myMethod();
    }catch(InterruptedException e){
      System.out.println(
              "Handle exception here");
    }//end catch block
  }//end main
  //---------------------------------//

  void myMethod()
           throws InterruptedException{
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep13
```

5.157

The solution to the problem

This solution to the problem is accomplished by surrounding the call to **myMethod** with a **try** block, which is followed immediately by an *appropriate* **catch** block. In this case, an appropriate **catch** block is one whose parameter type is either **InterruptedException** , or a superclass of **InterruptedException** .

(Note, however, that the superclass cannot be higher than the ***Throwable*** *class in the inheritance hierarchy.)*

The myMethod method declares the exception

As in the previous version, the method named **myMethod** (declares the exception and passes it up the call stack to the method from which it was called.

The main method handles the exception

In the new version shown in Listing 3 (p. 800) , the **main** method provides a **try** block with an *appropriate* **catch** block for dealing with the problem *(although it doesn't actually deal with it in any significant way).* This can be interpreted as follows:

- Try to execute the code within the **try** block.
- If an exception occurs, search for a **catch** block that matches the type of object thrown by the exception.

- If such a **catch** block can be found, immediately transfer control to the catch block without executing any of the remaining code in the **try** block.

(For simplicity, this program didn't have any remaining code. Some later sample programs will illustrate code being skipped due to the occurrence of an exception.)

Not a method call

Note that this transfer of control is not a method call. It is an unconditional transfer of control. There is no *return* from a catch block.

Matching catch block was found

In this case, there was a matching **catch** block to receive control. In the event that an **InterruptedException** is thrown, the program would execute the statement within the body of the **catch** block, and then transfer control to the code following the final **catch** block in the group of **catch** blocks *(in this case, there was only one **catch** block).*

No output is produced

It is unlikely that you will see any output when you run this program, because it is unlikely that an **InterruptedException** will be thrown. *(I didn't provide any code that will cause such an exception to occur.)*

A sample program that throws an exception

Now let's look at the sample program in Listing 4 (p. 802) , which throws an exception and deals with it. This program illustrates the implementation of exception handling using the try/catch block structure.

Listing 4: A sample program that throws an exception.

```
    /*File Excep14.java
Copyright 2002, R. G. Baldwin

Tested with JDK 1.4.0 under Win2000
**************************************/

class Excep14{
  public static void main(
                       String[] args){
    try{
      for(int cnt = 2; cnt >-1; cnt--){
        System.out.println(
               "Running. Quotient is: "
                             + 6/cnt);
      }//end for-loop
    }//end try block
    catch(ArithmeticException e){
      System.out.println(
              "Exception message is:  "
              + e.getMessage()
              + "\nStacktrace shows:");
      e.printStackTrace();
      System.out.println(
        "String representation is\n " +
                         e.toString());
      System.out.println(
         "Put corrective action here");
    }//end catch block
    System.out.println(
                "Out of catch block");
  }//end main

}//end class Excep14
```


Keeping it simple

I try to keep my sample programs as simple as possible, introducing the minimum amount of complexity necessary to illustrate the main point of the program. It is easy to write a *really simple* program that throws an unchecked **ArithmeticException** . Therefore, the program in Listing 4 (p. 802) was written to throw an **ArithmeticException** . This was accomplished by trying to perform an integer divide by zero.

The try/catch structure is the same ...

It is important to note that the *try/catch* structure illustrated in Listing 4 (p. 802) would be the same whether the exception is checked or unchecked. The main difference is that you are not required by the

compiler to handle unchecked exceptions and you are required by the compiler to either handle or declare checked exceptions.

Throwing an ArithmeticException

The code in Listing 4 (p. 802) executes a simple counting loop inside a **try** block. During each iteration, the counting loop divides the integer 6 by the value of the counter. When the value of the counter goes to zero, the runtime system tries to perform an integer divide by zero operation, which causes it to throw an **ArithmeticException** .

Transfer control immediately

At that point, control is transferred directly to the **catch** block that follows the **try** block. This is an *appropriate* **catch** block because the type of parameter declared for the **catch** block is **ArithmeticException** . It matches the type of the object that is thrown.

(It would also be appropriate if the declared type of the parameter were a superclass of ***ArithmeticException*** *, up to and including the class named* ***Throwable*** *.* ***Throwable*** *is a direct subclass of* ***Object*** *. If you were to declare the parameter type for the* ***catch*** *block as* ***Object*** *, the compiler would produce an incompatible type error.)*

Calling methods inside the catch block

Once control enters the **catch** block, three of the methods of the **Throwable** class are called to cause information about the situation to be displayed on the screen. The output produced by the program is similar to that shown in Image 5 (p. 803) .

Image 5: Output from program that throws ArithmeticException.

```
    Running. Quotient is: 3
Running. Quotient is: 6
Exception message is:  / by zero
Stacktrace shows:
java.lang.ArithmeticException:
  / by zero
  at Excep14.main(Excep14.java:35)
String representation is
java.lang.ArithmeticException:
/ by zero
Put corrective action here
Out of catch block
```

Key things to note

The key things to note about the code in Listing 4 (p. 802) and the output in Image 5 (p. 803) are:

- The code to be protected is contained in a **try** block.
- The **try** block is followed immediately by an appropriate **catch** block.
- When an exception is thrown within the **try** block, control is transferred immediately to the **catch** block with the matching or appropriate parameter type.
- Although the code in the **catch** block simply displays the current state of the program, it could contain code that attempts to rectify the problem.
- Once the code in the **catch** block finishes executing, control is passed to the next executable statement following the **catch** block, which in this program is a print statement.

Doesn't attempt to rectify the problem

This program doesn't attempt to show how an actual program might recover from an exception of this sort. However, it is clear that *(rather than experiencing automatic and unconditional termination)* the program remains in control, and in some cases, recovery might be possible.

This sample program illustrates **try** and **catch** . The use of **finally** , will be discussed and illustrated later.

A nuisance problem explained

While we are at it, I would be remiss in failing to mention a nuisance problem associated with exception handling.

As you may recall, the scope of a variable in Java is limited to the block of code in which it is declared. A block is determined by enclosing code within a pair of matching curly brackets: {...}.

Since a pair of curly brackets is required to define a **try** block, the scope of any variables or objects declared inside the **try** block is limited to the **try** block.

While this is not an insurmountable problem, it may require you to modify your programming style in ways that you find distasteful. In particular, if you need to access a variable both within and outside the **try** block, you must declare it before entering the **try** block.

The process in more detail

Now that you have seen some sample programs to help you visualize the process, lets discuss the process in more detail.

The try block

According to Campione and Walrath,

"The first step in writing any exception handler is putting the Java statements within which an exception can occur into a try block. The try block is said to govern the statements enclosed within it and defines the scope of any exception handlers (established by subsequent catch blocks) associated with it."

Note that the terminology being used by Campione and Walrath treats the **catch block** as the *"exception handler"* and treats the **try** block as something that precedes one or more exception handlers. I don't disagree with their terminology. I mention it only for the purpose of avoiding confusion over terminology.

The syntax of a try block

The general syntax of a **try** block, as you saw in the previous program, has the keyword **try** followed by one or more statements enclosed in a pair of matching curly brackets, as shown in Image 6 (p. 804) .

Image 6: Syntax of a try block.

```
    try{
  //java statements
}//end try block
```

5.160

Single statement and multiple exceptions

You may have more than one statement that can throw one or more exceptions and you will need to deal with all of them.

You could put each such statement that might throw exceptions within its own **try** block and provide separate exception handlers for each **try** block.

(Note that some statements, particularly those that call other methods, could potentially throw many different types of exceptions.)

Thus a **try** block consisting of a single statement might require many different exception handlers or **catch** blocks following it.

Multiple statements and multiple exceptions

You could put all or several of the statements that might throw exceptions within a single **try** block and associate multiple exception handlers with it. There are a number of practical issues involved here, and only you can decide in any particular instance which approach would be best.

The catch blocks must follow the try block

However you decide to do it, the exception handlers associated with a **try** block must be placed immediately following their associated *try* block. If an exception occurs within the **try** block, that exception is handled by the appropriate exception handler associated with the **try** block. If there is no appropriate exception handler associated with the **try** block, the system attempts to find an appropriate exception handler in the next method up the call stack.

A **try** block must be accompanied by at least one **catch** block *(or one **finally** block)*. Otherwise, a compiler error that reads something like *'try' without 'catch' or 'finally'* will occur.

The catch block(s)

Continuing with what Campione and Walrath have to say:

"Next, you associate exception handlers with a try block by providing one or more catch blocks directly after the try block."

There can be no intervening code between the end of the **try** block and the beginning of the first **catch** block, and no intervening code between **catch** blocks.

Syntax of a catch block

The general form of a **catch** block is shown in Image 7 (p. 805) .

Image 7: Syntax of a catch block.

```
    catch(ThrowableObjectType paramName){
  //Java statements to handle the
  // exception
}//end catch block
```

5.161

The declaration for the **catch** block requires a single argument as shown. The syntax for the argument declaration is the same as an argument declaration for a method.

Argument type specifies type of matching exception object

The argument type declares the type of exception object that a particular **catch** block can handle. The type must be **Throwable** , or a subclass of the **Throwable** class discussed earlier.

A parameter provides the local name

Also, as in a method declaration, there is a parameter, which is the name by which the handler can refer to the exception object. For example, in an earlier program, I used statements such as **e.getMessage()** to access an instance method of an exception object caught by the exception handler. In that case, the name of the parameter was **e** .

You access the instance variables and methods of exception objects the same way that you access the instance variables and methods of other objects.

Proper order of catch blocks

According to Campione and Walrath:

"The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is called. The runtime system calls the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown."

Therefore, the order of your exception handlers is very important, particularly if you have some handlers, which are further up the exception hierarchy than others.

Those handlers that are designed to handle exception types furthermost from the root of the hierarchy tree *(**Throwable**)* should be placed first in the list of exception handlers.

Otherwise, an exception handler designed to handle a specific type of object may be preempted by another handler whose exception type is a superclass of that type, if the superclass exception handler appears earlier in the list of exception handlers.

Catching multiple exception types with one handler

Exception handlers that you write may be more or less specialized. In addition to writing handlers for very specialized exception objects, the Java language allows you to write general exception handlers that handle multiple types of exceptions.

A hierarchy of Throwable classes

Java exceptions are **Throwable** objects *(instances of the **Throwable** class or a subclass of the **Throwable** class).*

The Java standard library contains numerous classes that are subclasses of **Throwable** and thus build a hierarchy of **Throwable** classes.

According to Campione and Walrath:

*"Your exception handler can be written to handle any class that inherits from **Throwable** . If you write a handler for a "leaf" class (a class with no subclasses), you've written a specialized handler: it will only handle exceptions of that specific type. If you write a handler for a "node" class (a class with subclasses), you've written a general handler: it will handle any exception whose type is the node class or any of its subclasses."*

You have a choice

Therefore, when writing exception handlers, you have a choice. You can write a handler whose exception type corresponds to a node in the inheritance hierarchy, and it will be appropriate to **catch** exceptions of that type, or any subclass of that type.

Alternately, you can write a handler whose exception type corresponds to a *leaf,* in which case, it will be appropriate to **catch** exceptions of that type only.

And finally, you can mix and match, writing some exception handlers whose type corresponds to a node, and other exception handlers whose type corresponds to a leaf. In all cases, however, be sure to position your exception handlers in reverse subclass order, with the furthermost subclass from the root appearing first, and the root class appearing last.

The finally block

And finally *(no pun intended),* Campione and Walrath tell us:

"Java's finally block provides a mechanism that allows your method to clean up after itself regardless of what happens within the try block. Use the finally block to close files or release other system resources."

To elaborate, the **finally** block can be used to provide a mechanism for cleaning up open files, etc., before allowing control to be passed to a different part of the program. You accomplish this by writing the cleanup code within a **finally** block.

Code in finally block is always executed

It is important to remember that the runtime system always executes the code within the **finally** block regardless of what happens within the **try** block.

If no exceptions are thrown, none of the code in **catch** blocks is executed, but the code in the **finally** block is executed.

If an exception is thrown and the code in an exception handler is executed, once the execution of that code is complete, control is passed to the **finally** block and the code in the **finally** block is executed.

*(There is one important exception to the above. If the code in the **catch** block terminates the program by executing **System.exit(0)** , the code in the **finally** block will not be executed.)*

The power of the finally block

The sample program shown in Listing 5 (p. 807) illustrates the power of the **finally** block.

Listing 5: The power of the finally block.

```
    /*File Excep15.java
Copyright 2002, R. G. Baldwin

Tested with JDK 1.4.0 under Win2000
**************************************/

class Excep15{
  public static void main(
                        String[] args){
    new Excep15().aMethod();
  }//end main
  //---------------------------------//

  void aMethod(){
    try{
      int x = 5/0;
    }//end try block
    catch(ArithmeticException e){
      System.out.println(
      "In catch, terminating aMethod");
      return;
    }//end catch block

    finally{
      System.out.println(
            "Executing finally block");
    }//end finally block

    System.out.println(
                "Out of catch block");
  }//end aMethod

}//end class Excep15
```

5.162

Execute return statement in catch block

The code in Listing 5 (p. 807) forces an **ArithmeticException** by attempting to do an integer divide by zero. Control is immediately transferred to the matching **catch** block, which prints a message and then executes a **return** statement.

Normally, execution of a **return** statement terminates the method immediately. In this case, however, before the method terminates and returns control to the calling method, the code in the **finally** block is

executed. Then control is transferred to the **main** method, which called this method in the first place.

Image 8 (p. 808) shows the output produced by this program.

Image 8: Output produced by the finally block.

```
    In catch, terminating aMethod
Executing finally block
```

5.163

This program demonstrates that the **finally** block really does have the final word.

Declaring exceptions thrown by a method

Sometimes it is better to handle exceptions in the method in which they are detected, and sometimes it is better to pass them up the call stack and let another method handle them.

In order to pass exceptions up the call stack, you must *declare* them in your method signature.

To *declare* that a method throws one or more exceptions, you add a **throws** clause to the method signature for the method. The **throws** clause is composed of the **throws** keyword followed by a comma-separated list of all the exceptions thrown by that method.

The **throws** clause goes after the method name and argument list and before the curly bracket that defines the scope of the method.

Image 9 (p. 808) shows the syntax for declaring that a method **throws** four different types of exceptions.

Image 9: Syntax for declaring that a method throws exceptions.

```
    void myMethod() throws
          InterruptedException,
          MyException,
          HerException,
          UrException
{
  //method code
}//end myMethod()
```

5.164

Assuming that these are checked exceptions, any method calling this method would be required to either handle these exception types, or continue passing them up the call stack. Eventually, some method must handle them or the program won't compile.

(Note however that while it might not represent good programming practice, it is allowable to declare that the ***main*** *method* ***throws*** *exceptions. This is a way to avoid handling checked exceptions and still get your program to compile.)*

The throw keyword

Before your code can **catch** an exception, some Java code must **throw** one. The exception can be thrown by code that you write, or by code that you are using that was written by someone else.

Regardless of who wrote the code that throws the exception, it's always thrown with the Java **throw** keyword. At least that is true for exceptions that are thrown by code written in the Java language.

(Exceptions such as ***ArithmeticException*** *are also thrown by the runtime system, which is probably not written using Java source code.)*

A single argument is required

When formed into a statement, the **throw** keyword requires a single argument, which must be a reference to an object instantiated from the **Throwable** class, or any subclass of the **Throwable** class. Image 10 (p. 809) shows an example of such a statement.

Image 10: Example of a throw statement.

```
throw new myThrowableClass("Message");
```

5.165

If you attempt to throw an object that is not instantiated from **Throwable** or one of its subclasses, the compiler will refuse to compile your program.

Defining your own exception classes

Now you know how to write exception handlers for those exception objects that are thrown by the runtime system, and thrown by methods in the standard class library.

It is also possible for you to define your own exception classes, and to cause objects of those classes to be thrown whenever an exception occurs. In this case, you get to decide just what constitutes an exceptional condition.

For example, you could write a data-processing application that processes integer data obtained via a TCP/IP link from another computer. If the specification for the program indicates that the integer value 10 should never be received, you could use an occurrence of the integer value 10 to cause an exception object of your own design to be thrown.

Choosing the exception type to throw

Before throwing an exception, you must decide on its type. Basically, you have two choices in this regard:

- Use an exception class written by someone else, such as the myriad of exception classes defined in the Java standard library.
- Define an exception class of your own.

An important question

So, an important question is, when should you define your own exception classes and when should you use classes that are already available. Because this is only one of many design issues, I'm not going to try to give you a ready answer to the question. However, I will refer you to The Java Tutorial [42] by Campione and Walrath where you will find a checklist to help you make this decision.

Choosing a superclass to extend

If you decide to define your own exception class, it must be a subclass of **Throwable** . You must decide which class you will extend.

The two existing subclasses of **Throwable** are **Exception** and **Error** . Given the earlier description of **Error** and its subclasses, it is not likely that your exceptions would fit the **Error** category. *(In concept, errors are reserved for serious hard errors that occur deep within the system.)*

Checked or unchecked exception

[42] http://java.sun.com/docs/books/tutorial/

Therefore, your new class should probably be a subclass of **Exception** . If you make it a subclass of **RuntimeException** , it won't be a checked exception. If you make it a subclass of **Exception** , but not a subclass of **RuntimeException** , it will be a checked exception.

Only you can decide how far down the **Exception** hierarchy you want to go before creating a new branch of exception classes that are unique to your application.

Naming conventions

Many Java programmers append the word **Exception** to the end of all class names that are subclasses of **Exception** , and append the word **Error** to the end of all class names that are subclasses of **Error** .

One more sample program

Let's wrap up this module with one more sample program named **Excep16** . We will define our own exception class in this program. Then we will **throw** , **catch and** process an exception object instantiated from that class.

Discuss in fragments

This program is a little longer than the previous programs, so I will break it down and discuss it in fragments. A complete listing of the program is shown in Listing 10 (p. 814) .

The class definition shown in Listing 6 (p. 810) is used to construct a custom exception object that encapsulates a message. Note that this class extends **Exception** . *(Therefore, it is a checked exception.)*

Listing 6: The class named MyException .

```
    class MyException extends Exception{
  MyException(String message){//constr
    super(message);
  }//end constructor
}//end MyException class
```

5.166

The constructor for this class receives an incoming **String** message parameter and passes it to the constructor for the superclass. This makes the message available for access by the **getMessage** method called in the catch block.

The try block

Listing 7 (p. 811) shows the beginning of the **main** method, including the entire **try** block

Listing 7: The try block.

```
  class Excep16{//controlling class
public static void main(
                     String[] args){
  try{
    for(int cnt = 0; cnt < 5; cnt++){
      //Throw a custom exception, and
      // pass  message when cnt == 3
      if(cnt == 3) throw
                new MyException("3");
      //Transfer control before
      // processing for cnt == 3
      System.out.println(
         "Processing data for cnt = "
                              + cnt);
    }//end for-loop
  }//end try block
```

5.167

The **main** method executes a **for** loop *(inside the **try** block)* that guarantees that the variable named **cnt** will reach a value of 3 after a couple of iterations.

Once during each iteration, *(until the value of **cnt** reaches 3)* a print statement inside the **for** loop displays the value of **cnt** . This results in the output shown in Image 11 (p. 811) .

Image 11: Output from the for loop.

```
   Processing data for cnt = 0
Processing data for cnt = 1
Processing data for cnt = 2
```

5.168

What happens when cnt equals 3?

However, when the value of **cnt** equals 3, the **throw** statement in Listing 7 (p. 811) is executed. This causes control to transfer immediately to the matching **catch** block following the **try** block *(see Listing 8 (p. 812))*. During this iteration, the print statement following the **throw** statement is not executed. Therefore, the output never shows a value for **cnt** greater than 2, as shown in Image 11 (p. 811) .

The catch block

Listing 8 (p. 812) shows a **catch** block whose type matches the type of exception thrown in Listing 7 (p. 811) .

Listing 8: A matching catch block.

```
    catch(MyException e){
   System.out.println(
            "In exception handler, "
               + "get the message\n"
                  + e.getMessage());
 }//end catch block
```

5.169

When the **throw** statement is executed in Listing 7 (p. 811) , control is transferred immediately to the **catch** block in Listing 8 (p. 812) . No further code is executed within the **try** block.

A reference to the object instantiated as the argument to the **throw** keyword in Listing 7 (p. 811) is passed as a parameter to the **catch** block. That reference is known locally by the name **e** inside the **catch** block.

Using the incoming parameter

The code in the **catch** block calls the method named **getMessage** *(inherited from the* ***Throwable*** *class)* on the incoming parameter and displays that message on the screen. This produces the output shown in Image 12 (p. 812) .

Image 12: Output from the exception handler.

```
    In exception handler, get the message
3
```

5.170

When the catch block finishes execution ...

When the code in the **catch** block has completed execution, control is transferred to the first executable statement following the **catch** block as shown in Listing 9 (p. 812) .

Listing 9: Code following the catch block.

```
      System.out.println(
     "Out of catch block");
  }//end main
}//end class Excep16
```

5.171

That executable statement is a print statement that produces the output shown in Image 13 (p. 813) .

Image 13: Output from code following the catch block.

```
Out of catch block
```

5.172

Complete program listing

A complete listing of the program named **Excep16** is shown in Listing 10 (p. 814) .

Listing 10: Complete program listing for Excep16.

```
    /*File Excep16.java
Copyright 2002, R. G. Baldwin
Illustrates defining, throwing,
catching, and processing a custom
exception object that contains a
message.

Tested using JDK 1.4.0 under Win 2000

The output is:

Processing data for cnt = 0

Processing data for cnt = 1
Processing data for cnt = 2
In exception handler, get the message
3
Out of catch block
**************************************/

//The following class is used to
// construct a customized exception
// object containing a message
class MyException extends Exception{
  MyException(String message){//constr
    super(message);
  }//end constructor
}//end MyException class
//====================================//

class Excep16{//controlling class
  public static void main(
                        String[] args){
    try{
      for(int cnt = 0; cnt < 5; cnt++){
        //Throw a custom exception, and
        // pass  message when cnt == 3
        if(cnt == 3) throw
                  new MyException("3");
        //Transfer control before
        // processing for cnt == 3
        System.out.println(
           "Processing data for cnt = "
                                + cnt);
      }//end for-loop
    }//end try block
    catch(MyException e){
      System.out.println(
                "In exception handler, "
           + "get the message\n"
                  + e.getMessage());
    }//end catch block
    //-------------------------------//

    System.out.println(
```

5.2.16.5 Summary

This module has covered many of the details having to do with exception handling in Java. By now, you should know that the use of exception handling is not optional in Java, and you should have a pretty good idea how to use exception handling in a beneficial way.

Along the way, the discussion has included the following topics:

- What is an exception?
- How do you throw and catch exceptions?
- What do you do with an exception once you have caught it?
- How do you make use of the exception class hierarchy provided by the Java development environment?

5.2.16.6 What's next?

That concludes the "Essence of OOP" portion of the ITSE 2321 sub-collection. The series is continued in the ITSE 2317 sub-collection.

5.2.16.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Exception Handling
- File: Java1630.htm
- Published: 09/03/02
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3 Multimedia

5.3.1 Java3000: The Guzdial-Ericson Multimedia Class Library[43]

5.3.1.1 Table of Contents

[43] This content is available online at <http://cnx.org/content/m44148/1.7/>.

5.3.1.2 Preface

In the weeks and months to come, I will publish several modules that use a multimedia class library developed and made available by Mark Guzdial [44] and Barbara Ericson [45] of the Georgia Institute of Technology. The purpose of this module is to provide the information that you will need to obtain and use a copy of that library.

5.3.1.3 Images and listings

5.3.1.3.1 Images

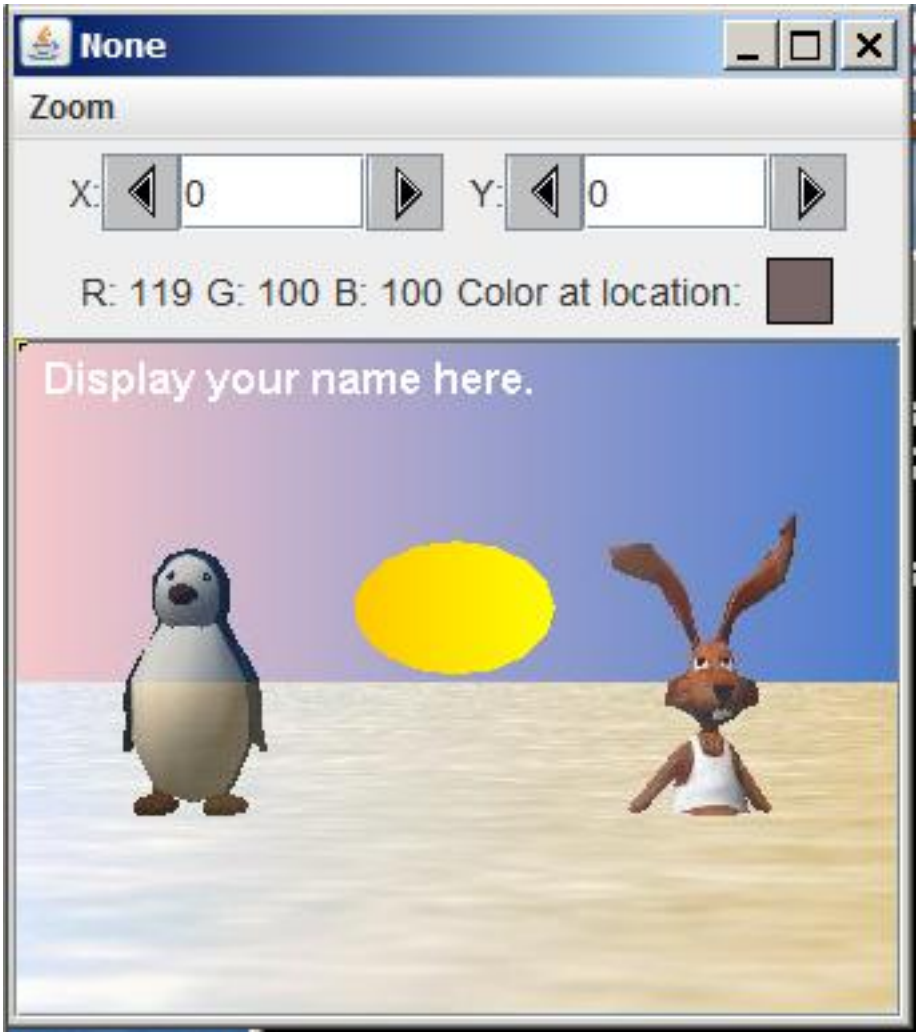
5.3.1.3.2 Listings

5.3.1.4 Preview

Among other things, the Guzdial-Ericson multimedia class library makes it practical to write object-oriented programs for the manipulation of images in an interesting and educational way. For example, the image in Image 1 (p. 817) was produced by manipulating and merging the images shown in Image 2 (p. 818) and Image 3 (p. 819) .

[44] http://www.cc.gatech.edu/~guzdial/
[45] http://coweb.cc.gatech.edu/ice-gt/8

Image 1: Result of merging two images.



5.174

One of the input images is shown in Image 2 (p. 818) .

Image 2: Input image #1.

5.175

The other input image is shown in Image 3 (p. 819) .

Image 3: Input image #2.

5.176

5.3.1.5 Discussion

The Guzdial-Ericson library was originally published by Guzdial and Ericson in conjunction with their book Introduction to Computing and Programming with Java: A Multimedia Approach [46] . While the book isn't free, the library is freely available and is published under a Creative Commons Attribution 3.0 United States License [47] .

As of July, 2012, the latest version of the library can be downloaded in a zip file named **bookClasses-3-9-10-with-doc.zip** at http://home.cc.gatech.edu/TeaParty/47. [48] Additional information is available at http://coweb.cc.gatech.edu/mediaComp-plan/101 [49] .

Download the library

In order to work with the programs in the modules that use the library, you will need to download a copy of the latest version of the library from the site listed above. To guard against the possibility of that

[46] http://www.pearsonhighered.com/educator/academic/product/0,3110,0131496980,00.html
[47] http://creativecommons.org/licenses/by/3.0/us/
[48] http://home.cc.gatech.edu/TeaParty/47.
[49] http://coweb.cc.gatech.edu/mediaComp-plan/101

link becoming broken at some point in the future, I am depositing a backup copy of the zip file containing the library on the cnx.org site and you can download it here [50] .

Prepare the library for use

Once you have downloaded the zip file, you will need to extract the folder named **bookClasses** from the zip file and store it somewhere on your computer's disk.

Library documentation

When you examine the contents of the **bookClasses** folder, you will see that in addition to source code and compiled class files for the library, the folder also contains **javadoc** documentation for the library in a folder named **doc** and some other material as well. Go to the **doc** folder and open the file named **index.html** in your browser to view the documentation.

Put the library on the classpath

You will need to put the path to the **bookClasses** folder on your classpath in order to incorporate classes from the library into your programs.

Image 4: Putting the library on the classpath.

```
    del *.class
javac -cp .;M:\bookClasses *.java
java -cp .;M:\bookClasses Prob01
```

5.177

For example, Image 4 (p. 820) shows three commands that you can execute at a Windows command prompt to

- delete the class files from the current folder,
- compile the source-code files in the current folder, and
- execute the compiled version of a program named **Prob01** in the current folder

*(The commands shown in Image 4 (p. 820) assume that the **bookClasses** folder is in the root directory on the M-drive. The **bookClasses** folder will likely be in a different location on your computer. They also assume that you are compiling and running a program having its **main** method in a class named **Prob01** in a file named **Prob01.java** .)*

Be sure to include the period and the semicolon shown before the M in Image 4 (p. 820) . This tells the Java compiler and the Java Virtual Machine to search first in the current folder before searching elsewhere on the disk for the required files.

You can also put the three commands in a batch file and run the batch file to avoid having to type the three commands each time you need to compile and run the program. In that case, you should add a **pause** command following the other three commands.

If you are using an IDE such as DrJava, JCreator, or JGrasp, see the instructions for the IDE to learn how to set the classpath in the IDE.

5.3.1.6 Two versions of Hello World

It is often useful to have the code for a simple program to test your system. This section presents two versions of the classical *Hello World* program. The first version is a simple text version that doesn't use

[50] http://cnx.org/content/m44148/latest/bookClasses-3-9-10-with-doc.zip

Ericson's library. This version can be used to confirm that the Java Development Kit (JDK) is properly installed on your computer.

The second version is a graphic version that does require Ericson's library. This version can be used to confirm proper installation and use of the library.

5.3.1.6.1 A text version of Hello World

The simple program shown in Listing 1 (p. 821) causes the words ***Text Hello World*** to be displayed on the standard output device when the program is compiled and executed.

Listing 1: A text version of Hello World.

```
class TextHelloWorld{
  public static void main(String[] args){
    System.out.println("Text Hello World");
  }//end main
}//end class
```

5.178

As mentioned above, this program does not require the use of Ericson's library. It can be compiled and executed using a batch file containing the commands shown in Image 5 (p. 821) . You should make certain that you can compile and execute this program before progressing to the graphic version.

Image 5: Batch file for text version of Hello World.

```
echo off
del *.class
javac TextHelloWorld.java
java TextHelloWorld
pause
```

5.179

5.3.1.6.2 A graphic version of Hello World

Listing 2 (p. 822) contains the code for a graphic version of Hello World.

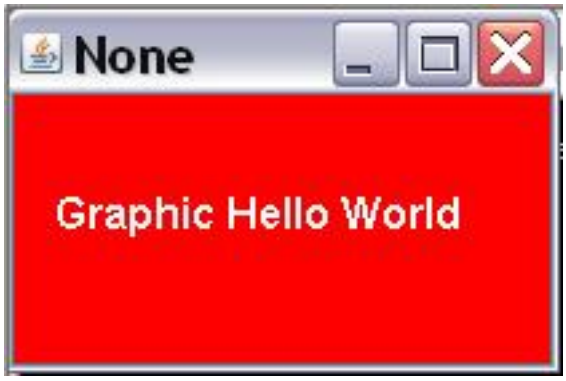
Listing 2: A graphic version of Hello World.

```
import java.awt.Color;

class GraphicHelloWorld{
  public static void main(String[] args){
    Picture pix = new Picture(200,100);
    pix.setAllPixelsToAColor(Color.RED);
    pix.addMessage("Graphic Hello World",15,50);
    pix.show();
  }// end main
}//end class
```

5.180

The program shown in Listing 2 (p. 822) causes the image shown in Image 6 (p. 822) to be displayed when the program is compiled and executed.

Image 6: Hello World in graphics.



5.181

As mentioned above, this program does require the use of Ericson's library. It can be compiled and executed using a batch file containing the commands shown in Image 7 (p. 823) . You should make certain that you can compile and execute this program before progressing to more complex programs involving Ericson's library.

Image 7: Batch file for graphic version of Hello World. .

```
echo off

del *.class
javac -cp .;M:\bookClasses GraphicHelloWorld.java
java -cp .;M:\bookClasses GraphicHelloWorld
pause
```

5.182

(Once again, the commands shown in Image 7 (p. 823) assume that the ***bookClasses*** *folder is in the root directory on the M-drive. The* ***bookClasses*** *folder will likely be in a different location on your computer.)*

5.3.1.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: The Guzdial-Ericson Multimedia Class Library
- File: Java3000.htm
- Published: 07/27/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.2 Java3000r Review[51]

5.3.2.1 Table of Contents

5.3.2.2 Preface

This module contains review questions and answers keyed to the module titled Java3000: The Guzdial-Ericson Multimedia Class Library [52] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.2.3 Questions

5.3.2.3.1 Question 1 .

True or false? The multimedia library used in this course originated at MIT.

Answer 1 (p. 830)

5.3.2.3.2 Question 2

True or false? Among other things, the Guzdial-Ericson multimedia class library makes it practical to write object-oriented programs for the manipulation of images in an interesting and educational way.

Answer 2 (p. 830)

5.3.2.3.3 Question 3

True or false? The textbook for the course is available at no cost.

Answer 3 (p. 830)

5.3.2.3.4 Question 4

True or false? Once you have downloaded the zip file containing the library, you will need to extract the folder named **MediaComp** from the zip file and store it somewhere on your computer's disk.

Answer 4 (p. 830)

5.3.2.3.5 Question 5

True or false? The **bookClasses** folder contains **javadoc** documentation for the library in a folder named **doc** . You can view the documentation by going to the **doc** folder and opening the file named **index.html** in your browser.

Answer 5 (p. 830)

[51] This content is available online at <http://cnx.org/content/m45761/1.4/>.

[52] http://cnx.org/content/m44148

5.3.2.3.6 Question 6

True or false? You will need to put the path to the **bookClasses** folder on your **path** in order to incorporate classes from the library into your programs.

Answer 6 (p. 830)

5.3.2.3.7 Question 7

True or false? Commands similar to those shown in Listing 1 (p. 825) can be used to put the **bookClasses** library on the **classpath** :

Listing 1, Question 7.

```
    javac -cp .;M:\bookClasses *.java
java -cp .;M:\bookClasses Prob01
```

5.183

Answer 7 (p. 829)

5.3.2.3.8 Question 8

True or false? Commands similar to those shown in Listing 2 (p. 825) can be used to:

- delete the class files from the current folder,
- compile the source-code files in the current folder, and
- execute the compiled version of a program named **Prob01** in the current folder

---

Listing 2, Question 8.

```
    del *.class
javac -cp .;M:\bookClasses *.java
java -cp .;M:\bookClasses Prob01
```

5.184

Answer 8 (p. 829)

5.3.2.3.9 Question 9

True or false? You can put commands in a batch file and run the batch file to avoid having to type the commands each time you need to perform a specific operation from the command line. In that case, you should add a **pause** command as the last command in the sequence.

Answer 9 (p. 829)

5.3.2.3.10 Question 10

True or false? The program shown in Listing 3 (p. 826) causes the words ***Text Hello World*** to be displayed on the standard output device when the program is compiled and executed.

Listing 3, Question 10.

```
    class TextHelloWorld{
  public static void main(String[] args){
    System.out.println("Text Hello World");
  }//end main
}//end class
```

5.185

Answer 10 (p. 829)

5.3.2.3.11 Question 11

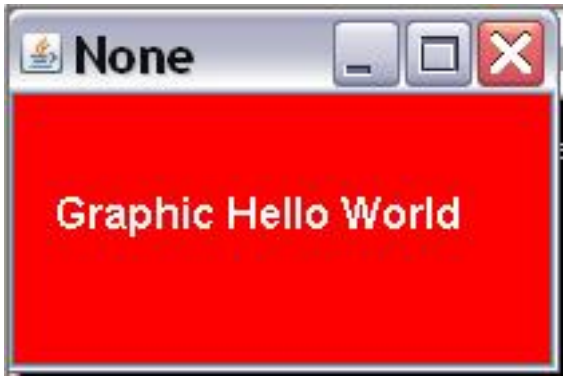
True or false? The code shown in Listing 4 (p. 826) can be compiled and run to produce the graphic output shown in Image 1 (p. 827) .

Listing 4, Question 11.

```
    import java.awt.Color;

class GraphicHelloWorld{
  public static void main(String[] args){
    PictureExplorer pix = new PictureExplorer(200,100);
    pix.setAllPixelsToAColor(Color.RED);
    pix.addMessage("Graphic Hello World",15,50);
    pix.show();
  }// end main
}//end class
```

5.186

Image 1, Question 11.



5.187

Answer 11 (p. 829)

5.3.2.3.12 Question 12

True or false? The program shown *(incorrectly)* in Listing 4 (p. 826) and correctly in Listing 5 (p. 829) requires the use of Ericson's multimedia library plus a jpg image file.

Answer 12 (p. 829)

5.3.2.4 Images

- Image 1 (p. 827) . Question 11.

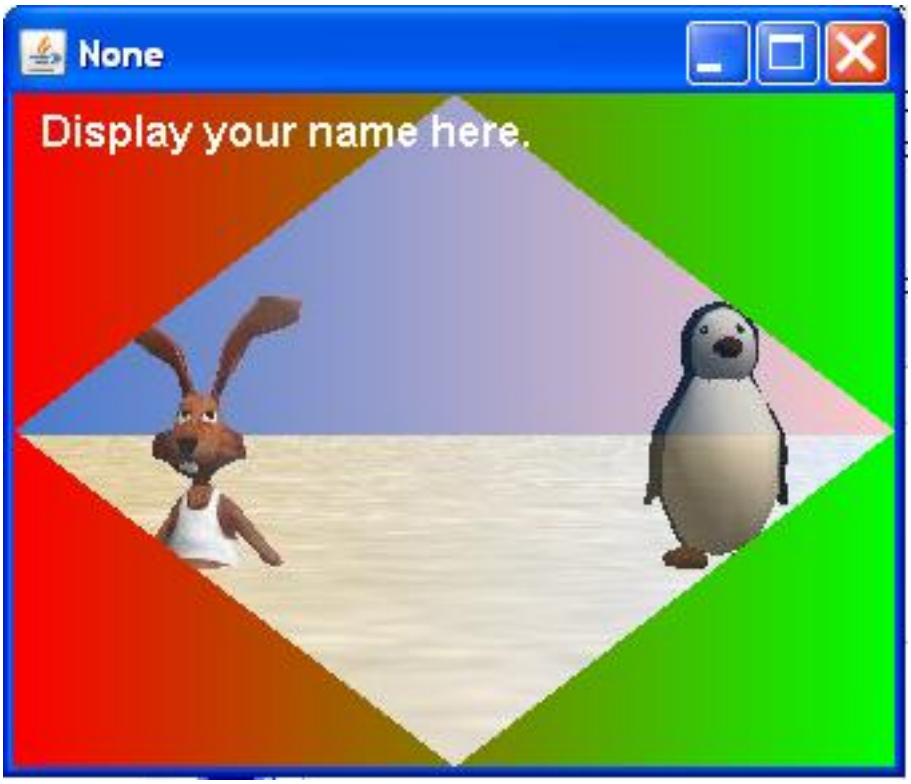
5.3.2.5 Listings

- Listing 1 (p. 825) . Question 7.
- Listing 2 (p. 825) . Question 8.
- Listing 3 (p. 826) . Question 10.
- Listing 4 (p. 826) . Question 11.
- Listing 5 (p. 829) . Answer 11.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.2.6 Answers

5.3.2.6.1 Answer 12

False. The program does require the use of Ericson's multimedia library but it does not require a jpg image file.

Back to Question 12 (p. 827)

5.3.2.6.2 Answer 11

False. The correct code is shown in Listing 5 (p. 829) .

Listing 5, Answer 11.

```
    import java.awt.Color;

class GraphicHelloWorld{
  public static void main(String[] args){
    Picture pix = new Picture(200,100);
    pix.setAllPixelsToAColor(Color.RED);
    pix.addMessage("Graphic Hello World",15,50);
    pix.show();
  }// end main
}//end class
```

5.188

Back to Question 11 (p. 826)

5.3.2.6.3 Answer 10

True

Back to Question 10 (p. 826)

5.3.2.6.4 Answer 9

True.

Back to Question 9 (p. 825)

5.3.2.6.5 Answer 8

True.

Back to Question 8 (p. 825)

5.3.2.6.6 Answer 7

True.

Back to Question 7 (p. 825)

5.3.2.6.7 Answer 6

False. You will need to put the path to the **bookClasses** folder on your **classpath** in order to incorporate classes from the library into your programs.

Back to Question 6 (p. 825)

5.3.2.6.8 Answer 5

True

Back to Question 5 (p. 824)

5.3.2.6.9 Answer 4

False. Once you have downloaded the zip file containing the library, you will need to extract the folder named **bookClasses** from the zip file and store it somewhere on your computer's disk.

Back to Question 4 (p. 824)

5.3.2.6.10 Answer 3

False. The Guzdial-Ericson library was originally published by Guzdial and Ericson in conjunction with their book Introduction to Computing and Programming with Java: A Multimedia Approach [53] . While the book isn't free, the library is freely available and is published under a Creative Commons Attribution 3.0 United States License [54] .

Back to Question 3 (p. 824)

5.3.2.6.11 Answer 2

True.

Back to Question 2 (p. 824)

5.3.2.6.12 Answer 1

False. The multimedia class library was developed and made available by Mark Guzdial [55] and Barbara Ericson [56] of the Georgia Institute of Technology.

Back to Question 1 (p. 824)

5.3.2.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3000r Review: Java3000: The Guzdial-Ericson Multimedia Class Library
- File: Java3000r.htm
- Published: 02/09/13
- Revised: 02/12/13

[53] http://www.pearsonhighered.com/educator/academic/product/0,3110,0131496980,00.html
[54] http://creativecommons.org/licenses/by/3.0/us/
[55] http://www.cc.gatech.edu/~guzdial/
[56] http://coweb.cc.gatech.edu/ice-gt/8

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.3 Java3000s Slides[57]

5.3.3.1 Table of Contents

5.3.3.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3000: The Guzdial-Ericson Multimedia Class Library [58] .

Click here [59] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.3.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3000s Slides: The Guzdial-Ericson Multimedia Class Library
- File: Java3000s.htm
- Published: 01/05/13

[57] This content is available online at <http://cnx.org/content/m45620/1.3/>.
[58] http://cnx.org/content/m44148
[59] http://cnx.org/content/m45620/latest/a0-Index.htm

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.4 Java3002: Creating and Manipulating Turtles and Pictures in a World Object[60]

5.3.4.1 Table of Contents

5.3.4.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [61] .

5.3.4.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

[60] This content is available online at <http://cnx.org/content/m44149/1.7/>.
[61] http://cnx.org/content/m44148/latest/

5.3.4.2.1.1 Images

5.3.4.2.1.2 Listings

5.3.4.3 Preview

In this module, I will explain a program that uses Java and Ericson's media library to *(see Image 1 (p. 834))* :

- Add a picture and two turtles to a world.
- Manipulate the turtles, their colors, and their pens.

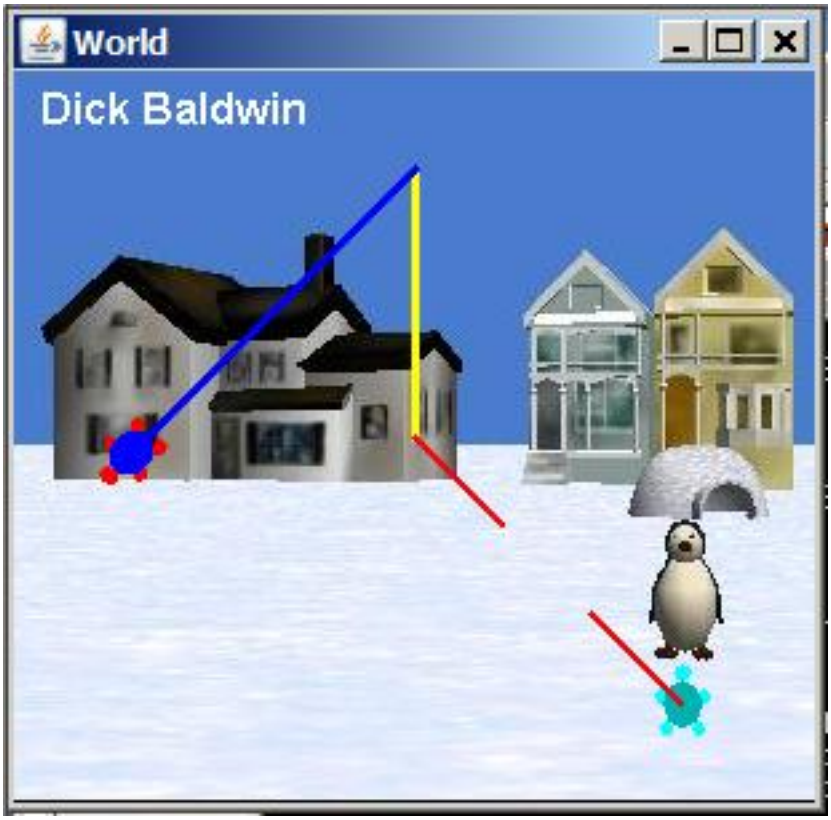
Stated in more detail, the program will:

- Create a **Picture** object from an image file and replace the default white **Picture** in a **World** object with the new **Picture** object.
- Place two **Turtle** objects in a **World** object.
- Apply a series of operations to manipulate the two turtle objects so as to produce a specific graphic output.
- Provide accessor methods to get references to two **Turtle** objects and the **World** object.
- Use the references to get information from the **World** and **Turtle** objects and display that information on the command-line screen.
- Display text on a **Picture** in a **World** object.

Program output

Image 1 (p. 834) shows the graphic screen output produced by this program.

Image 1: Graphic screen output.



5.189

Image 2 (p. 835) shows the text output produced by the program on the command line screen.

Image 2: Command-line output.

```
    javac 1.6.0_14
java version "1.6.0_14"
Java(TM) SE Runtime Environment (build 1.6.0_14-b08)
Java HotSpot(TM) Client VM (build 14.0-b16, mixed mode,
 sharing)
Dick Baldwin.
A 300 by 274 world with 2 turtles in it.
joe turtle at 44, 143 heading -135.0.
sue turtle at 250, 237 heading 0.0.
```

5.190

Output produced by the system

Image 2 (p. 835) not only shows the output produced by the program. It also shows information produced by the Java compiler and the Java virtual machine as a result of executing the following commands at runtime:

```
    javac -version
java -version
```

5.3.4.4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 9 (p. 852) near the end of the module.

I will begin with the driver class named **Prob01** , which is shown in its entirety in Listing 1 (p. 836) .

Listing 1: The driver class.

```
    import java.awt.Color;

public class Prob01{//Driver class
  public static void main(String[] args){
    //Instantiate an object and call its method named run.
    Prob01Runner obj = new Prob01Runner();
    obj.run();

    //Get information from the object and display it on
    // the command-line screen.
    System.out.println(obj.getMars());
    System.out.println(obj.getJoe());
    System.out.println(obj.getSue());
  }//end main
}//end class Prob01
```

5.191

The import directive

Note the *import directive* at the very beginning of Listing 1 (p. 836) . This is a directive to the compiler and the virtual machine notifying them that the class named **Color** can be found in the package named **java.awt** .

What is a package?

Boiled down to its simplest description, a package is nothing more than the specification of a particular folder on the disk relative to a standard root folder. *(Think of it as a disk-path specification with the periods representing \ characters on a Windows machine and representing / characters on a Unix machine.)*

The public class named Prob01

Every Java application *(not true for Java applets)* must include the definition of a class that contains the definition of a method named **main** with a method signature having the syntax shown in Listing 1 (p. 836) .

The name of the class/application

The name of the class containing the **main** method is also the name of the application insofar as being able to compile and execute the application is concerned. In this case, the name of the class, and hence the name of the application is **Prob01** .

Source code file name

The name of the source code file containing this class must match the name of the class. In this case, the source code file must be named **Prob01.java** . *(Note the .java extension.)*

Compiling the application

In its simplest form, this application can be compiled by executing the following command at the command prompt:

```
javac Prob01.java
```

Note however that it is often necessary to specify the path to various library files on the command line when compiling the application. In that case, the simplest form given above is not sufficient.

Compiler output file names

When the application is successfully compiled, it will produce one or more output files with an extension of *.class* . In this case, one of those files will be named **Prob01.class** .

Execution of the application

The execution of C and C++ programs begins and ends in the **main** function. The execution of Java applications begin and end in the method named **main** .

Once again, in its simplest form, this application can be executed by entering the following command at the command prompt:

```
java Prob01
```

Again, it is often necessary to specify the path to various library files on the command line when executing the application. In that case, the simplest form is not sufficient.

Compilation and execution on my machine

This application can be compiled and executed on my machine by entering the two commands shown in Image 3 (p. 837) at the command prompt. *(Note that artificial line breaks were inserted into* *Image 3* (p. 837) *to force the long commands to fit this narrow format.)*

Image 3: Commands to compile and execute the application.

```
    javac -cp .;M:\Baldwin\AA-School\Itse2321IntroOOP\
MediaCompBookMaterial\bookClasses Prob01.java

java -cp .;M:\Baldwin\AA-School\Itse2321IntroOOP\
MediaCompBookMaterial\bookClasses Prob01
```

5.192

The compiler and the virtual machine

The **javac** portion of the first command causes the Java compiler to run.

The **java** portion of the second command causes the Java virtual machine to run.

The input files

The **Prob01.java** and **Prob01** at the ends of the two commands specify the files being operated on by the compiler and the virtual machine respectively.

A classpath

In both cases, the **-cp** indicates that a classpath follows.

A classpath consists of one or more path specifications separated by semicolon characters.

The purpose of the classpath is to tell the compiler and the virtual machine where to look for previously compiled class files that the application needs in order to successfully compile and execute.

The current folder

The period ahead of the semicolon says to search the current folder first.

The path to the Ericson library

The material following the semicolon is the absolute path to the folder containing the class files that make up Ericson's library on my machine. The location of that folder will probably be different on your machine.

The main method

Now that we have the preliminaries out of the way, let's go back and examine the body of the main method in Listing 1 (p. 836) .

The first statement in the body of the main method in Listing 1 (p. 836) instantiates a new object of the class named **Prob01Runner** .

The statement saves a reference to that object in a reference variable named **obj** . Note that the type of the variable is the same as the name of the class in this case. In general, the type of the variable must be:

- The name of the class, or
- The name of a superclass of the class, or
- The name of an interface implemented by the class.

Accessing the object

You must save a reference to an object in order to gain access to the object later. In this case, the reference is stored in the variable named **obj** .

Call the method named run

The second statement in the body of the **main** method in Listing 1 (p. 836) uses that reference to call the method named **run** encapsulated in the object. As you will see later, most of the work in this application is performed in the method named **run** .

Get and display information about the object

When the **run** method returns control to the **main** method, the last three statements in the body of the **main** method in Listing 1 (p. 836) use the object's reference to call the following three methods belonging to the object:

- getMars
- getJoe
- getSue

Accessor methods

These three methods are of a type that is commonly referred to as *accessor methods* . They access and return values encapsulated inside an object. In most cases, *(using good programming practice)* they return copies of the values. This protects the encapsulated values from being corrupted by code outside the object.

The method named println

In each case in Listing 1 (p. 836) , the value returned by the method is passed to a method named **println** . This is a method belonging to a standard system object that represents the standard output device *(usually the command-line screen)* . The purpose of the **println** method is to display material on the command-line screen.

System.out.println...

Without going into detail about how this works, you should simply memorize the syntax of the last three statements in the body of the **main** method in Listing 1 (p. 836) . I explain the concepts involved in great detail on my website. Go to Google and search for the following keywords:

```
baldwin java "class variable named out"
```

This code *(System.out.println...)* provides the mechanism by which you can display material on the command line screen in a running Java application. The last three statements in the main method in Listing 1 (p. 836) produced the last three lines of text in Image 2 (p. 835) .

(Note that only the last four lines of text in Image 2 (p. 835) *were produced by the program. Everything above that was produced by the system during the compilation and initial execution of the application.)*

What do you know so far?

So far, you know the following to be true:

- The program instantiates an object of the class named **Prob01Runner** .
- The program causes a method named **run** belonging to that object to be executed.
- When the **run** method returns, the program calls three accessor methods in succession, causing the values returned by those methods to be displayed on the command-line screen.

- The fourth line of text from the bottom in Image 2 (p. 835) *(Dick Baldwin)* was produced before the last three lines of text discussed above. Therefore, that line of text must have been produced before control reached the call to the **getMars** method in Listing 1 (p. 836) .

The public modifier

Java uses four access modifiers to specify the accessibility of various classes and members in a Java application:

- public
- private
- protected
- package-private

Rather than trying to explain all four at this time, I will explain **public** here and explain the other three when we encounter them in code.

The **public** modifier is the easiest of the four to explain. As the name implies, it is analogous to a public telephone. Any code that can find a class or class member with a public modifier can access and use it. In this case, any code that can find the class definition for the class named **Prob01** can instantiate an object of that class.

The class named Prob01Runner

There's not a lot more that we can say about the driver class named **Prob01** , so it's time to analyze the class named **Prob01Runner** . We need to figure out what it is about that class that causes the program output to match the material shown in Image 1 (p. 834) and Image 2 (p. 835) .

Beginning of the class named Prob01Runner

The definition of the class named **Prob01Runner** begins in Listing 2 (p. 839) .

Listing 2: Beginning of the class named Prob01Runner.

```
 class Prob01Runner{
//Instantiate the World and Turtle objects.
private World mars = new World(300,274);
private Turtle joe = new Turtle(mars);
private Turtle sue = new Turtle(mars);
```

5.193

No access modifier

Note that this class definition does not have an access modifier. This puts it in **package-private** access category. A class with package-private access can be accessed by code that is stored in the same package and cannot be accessed by code stored in other packages.

Three private variables

The last three statements in Listing 2 (p. 839) declare three private variables. Because these variables are declared private, they can only be accessed by code contained in methods defined inside the class.

(They are also accessible by code contained in methods defined in classes defined inside the class, but that is beyond the scope of this module.)

Three private instance variables

These variables are also *instance variables* as opposed to *class variables* . *(We will discuss class variables in a future module.)*

Because they are instance variables, they belong to an object instantiated from the class. *(An object is an instance of a class.)* Even if the variables were public, they could only be accessed by first gaining access to the object to which they belong.

Multiple instances of the class

If you instantiate multiple objects of this same class *(create multiple instances which you often do)* , each object will encapsulate the same set of the three private instance variables shown in Listing 2 (p. 839) . Instance variables have the same name but may have different values in the different objects.

Three private instance reference variables

The three variables declared in Listing 2 (p. 839) are also *reference variables (as opposed to primitive variables)* . This means that they are capable of storing references to objects as opposed to simply being able to store primitive values of the following eight types:

- char
- byte
- short
- int
- long
- float
- double
- boolean

Primitive variables can only store primitive values of the types in the above list.

Classes, classes, and more classes

A Java application consists almost exclusively of objects. Objects are instances of classes. Therefore, class definitions must exist before objects can exist.

The true power of Java

The Java programming language is small and compact. The true power of Java lies in its libraries of predefined classes.

The Java standard edition development kit and runtime engine available from Sun contains a library consisting of thousands of predefined classes. Other class libraries containing thousands of classes are available from sun in the enterprise edition and the micro edition.

Non-standard class libraries

In some cases, you or your company may create your own class libraries and/or obtain class libraries from other sources such as the Ericson class library that we are using in this module.

Custom class definitions

In almost all cases, you will need to define a few new classes for new applications that you write. We will define two new classes for this application. The remainder of the classes that we use will come either from Sun's standard library or Ericson's library.

Objects of the World class and the Turtle class

Ericson's class library contains a class named **World** and another class named **Turtle** . The code in Listing 2 (p. 839) instantiates one object of the **World** class and populates that world with two objects of the **Turtle** class.

Every class has a constructor

Every class definition has one or more method-like members called constructors. *(If you don't define a constructor when you define a class, a default constructor will be automatically defined for your class.)*

The name of the constructor must always be the same as the name of the class. Like a method, a constructor may or may not take arguments. If there are two or more *(overloaded)* constructors, they must have different argument lists.

Instantiating an object of a class

To instantiate an object of a class, you apply the **new** operator *(see Listing 2* (p. 839) *)* to the class' constructor, passing parameters that satisfy the required arguments for the constructor.

Return a reference to the object

Once the object has been instantiated, the constructor returns a reference to the new object.

A new World object

For example, the first statement in Listing 2 (p. 839) applies the **new** operator to Ericson's **World** class constructor passing two integer values as parameters. This causes a new **World** object to be instantiated.

A reference is returned

A reference to the new **World** object is returned and stored in the reference variable named **mars** .

Once the reference is stored in the reference variable, it can be used to access the **World** object later.

Constructors for the World class

Image 4 (p. 841) s shows the constructors that are available for Ericson's **World** class. *(See javadocs for the Ericson library.)*

Image 4: Constructors for the World class.

Constructor Summary
World() Constructor that takes no arguments
World(boolean visibleFlag) Constructor that takes a boolean to say if this world should be visible or not
World(int w, int h) Constructor that takes a width and height for this world

5.194

A new World object

The third constructor in Image 4 (p. 841) was used to construct a **World** object in Listing 2 (p. 839) with a width of 300 pixels and a height of 274 pixels. As explained earlier, this object's reference was saved in the variable named **mars** .

Two new Turtle objects

The last two statements in Listing 2 (p. 839) instantiate two objects of the **Turtle** class and use them to populate the **World** object whose reference is stored in the variable named **mars** .

More complicated than before

This is a little more complicated than the instantiation of the **World** object. Ericson's javadocs indicate that the **Turtle** class provides the four constructors shown in Image 5 (p. 842) .

Image 5: Constructors for the Turtle class.

Constructor Summary
Turtle(int x, int y, ModelDisplay modelDisplayer) Constructor that takes the x and y and a model display to draw it on
Turtle(int x, int y, Picture picture) Constructor that takes the x and y and a picture to draw on
Turtle(ModelDisplay modelDisplay) Constructor that takes the model display
Turtle(Picture p) Constructor that takes a picture to draw on

5.195

A World object as a parameter

If you dig deep enough, and if you study Ericson's textbook, you can determine that the third constructor in Image 5 (p. 842) will accept a reference to a **World** object as a parameter. This is the constructor that was used in the last two statements in Listing 2 (p. 839) .

NOTE: **ModelDisplay interface** The World class implements the ModelDisplay interface. Therefore, an object of the World class can be treated as it is type ModelDisplay. I explain the relationship between classes and interfaces in detail on my website.

Displayed in the center of the world

When the two **Turtle** objects instantiated in Listing 2 (p. 839) come into existence, they will be displayed in the center of the **World** object referred to by the contents of the variable named **mars** . However, that happens so quickly that you probably won't see it when you run this program.

Eliminating the run method call

If you were to eliminate the call to the **run** method in Listing 1 (p. 836) , you would see a world with a white background and a single turtle positioned in the center of the world facing north. There would actually be two turtles there, but they would be in exactly the same location so only the one closest to you would be visible.

The constructor for the Prob01Runner class

That's probably enough discussion of the three statements in Listing 2 (p. 839) . The constructor for the class named **Prob01Runner** is shown in its entirety in Listing 3 (p. 843) .

Listing 3: The constructor for the Prob01Runner class.

```
    public Prob01Runner(){//constructor
  System.out.println("Dick Baldwin.");
}//end constructor
```

5.196

The purpose of constructors

The primary purpose for which constructors exist is to assist in the initialization of the variables belonging to the object being constructed. However, it is possible to directly initialize the variables as shown in Listing 2 (p. 839) .

Initialization of variables

When an object comes into existence, the variables belonging to that object will have been initialized by any direct initializers like those shown in Listing 2 (p. 839) as well any initialization produced by code written into the constructor.

Default initialization

If a variable *(exclusive of local variables inside of methods)* is not initialized in one of those two ways, it will receive a default initialization value. The default values are:

- 0 or 0.0 for numeric variables
- false for boolean variables
- null for reference variables

Non-initialization code in constructors

Although it is usually not good programming practice to do so, there is no technical reason that you can't write code into the constructor that has nothing to do with variable initialization. Such code will be executed when the object is instantiated.

An object counter

For example, you might need to keep track of the number of objects that are instantiated from a particular class, such as the total number of asteroid objects in a game program for example You could write the code to do the counting in the constructor.

Display my name

The code in the constructor in Listing 3 (p. 843) simply causes my name to be displayed on the command-line screen when the object is instantiated. That is how my name appears ahead of the other lines of output text in Image 2 (p. 835) . My name is displayed when the object is instantiated. The remaining three lines of text in Image 2 (p. 835) are displayed later by manipulating the object.

Three accessor methods

Listing 4 (p. 844) defines three accessor methods that are used to access and return copies of the contents of the private instance variables named **joe** , **sue** , and **mars** .

Listing 4: Three accessor methods.

```
    public Turtle getJoe(){return joe;}
public Turtle getSue(){return sue;}
public World getMars(){return mars;}
```

5.197

Good OOP practice

Good object-oriented programming practice says that most of the instance variables encapsulated in an object should be declared private. If there is a need to make the contents of those variables available outside the object, that should be accomplished by defining public accessor methods. *(Accessor methods are often referred to as getter methods because the name of the accessor method often includes the word "get".)*

Setter methods

If there is a need for code outside the object to store information in the object's private instance variables, this should be accomplished by writing public *setter* methods. Code in the setter methods can filter incoming data to make certain that the state of the object doesn't become corrupt as a result of outside influences.

Pass and return by value

Everything in Java is passed and returned *by value* , not by reference.

Each of the accessor methods shown in Listing 4 (p. 844) returns a copy of the reference belonging to either a **Turtle** object or a **World** object.

Pass to the println method

As you saw earlier, each of the three references is passed to the println method in Listing 1 (p. 836) causing information about the objects to be displayed on the command-line screen.

The toString method

Although it isn't obvious in Listing 1 (p. 836) , the code in the **println** method calls a method named **toString** on the incoming object reference and displays the string value returned by that method. I discuss this in detail on my website. Go to Google and search for the following:

```
baldwin java "the toString method"
```

An overridden method

The **toString** method is overridden *(not overloaded)* in the **World** and **Turtle** classes so as to return a string value describing the object.

The Ericson javadocs

Normally, the javadocs would tell you what information is contained in that string, but that is not the case in Ericson's javadocs. You would have to get into her source code, *(which is readily available),* to get that information. However, you can see the information that is contained in the string values for the two different types of objects in the last three lines of text in Image 2 (p. 835) .

The beginning of the run method

This is where thing start to get really interesting. Listing 5 (p. 845) shows the beginning of the public method named **run** .

Listing 5: The beginning of the run method.

```
  public void run(){
//Replace the default all-white picture with another
// picture.
mars.setPicture(new Picture("Prob01.jpg"));
```

5.198

Recall that the code in the **main** method in Listing 1 (p. 836) calls the **run** method on the object immediately after it is instantiated.

A turtle on a white background

I told you earlier that if you were to eliminate the call to the **run** method, you would see a turtle at the center of the world with a white background.

The background is a Picture object

The background of a **World** object consists of an object of Ericson's **Picture** class. *(A **Picture** object is encapsulated in the **World** object.)*

By default, the **Picture** object encapsulated in a **World** object is all white and is exactly the right size and shape to completely fill the area inside the world's border *(see Image 1* (p. 834)).

Can be replaced

As you will see shortly, we can replace the default **Picture** object with a new **Picture** object of our own choosing.

What if it doesn't fit?

If the new **Picture** object isn't large enough to completely fill the area inside the borders of the **World** object, it will be placed in the upper-left corner of the **World** object and the remainder of the **World** object will be a light gray color.

If the **Picture** object is too large, an upper-left rectangular portion of the **Picture** object, sufficient to fill the **World** object, will be displayed. The remainder of the **Picture** object will not be visible even if you manually resize the **World** object to make it larger.

Constructors for the Picture class

Image 6 (p. 846) shows the javadocs for the constructors for Ericson's **Picture** class.

Image 6: Constructors for the Picture class.

Constructor Summary
Picture() Constructor that takes no arguments
Picture(BufferedImage image) Constructor that takes a buffered image
Picture(int width, int height) Constructor that takes the width and height
Picture(Picture copyPicture) Constructor that takes a picture and creates a copy of that picture
Picture(String fileName) Constructor that takes a file name and creates the picture

5.199

Replace the default picture object

The right-hand portion of the last statement in Listing 5 (p. 845) uses the last constructor in Image 6 (p. 846) to instantiate a new **Picture** object that encapsulates the image contained in the image file named **Prob01.jpg** .

(Click here [62] *to download a copy of the file named Prob01.jpg.)*

What about the size of the Picture object?

I was careful to use an image that was a little wider than and exactly as tall as the dimensions of my **World** object *(300 x 274)* . Therefore, the image completely filled the world as shown in Image 1 (p. 834) .

Pass the reference to a setter method

The reference belonging to the new **Picture** object was passed to the **setPicture** method of the **World** object *(a setter method)* . This caused the new picture containing the penguin to replace the default all-white picture that forms the background for the World object. *(See Image 1 (p. 834) .)*

A subclass of the SimplePicture class

Ericson's **Picture** class is a subclass of *(extends)* the class named **SimplePicture** . Therefore, an object of the **Picture** class encapsulates all of the methods defined in the **Picture** class in addition to all of the methods defined in the **SimplePicture** class.

A subclass of the Object class

Further, the **SimplePicture** class is a subclass of *(extends)* the **Object** class. Therefore, an object of the **Picture** class also encapsulates all of the methods defined in the **Object** class.

[62] http://cnx.org/content/m44149/latest/Prob01.jpg

The AddMessage method

One of the methods defined in the **SimplePicture** class and inherited into the **Picture** class is named **AddMessage** .

The **addMessage** method requires three parameters:

- a string and
- two coordinate values of type **int** .

The method will draw the string as text characters onto the image at the location specified by the two coordinate values.

(The origin of the coordinate system is the upper-left corner of the image with positive horizontal values going to the right and positive vertical values going down.)

Add text to the image

The code in Listing 6 (p. 847) uses two levels of indirection to add my name as a message to the picture that forms the background of the world as shown in Image 1 (p. 834) .

Listing 6: Add text to the image.

```
mars.getPicture().addMessage(
                  "Dick Baldwin",10,20);
```

5.200

Get and access the World object

To begin with, Listing 6 (p. 847) goes to the variable named **mars** to get a reference to the **World** object. This reference is used to access the **World** object.

Access the Picture object via a getter method

Then the code in Listing 6 (p. 847) calls the getter method named **getPicture** to get access to the **Picture** object encapsulated in the **World** object.

Call the addMessage method

Having gained access to the **Picture** object, Listing 6 (p. 847) calls the **addMessage** method on that object passing my name as a **String** object along with a pair of coordinate values that specify a location near the upper-left corner of the image. The result is that my name appears in the world as shown in Image 1 (p. 834) .

Methods encapsulated in the Turtle object

The **Turtle** class extends the **SimpleTurtle** class, which in turn extends the **Object** class. Therefore, an object of the **Turtle** class encapsulates all of the methods defined in all three classes.

Manipulate the turtle referred to by the variable named joe

A **Turtle** object encapsulates many methods that can be used to manipulate the turtle in a variety of different ways. This is illustrated by the series of statements in Listing 7 (p. 848) .

Listing 7: Manipulate the turtle named joe.

```
    joe.setName("joe");

joe.setBodyColor(Color.RED);
joe.setShellColor(Color.BLUE);

joe.setPenColor(Color.YELLOW);
joe.setPenWidth(3);

joe.forward();

joe.turn(-135);
joe.setPenColor(Color.BLUE);
joe.forward(150);
```

5.201

Initial (default) state of a Turtle object

When a new **Turtle** object is instantiated and added to a **World** object *(using the constructor shown in Listing 2 (p. 839))* , it doesn't have a name property. *(Actually, its name is probably null.)*

The turtle initially appears in the center of the world, facing north with a default color.

Every **Turtle** object has a pen attached to its belly that can draw a line with a default width of one pixel in a default color when the turtle moves.

The pen can be raised so that it won't draw a line or lowered so that it will draw a line. Initially it is down and will draw a line.

Set the name property to "joe"

Listing 7 (p. 848) begins by setting the name property of one of the turtles to the string value "joe."

(Note that this is completely independent of the fact that a reference to this turtle is stored in a variable named ***joe*** *. The name property could have been set to "Tom", "Dick", "Harry", or any other string value. It is the value of the name property and not the name of the variable that determines the text output shown in Image 2 (p. 835) .)*

Set the turtle's body and shell color

Listing 7 (p. 848) continues by calling two *setter* methods on the turtle object to set the body color *(head and feet)* to red and the color of the shell to blue. You can see the effect of this in Image 1 (p. 834) .

Set the pen color and width

Then Listing 7 (p. 848) calls two *setter* methods that set the turtle's pen color to yellow and the pen width to three pixels. You can also see the result of this in Image 1 (p. 834) .

Make the turtle move forward

After that, Listing 7 (p. 848) calls the **forward** method *(with no parameters)* to cause the turtle to move forward by a default distance of 100 pixels.

Recall that the turtle initially faces north. In this case, the forward method causes the turtle to move from the center of the world to a location that is 100 pixels due north of the center of the world, drawing a wide yellow line along the way.

Turn counter clockwise

Then Listing 7 (p. 848) calls the **turn** method causing the turtle to rotate its body by 135 degrees counter-clockwise. *(A positive parameter causes a clockwise rotation and a negative parameter causes a*

counter clockwise rotation.)

Change the pen color and move forward again

Finally Listing 7 (p. 848) calls methods to change the pen color to blue and to cause the turtle to move forward by 150 pixels.

The final location

After making the turn, the turtle is facing southwest. Therefore, the forward movement causes a diagonal blue line to be drawn from the position at the top of the yellow line down toward the southwest. As you can see in Image 1 (p. 834) , the turtle comes to rest at the end of that line.

A few words about color

I have published extensively on the concept of color in Java on my website. The best way to find that information is probably to go to Google and search for the keywords:

```
baldwin java color
```

Google is also probably your best bet for finding information on other topics that I have published on my website. For example, if you go to Google Images and search for the following keywords, you will find a lot of the work that I have published using Ericson's media library.

```
richard baldwin java ericson
```

Manipulate the turtle named sue

Listing 8 (p. 849) calls several methods on the object whose reference is stored in the variable named **sue** .

Listing 8: Manipulate the turtle named sue.

```
        sue.setName("sue");

    sue.setPenWidth(2);
    sue.setPenColor(Color.RED);

    sue.moveTo(183,170);
    sue.setPenDown(false);

    sue.moveTo(216,203);
    sue.setPenDown(true);

    sue.moveTo(250,237);
  }//end run method
}//end class Prob01Runner
```

5.202

The end result

These method calls result in the turtle facing north in the lower right corner of the window, having drawn the broken red line shown in Image 1 (p. 834) in getting there.

The moveTo method

Listing 8 (p. 849) calls the **moveTo** method to cause the turtle to move to a new location on the basis of coordinate values instead of on the basis of a distance value.

Pen control

Listing 8 (p. 849) also calls the **setPenDown** method twice passing false and then true as the parameter to first raise and then lower the pen. This produced the gap in the red line shown in Image 1 (p. 834) .

The end of the program

Listing 8 (p. 849) also signals the end of the method named **run** and the end of the class named **Prob01Runner** . As such, Listing 8 (p. 849) signals the end of the program.

5.3.4.5 Run the program

II encourage you to copy the code from Listing 9 (p. 852) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

5.3.4.6 Summary

II explained a program that uses Java and Ericson's media library to:

- Add a picture and two turtles to a world.
- Manipulate the turtles, their color, and their pens.

Stated in more detail, the program:

- Creates a **Picture** object and replaces the default white **Picture** in a **World** object with the new **Picture** object.
- Places two **Turtle** objects in a **World** object.
- Applies a series of operations to manipulate the two turtle objects so as to produce a specific graphic output.
- Provides accessor methods to get references to two **Turtle** objects and the **World** object.
- Gets information from the **World** and **Turtle** objects and displays the information on the command-line screen.
- Displays text on a **Picture** in a **World** object.

5.3.4.7 What's next?

In the next module, I will teach you how to invert images and how to display images using Ericson's **PictureExplorer** object.

5.3.4.8 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 01 [63]
  - Part01 [64]
  - Part02 [65]
  - Part03 [66]
  - Part04 [67]
  - Part05 [68]

[63] http://www.youtube.com/playlist?list=PL26BF7154F10D3854
[64] http://www.youtube.com/watch?v=7KjSLqTgMec
[65] http://www.youtube.com/watch?v=Jnra7RfPKOI
[66] http://www.youtube.com/watch?v=mJDGp1HPCuE
[67] http://www.youtube.com/watch?v=mYrGKI16j_4
[68] http://www.youtube.com/watch?v=UUTlMh3J5Ow

- Part06 [69]
- Part07 [70]
- Part08 [71]
- Part09 [72]
- Part10 [73]

5.3.4.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Creating and Manipulating Turtles and Pictures in a World Object
- File: Java3002.htm
- Published: 07/26/12
- Revised: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have misappropriated copies of my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I receive no compensation for those sales and don't know who does receive compensation. If you purchase such a book, please be aware that it is a bootleg copy of a module that is freely available on cnx.org.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.4.10 Complete program listings

A complete listing of the program discussed in this module is shown in Listing 9 below.

[69] http://www.youtube.com/watch?v=hmu-l1a7VyE
[70] http://www.youtube.com/watch?v=tRpS7c-aPd0
[71] http://www.youtube.com/watch?v=cE_ks6Oq3OY
[72] http://www.youtube.com/watch?v=4f39g8oEWsY
[73] http://www.youtube.com/watch?v=yeay7REkLBY

Listing 9: Source code for Prob01.

```
    /*File Prob01 Copyright 2008 R.G.Baldwin

Command-line output

javac 1.6.0_14
java version "1.6.0_14"
Java(TM) SE Runtime Environment (build 1.6.0_14-b08)
Java HotSpot(TM) Client VM (build 14.0-b16, mixed mode,
 sharing)
Dick Baldwin.
A 300 by 274 world with 2 turtles in it.
joe turtle at 44, 143 heading -135.0.
sue turtle at 250, 237 heading 0.0.
**********************************************************/
import java.awt.Color;

public class Prob01{//Driver class
  public static void main(String[] args){
    Prob01Runner obj = new Prob01Runner();
    obj.run();

    System.out.println(obj.getMars());
    System.out.println(obj.getJoe());
    System.out.println(obj.getSue());
  }//end main
}//end class Prob01
//**********************************************************/

class Prob01Runner{
  //Instantiate the World and Turtle objects.
  private World mars = new World(300,274);
  private Turtle joe = new Turtle(mars);
  private Turtle sue = new Turtle(mars);

  public Prob01Runner(){//constructor
    System.out.println("Dick Baldwin.");
  }//end constructor
  //-------------------------------------------------------//

  //Accessor methods
  public Turtle getJoe(){return joe;}
  public Turtle getSue(){return sue;}
  public World getMars(){return mars;}
  //-------------------------------------------------------//

  //This method is where the action is.
  public void run(){
    //Replace the default all-white picture with another
    // picture.
    mars.setPicture(new Picture("Prob01.jpg"));
    mars.getPicture().addMessage(
                        "Dick Baldwin",10,20);

    //Manipulate the turtle named joe.
    joe.setName("joe");
```

-end-

5.3.5 Java3002r Review[74]

5.3.5.1 Table of Contents

5.3.5.2 Preface

This module contains review questions and answers keyed to the module titled Java3002: Creating and Manipulating Turtles and Pictures in a World Object [75] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.5.3 Questions

5.3.5.3.1 Question 1 .

True or False? The *import directive* at the very beginning of Listing 1 (p. 855) is a directive to the compiler and the virtual machine notifying them that the class named **Color** can be found in the package named **java.awt** .

[74] This content is available online at <http://cnx.org/content/m45762/1.3/>.
[75] http://cnx.org/content/m44149

Listing 1. Question 1.

```
    import java.awt.Color;

public class Prob01{//Driver class
  public static void main(String[] args){
    //Instantiate an object and call its method named run.
    Prob01Runner obj = new Prob01Runner();
    obj.run();

    //Get information from the object and display it on
    // the command-line screen.
    System.out.println(obj.getMars());
    System.out.println(obj.getJoe());
    System.out.println(obj.getSue());
  }//end main
}//end class Prob01
```

5.204

Answer 1 (p. 871)

5.3.5.3.2 Question 2

True or False? The class named **Color** imported in Listing 1 (p. 855) is a member of Ericson's multimedia library.

Answer 2 (p. 871)

5.3.5.3.3 Question 3

True or False? A *package* is the specification of a particular folder on the disk relative to a standard root folder.

Answer 3 (p. 871)

5.3.5.3.4 Question 4

True or False? Every Java application and every Java applet must include the definition of a class that contains the definition of a method named **main** .

Answer 4 (p. 870)

5.3.5.3.5 Question 5

True or False? The name of the class containing the **main** method is also the name of the application insofar as being able to compile and execute the application is concerned.

Answer 5 (p. 870)

5.3.5.3.6 Question 6

True or False? The name of the application shown in Listing 1 (p. 855) is **Proj01** .
Answer 6 (p. 870)

5.3.5.3.7 Question 7

True or False? The name of the source code file containing the class definition shown in Listing 1 (p. 855) must be **Prob01.java** in order for the application to compile and run as an application named **Prob01** .
Answer 7 (p. 870)

5.3.5.3.8 Question 8

True or False? In its simplest form, an application can be compiled by executing the command shown in Listing 2 (p. 856) at the command prompt where **Prob01.java** is the name of the file containing the **main** method.

Listing 2. Question 8.

```
javac Prob01
```

5.205

Answer 8 (p. 870)

5.3.5.3.9 Question 9

True or False? It is often necessary to specify the path to various library files on the command line when compiling an application. In that case, the simplest form given in Listing 3 (p. 870) is not sufficient.
Answer 9 (p. 870)

5.3.5.3.10 Question 10

True or False? When a Java application is successfully compiled, it will produce one or more output files with an extension of *.class* .
Answer 10 (p. 870)

5.3.5.3.11 Question 11

True or False? The execution of a Java application begins and ends in the method named **main** .
Answer 11 (p. 870)

5.3.5.3.12 Question 12

True or False? The two commands shown in Listing 4 (p. 857) can be used to compile and execute a Java application named **Prob01** where:

- The only special class library required is contained in the folder named **bookClasses** .
- The path from the root to the folder named **bookClasses** is represented by —\bookClasses.

Listing 4. Question 12.

```
javac -cp .;---\bookClasses Prob01.java

java -cp .;---\bookClasses Prob01
```

5.206

Answer 12 (p. 869)

5.3.5.3.13 Question 13

True or False? The Java compiler program is named **java.exe** .
Answer 13 (p. 869)

5.3.5.3.14 Question 14

True or False? The **Prob01.java** and **Prob01** at the ends of the two commands in Listing 4 (p. 857) specify the files being operated on by the virtual machine and the compiler respectively.
Answer 14 (p. 869)

5.3.5.3.15 Question 15

True or False? In Listing 4 (p. 857) , the **-cp** indicates that a classpath follows.
Answer 15 (p. 869)

5.3.5.3.16 Question 16

True or False? A classpath consists of one or more path specifications separated by semicolon characters.
Answer 16 (p. 869)

5.3.5.3.17 Question 17

True or False? The purpose of the classpath in Listing 4 (p. 857) is to tell the compiler and the virtual machine where to look for source code files that the application needs in order to successfully compile and execute.
Answer 17 (p. 869)

5.3.5.3.18 Question 18

True or False? The period ahead of the semicolon in Listing 4 (p. 857) says to search the root folder first.
Answer 18 (p. 869)

5.3.5.3.19 Question 19

True or False? The first statement in the body of the **main** method in Listing 1 (p. 855) instantiates a new object of the class named **Prob01Runner** .
Answer 19 (p. 869)

5.3.5.3.20 Question 20

True or False? The first statement in Listing 1 (p. 855) saves a reference to a new object of the class named **Prob01Runner** in a reference variable named **obj** . In general, the type of the variable must be:

- The name of the object, or
- The name of a superclass of the object, or
- The name of an interface implemented by the object.

Answer 20 (p. 869)

5.3.5.3.21 Question 21

True or False? In Java, you must save a reference to a newly instantiated object in order to gain access to that object later in the program.

Answer 21 (p. 868)

5.3.5.3.22 Question 22

True or False? The second statement in the body of the **main** method in Listing 1 (p. 855) uses the reference stored in the variable named **obj** to call the method named **run** encapsulated in the object referred to by the contents of **obj** .

Answer 22 (p. 868)

5.3.5.3.23 Question 23

True or False? The following three methods that are called in Listing 1 (p. 855) are of a type that is commonly referred to as *accessor methods* . They are also sometimes referred to by the slang term *getter methods* .

- getMars
- getJoe
- getSue

Answer 23 (p. 868)

5.3.5.3.24 Question 24

True or False? The method named **println** that is called in Listing 1 (p. 855) is a member of Ericson's multimedia library. The purpose of the **println** method is to display text on an image.

Answer 24 (p. 868)

5.3.5.3.25 Question 25

True or False? Java uses four access modifiers to specify the accessibility of various classes and members in a Java application:

- public
- private
- protected
- package-private

Answer 25 (p. 868)

5.3.5.3.26 Question 26

True or False? Listing 5 (p. 859) shows the beginning of a class named **Prob01Runner** .

Listing 5. Listing 3. Question 26.

```
  class Prob01Runner{
//Instantiate the World and Turtle objects.
private World mars = new World(300,274);
private Turtle joe = new Turtle(mars);
private Turtle sue = new Turtle(mars);
```

5.207

Answer 26 (p. 868)

5.3.5.3.27 Question 27

True or False? The class that begins in Listing 5 (p. 859) does not have an access modifier. This puts it in **package-private** access category. A class with package-private access can be accessed by code that is stored in the same package and cannot be accessed by code stored in other packages.

Answer 27 (p. 868)

5.3.5.3.28 Question 28

True or False? The last three statements in Listing 5 (p. 859) declare three **private** variables. Because these variables are declared private, they can be accessed by any method defined in any class in the same package.

Answer 28 (p. 868)

5.3.5.3.29 Question 29

True or False? The three variables declared in Listing 5 (p. 859) are *instance variables* as opposed to *class variables* .

Answer 29 (p. 868)

5.3.5.3.30 Question 30

True or False? Because the three variables declared in Listing 5 (p. 859) are instance variables, they belong to an object instantiated from the class. Even if the variables were public, they could only be accessed by first gaining access to the object to which they belong.

Answer 30 (p. 867)

5.3.5.3.31 Question 31

True or False? The three variables declared in Listing 5 (p. 859) are *reference variables* . This means that they are capable of storing references to objects and are also capable of storing values of the eight primitive types.

Answer 31 (p. 867)

5.3.5.3.32 Question 32

True or False? Ericson's class library contains a class named **World** and another class named **Turtle**. The code in Listing 5 (p. 859) instantiates one object of the **World** class and populates that world with three objects of the **Turtle** class.

Answer 32 (p. 867)

5.3.5.3.33 Question 33

True or False? Every class definition has one or more method-like members called constructors. *(If you don't define a constructor when you define a class, a default constructor will be automatically defined for your class.)*

Answer 33 (p. 867)

5.3.5.3.34 Question 34

True or False? The name of the constructor must always be the same as the name of the class in which it is defined. Like a method, a constructor may or may not take arguments. If there are two or more *(overloaded)* constructors, they must have different argument lists.

Answer 34 (p. 867)

5.3.5.3.35 Question 35

True or False? To instantiate an object of a class, you apply the **new** operator to the class' constructor, passing parameters that satisfy the required arguments for the constructor.

Answer 35 (p. 867)

5.3.5.3.36 Question 36

True or False? When an object is instantiated, the constructor returns an array containing the values in all of the instance variables.

Answer 36 (p. 867)

5.3.5.3.37 Question 37

True or False? The last two statements in Listing 5 (p. 859) instantiate two objects of the **Turtle** class and use them to populate the **World** object whose reference is stored in the variable named **mars**.

Answer 37 (p. 867)

5.3.5.3.38 Question 38

True or False? If a variable *(exclusive of local variables inside of methods)* is not purposely initialized when the object in instantiated, it will receive a default initialization value. The default values are:

- 0 or 0.0 for numeric variables
- true for boolean variables
- null for reference variables

Answer 38 (p. 867)

5.3.5.3.39 Question 39

True or False? Code written into a class' constructor is executed when an object of the class is instantiated.

Answer 39 (p. 866)

5.3.5.3.40 Question 40

True or False? Good object-oriented programming practice says that most of the instance variables encapsulated in an object should be declared private. If there is a need to make the contents of those variables available outside the object, that should be accomplished by defining protected accessor methods.

Answer 40 (p. 866)

5.3.5.3.41 Question 41

True or False? Everything in Java is passed and returned by reference.

Answer 41 (p. 866)

5.3.5.3.42 Question 42

True or False? Each of the accessor methods shown in Listing 6 (p. 861) returns a copy of the reference pointing to either a **Turtle** object or a **World** object.

Listing 6. Question 42.

```
    public Turtle getJoe(){return joe;}
public Turtle getSue(){return sue;}
public World getMars(){return mars;}
```

5.208

Answer 42 (p. 866)

5.3.5.3.43 Question 43

True or False? Code in the **println** method calls a method named **toString** on each incoming primitive value and displays the string value returned by that method.

Answer 43 (p. 866)

5.3.5.3.44 Question 44

True or False? The **toString** method is overridden *(not overloaded)* in the **World** and **Turtle** classes so as to return a string value describing the object.

Answer 44 (p. 866)

5.3.5.3.45 Question 45

True or False? The code in Listing 7 (p. 862) replaces the default all-white picture in a **World** object with another picture. *(Note, the variable named* ***mars*** *contains a reference to an object of the class* ***World*** *.)*

Listing 7. Question 45.

```
  public void run(){
mars.setPicture(new Picture("Prob01.jpg"));
```

5.209

Answer 45 (p. 866)

5.3.5.3.46 Question 46

True or False? The background of a **World** object consists of an object of Ericson's **Picture** class. *(A **Picture** object is encapsulated in the **World** object.)*
Answer 46 (p. 866)

5.3.5.3.47 Question 47

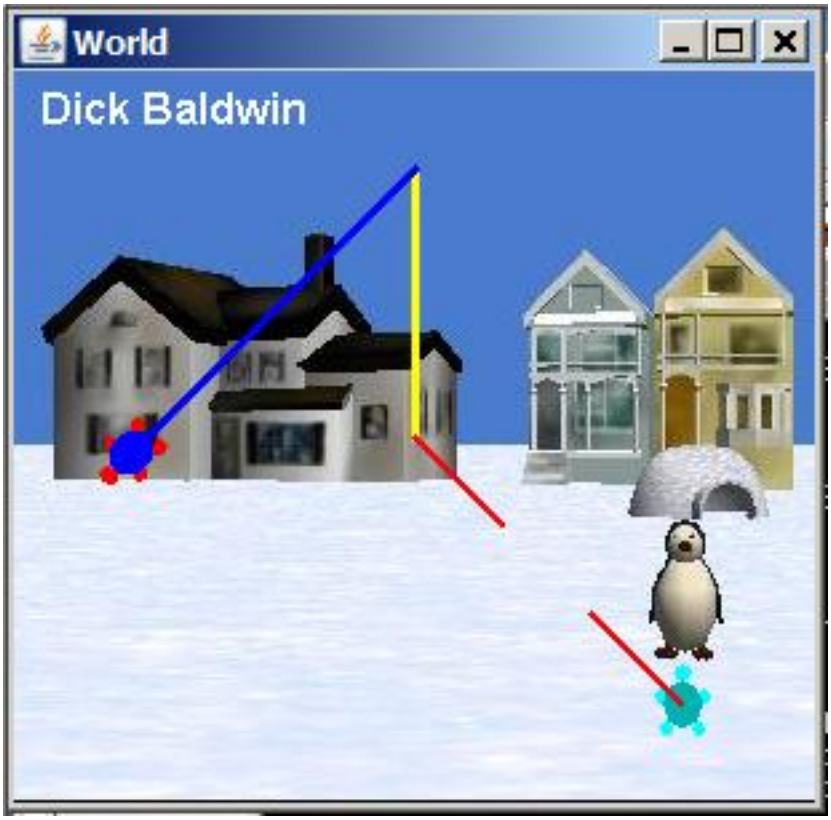
True or False? The code in Listing 8 (p. 862) uses two levels of indirection to add my name to the picture that forms the background of the world shown in Image 1 (p. 863) .

Listing 8. Question 47.

```
mars.getPicture().toString(
                "Dick Baldwin",10,20);
```

5.210

Image 1. Question 47.



5.211

Answer 47 (p. 865)

5.3.5.3.48 Question 48

True or False? A **Turtle** object encapsulates many methods that can be used to manipulate the turtle in a variety of different ways.

Answer 48 (p. 865)

5.3.5.3.49 Question 49

True or False? A call to the **forward** method of a **turtle** object with no parameters causes the turtle to move forward by a default distance of 100 pixels.

Answer 49 (p. 865)

5.3.5.3.50 Question 50

True or False? A call to the **moveTo** method of a **turtle** object with a single parameter value of 150 causes the turtle to move forward by a distance of 150 pixels.

Answer 50 (p. 865)

5.3.5.4 Images

- Image 1 (p. 863) . Question 47.

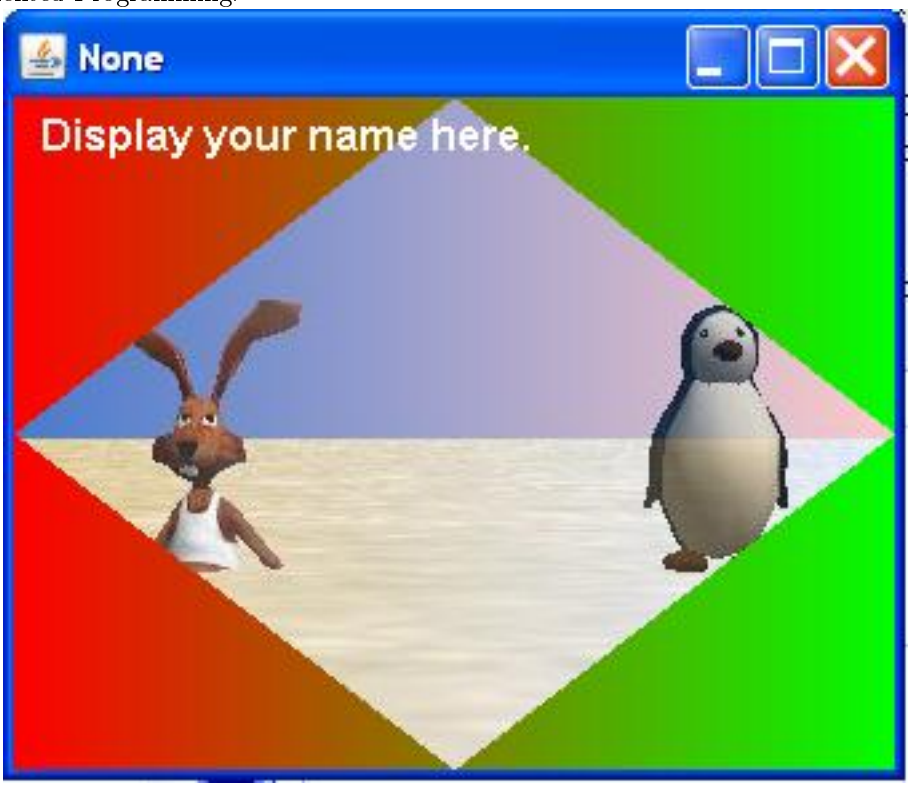
5.3.5.5 Listings

- Listing 1 (p. 855) . Question 1.
- Listing 2 (p. 856) . Question 8.
- Listing 3 (p. 870) . Answer 8.
- Listing 4 (p. 857) . Question 12.
- Listing 5 (p. 859) . Question 26.
- Listing 6 (p. 861) . Question 42.
- Listing 7 (p. 862) . Question 45.
- Listing 8 (p. 862) . Question 47.
- Listing 9. (p. 866) Answer 47.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.5.6 Answers

5.3.5.6.1 Answer 50

False. The moveTo method of the Turtle class cannot be called with a single parameter. Two parameters are required.

Back to Question 50 (p. 863)

5.3.5.6.2 Answer 49

True.

Back to Question 49 (p. 863)

5.3.5.6.3 Answer 48

True.

Back to Question 48 (p. 863)

5.3.5.6.4 Answer 47

False. The code in Listing 8 (p. 862) won't compile. The **toString** method does not apply to images. The correct code is shown in Listing 9 (p. 866) .

Listing 9. Answer 47.

```
mars.getPicture().addMessage(
                  "Dick Baldwin",10,20);
```

5.212

Back to Question 47 (p. 862)

5.3.5.6.5 Answer 46

True.

Back to Question 46 (p. 862)

5.3.5.6.6 Answer 45

True.

Back to Question 45 (p. 861)

5.3.5.6.7 Answer 44

True.

Back to Question 44 (p. 861)

5.3.5.6.8 Answer 43

False. Code in the **println** method calls a method named **toString** on each incoming **object reference** and displays the string value returned by that method.

Back to Question 43 (p. 861)

5.3.5.6.9 Answer 42

True.

Back to Question 42 (p. 861)

5.3.5.6.10 Answer 41

False. Everything in Java is passed and returned *by value* , not by reference.

Back to Question 41 (p. 861)

5.3.5.6.11 Answer 40

False. If there is a need to make the contents of those variables available outside the object, that should be accomplished by defining **public** accessor methods.

Back to Question 40 (p. 861)

5.3.5.6.12 Answer 39

True.

Back to Question 39 (p. 860)

5.3.5.6.13 Answer 38

False.
The default values are:

- 0 or 0.0 for numeric variables
- **false** for boolean variables
- null for reference variables

Back to Question 38 (p. 860)

5.3.5.6.14 Answer 37

True.
Back to Question 37 (p. 860)

5.3.5.6.15 Answer 36

False. When an object is instantiated, the constructor returns a reference to the new object.
Back to Question 36 (p. 860)

5.3.5.6.16 Answer 35

True.
Back to Question 35 (p. 860)

5.3.5.6.17 Answer 34

True.
Back to Question 34 (p. 860)

5.3.5.6.18 Answer 33

True.
Back to Question 33 (p. 860)

5.3.5.6.19 Answer 32

False. The code in Listing 5 (p. 859) instantiates one object of the **World** class and populates that world with ***two*** objects of the **Turtle** class.
Back to Question 32 (p. 860)

5.3.5.6.20 Answer 31

False. The three variables declared in Listing 5 (p. 859) are *reference variables (as opposed to primitive variables)* . This means that they are capable of storing references to objects as opposed to simply being able to store values of the eight primitive types. It also means that they are incapable of storing values of the eight primitive types.
Back to Question 31 (p. 859)

5.3.5.6.21 Answer 30

True.
Back to Question 30 (p. 859)

5.3.5.6.22 Answer 29

True.

Back to Question 29 (p. 859)

5.3.5.6.23 Answer 28

False. Because these variables are declared **private** , they can only be accessed by code contained in methods defined inside the same class *(and in inner classes of the class, which is beyond the scope of this module)* .

Back to Question 28 (p. 859)

5.3.5.6.24 Answer 27

True.

Back to Question 27 (p. 859)

5.3.5.6.25 Answer 26

True.

Back to Question 26 (p. 859)

5.3.5.6.26 Answer 25

True.

Back to Question 25 (p. 858)

5.3.5.6.27 Answer 24

False. The method named **println** that is called in Listing 1 (p. 855) is a method belonging to a standard system object that represents the standard output device *(usually the command-line screen)* . The purpose of the **println** method is to display material on the command-line screen.

Back to Question 24 (p. 858)

5.3.5.6.28 Answer 23

True.

Back to Question 23 (p. 858)

5.3.5.6.29 Answer 22

True.

Back to Question 22 (p. 858)

5.3.5.6.30 Answer 21

True.

Back to Question 21 (p. 858)

5.3.5.6.31 Answer 20

False.
In general, the type of the variable must be:

- The name of the class, or
- The name of a superclass of the class, or
- The name of an interface implemented by the class.

Back to Question 20 (p. 858)

5.3.5.6.32 Answer 19

True.
Back to Question 19 (p. 857)

5.3.5.6.33 Answer 18

False. The period ahead of the semicolon in Listing 4 (p. 857) says to search the current folder first.
Back to Question 18 (p. 857)

5.3.5.6.34 Answer 17

False. The purpose of the classpath is to tell the compiler and the virtual machine where to look for previously compiled class files that the application needs in order to successfully compile and execute.
Back to Question 17 (p. 857)

5.3.5.6.35 Answer 16

True.
Back to Question 16 (p. 857)

5.3.5.6.36 Answer 15

True
Back to Question 15 (p. 857)

5.3.5.6.37 Answer 14

False. The **Prob01.java** and **Prob01** at the ends of the two commands in Listing 4 (p. 857) specify the files being operated on by the compiler and the virtual machine respectively.
Back to Question 14 (p. 857)

5.3.5.6.38 Answer 13

False. The Java compiler program is named **javac.exe** . The virtual machine is named **java.exe** .
Back to Question 13 (p. 857)

5.3.5.6.39 Answer 12

True.
Back to Question 12 (p. 856)

5.3.5.6.40 Answer 11

True.
Back to Question 11 (p. 856)

5.3.5.6.41 Answer 10

True.
Back to Question 10 (p. 856)

5.3.5.6.42 Answer 9

True.
Back to Question 9 (p. 856)

5.3.5.6.43 Answer 8

False. The required command is shown in Listing 3 (p. 870) .

Listing 3. Answer 8.

```
javac Prob01.java
```

5.213

Back to Question 8 (p. 856)

5.3.5.6.44 Answer 7

True.
Back to Question 7 (p. 856)

5.3.5.6.45 Answer 6

False. The name of the application shown in Listing 1 (p. 855) is **Prob01** .
Back to Question 6 (p. 856)

5.3.5.6.46 Answer 5

True.
Back to Question 5 (p. 855)

5.3.5.6.47 Answer 4

False. Java applets do not require a method named **main** .
Back to Question 4 (p. 855)

5.3.5.6.48 Answer 3

True.
Back to Question 3 (p. 855)

5.3.5.6.49 Answer 2

False. **java.awt.Color** belongs to the Java standard edition class library.
Back to Question 2 (p. 855)

5.3.5.6.50 Answer 1

True.
Back to Question 1 (p. 854)

5.3.5.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3002r Review: Creating and Manipulating Turtles and Pictures in a World Object
- File: Java3002r.htm
- Published: 02/10/13
- Revised: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.6 Java3002s Slides[76]

5.3.6.1 Table of Contents

[76] This content is available online at <http://cnx.org/content/m45621/1.5/>.

5.3.6.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3002: Creating and Manipulating Turtles and Pictures in a World Object [77] .

Click here [78] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.6.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3002s Slides: Creating and Manipulating Turtles and Pictures in a World Object
- File: Java3002s.htm
- Published: 01/05/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.7 Java3004: Image Processing Algorithms, Image Inversion, and PictureExplorer Objects[79]

5.3.7.1 Table of Contents

[77] http://cnx.org/content/m44149
[78] http://cnx.org/content/m45621/latest/a0-Index.htm
[79] This content is available online at <http://cnx.org/content/m44203/1.6/>.

5.3.7.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [80] .

5.3.7.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.7.2.1.1 Images

5.3.7.2.1.2 Listings

5.3.7.3 Preview

The program that I will explain in this module is designed to be used as a test of the student's understanding of programming using Java and Ericson's media library.

The student is provided an image file named **Prob02.jpg** along with a pair of **PictureExplorer** windows containing the raw image and a modified version of the image. *(See Image 1 (p. 875) and Image 2 (p. 876) .)*

Deduce the algorithm

[80] http://cnx.org/content/m44148/latest/

The first part of the test is to determine if the student can examine the raw image shown in the **PictureExplorer** window in *Image 1 (p. 875)* and deduce the algorithm required to produce the output shown in the **PictureExplorer** window in *Image 2 (p. 876)* .

Implement the algorithm

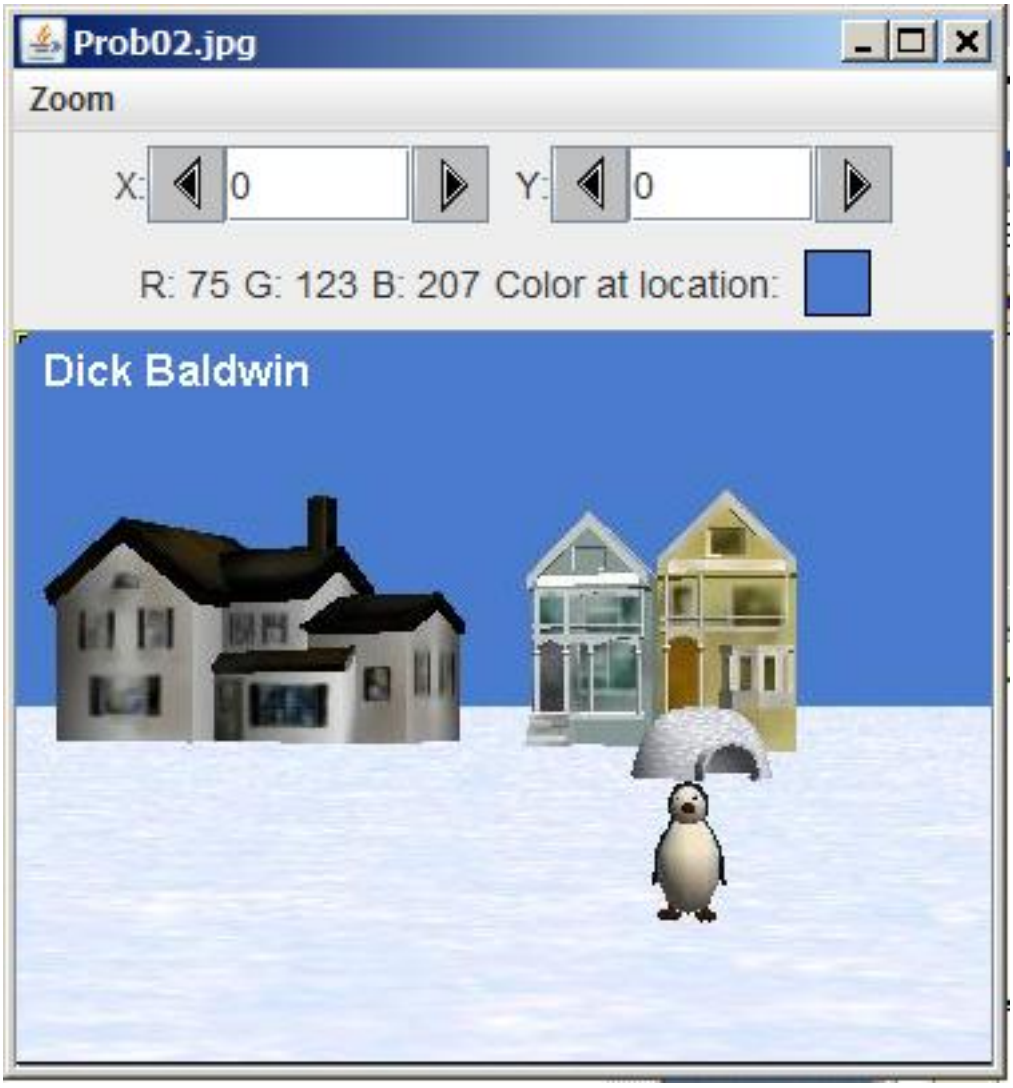
The second part of the test is to determine if the student can implement the algorithm once it is established and also satisfy some requirements for text output on the command line screen. Among other things, this requires that the student be able to:

- Create a **Picture** object from an image file.
- Write an accessor method to return a reference to the **Picture** object.
- Modify the pixels in the picture according to the algorithm.
- Display the raw picture and the modified picture in **PictureExplorer** objects by calling the **explore** method on the **Picture** object before and after it is modified.

Program output

The raw image is displayed in the **PictureExplorer** window shown in Image 1 (p. 875) .

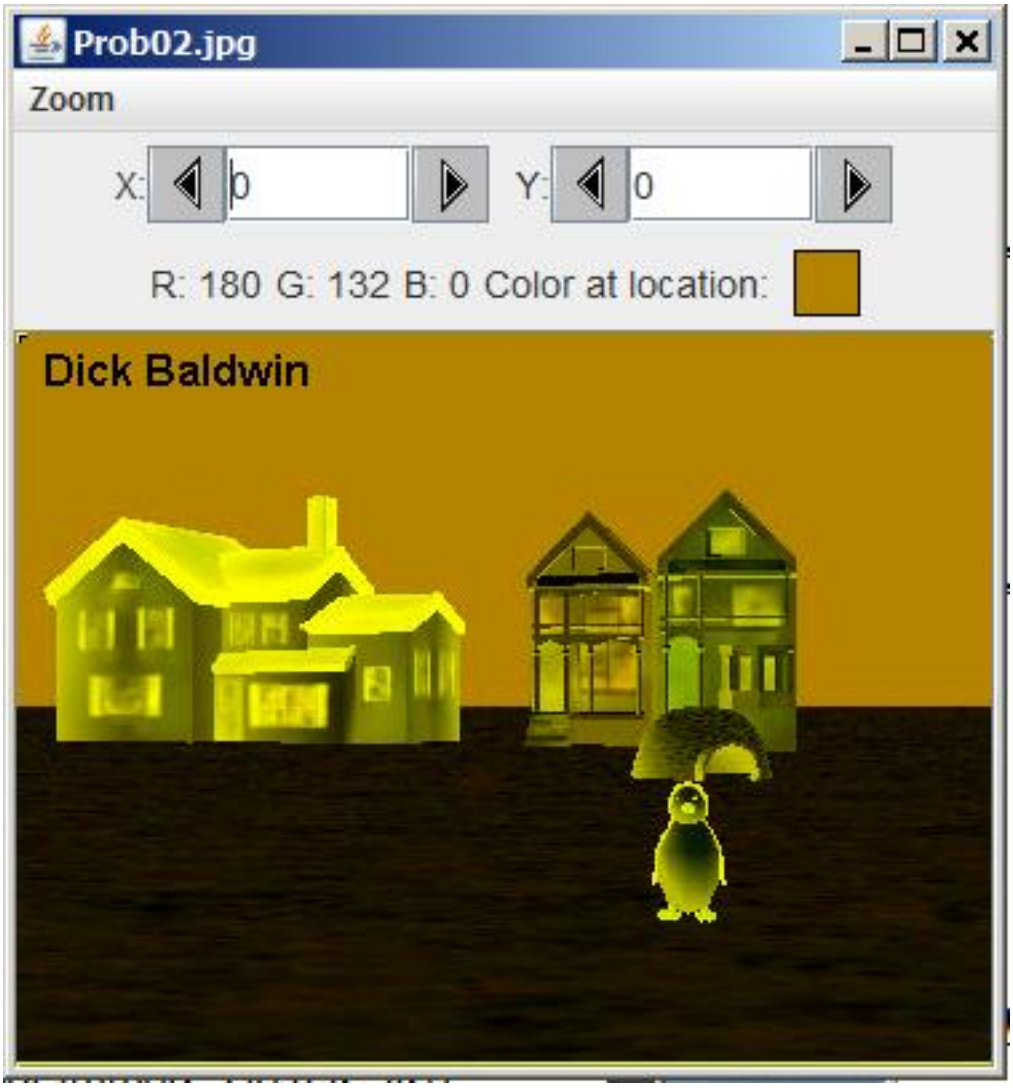
Image 1: The raw image.



5.214

The modified image is shown in the **PictureExplorer** window in Image 2 (p. 876) .

Image 2: The modified image.



5.215

The required output on the command-line screen is shown by the last two lines of text in Image 3 (p. 877) . The other text in Image 3 (p. 877) is produced by the system during the compilation and execution process.

Image 3: Output text on the command line screen.

```
    java version "1.6.0_14"
Java(TM) SE Runtime Environment (build 1.6.0_14-b08)
Java HotSpot(TM) Client VM (build 14.0-b16, mixed mode,
 sharing)
javac 1.6.0_14

Dick Baldwin
Picture, filename Prob02.jpg height 274 width 365
```

5.216

The algorithm

The algorithm required to transform the image from Image 1 (p. 875) to Image 2 (p. 876) is:

- Set the blue color value for every pixel to zero.
- Invert the red and green color values for every pixel.

A color value is inverted by subtracting the value from 255.

Obvious that the blue color value is reduced to zero

It should be obvious to the student when comparing the two images in the **PictureExplorer** objects that the blue pixel value has been set to zero for every pixel in the modified image.

Color inversion is not quite so obvious

Deducing that the red and green colors in the output pixels are the inverse of the red and green colors in the input image isn't as straightforward. However, color inversion is one of the examples provided in the Ericson textbook, so that should serve as a clue to the student. I have also published several online tutorials that involve color inversion.

The implementation of the algorithm will be explained below.

5.3.7.4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in Listing 6 (p. 884) near the end of the module.

I will begin with the driver class named **Prob02** , which is shown in its entirety in Listing 1 (p. 878) .

Listing 1: The driver class.

```
    public class Prob02{//the driver class
  public static void main(String[] args){
    Prob02Runner obj = new Prob02Runner();

    obj.run();

    System.out.println(obj.getPicture());
  }//end main
}//end class Prob02
```

5.217

You should already be familiar with everything in Listing 1 (p. 878) . The most important aspect of Listing 1 (p. 878) for purposes of this discussion is the call to the **run** method belonging to the object instantiated from the **Prob02Runner** class. I will explain the **run** method shortly.

Beginning of the class named Prob02Runner

The class definition for the class named **Prob02Runner** begins in Listing 2 (p. 878) .

Listing 2: Beginning of the class named Prob02Runner.

```
    class Prob02Runner{
  private Picture pic = new Picture("Prob02.jpg");

  public Prob02Runner(){//constructor
    System.out.println("Dick Baldwin");
  }//end constructor
  //-----------------------------------------------------//

  //Accessor method
  public Picture getPicture(){return pic;}
```

5.218

Again, you should be familiar with everything in Listing 2 (p. 878) . I will simply highlight the instantiation of a new **Picture** object using an image file as input and the saving of a reference to that object in the private instance variable named **pic** .

The beginning of the run method

The **run** method begins in Listing 3 (p. 879) . This is where the action is, so to speak.

Listing 3: The beginning of the run method.

```
public void run(){

pic.addMessage("Dick Baldwin",10,20);

pic.explore();
```

5.219

You are already familiar with the call to the **addMessage** method to add my name as text to the image encapsulated in the **Picture** object. *(See Image 1 (p. 875) .)*

The explore method

The call to the **explore** method is new to this module.

The **explore** method is defined in the **SimplePicture** class, which is the superclass of the **Picture** class. The method is inherited into the **Picture** class.

javadocs description of the explore method

The javadocs description of this method is shown in Image 4 (p. 879) .

Image 4: javadocs description of the explore method.

```
Method to open a picture explorer on a copy of this simple picture.
```

5.220

Result of calling the explore method

The result of calling the **explore** method in Listing 3 (p. 879) is to create and display the **PictureExplorer** object shown in Image 1 (p. 875) .

Very important capability

The availability of the **explore** method and the **PictureExplorer** class is very important in at least two respects:

- The **explore** method makes it easy to display copies of an image at various stages during the processing of the image. Once the **PictureExplorer** object is created and displayed, it won't be effected by subsequent changes to the image.
- The availability of a **PictureExplorer** object makes it easy to manually analyze the colors of the individual pixels in an image encapsulated in that object.

Implementing the algorithm

The code in Listing 4 (p. 880) implements the algorithm required to modify the original image to make it look like the image shown in Image 2 (p. 876) .

Listing 4: Implementing the algorithm.

```
    Pixel[] pixelArray = pic.getPixels();

for(Pixel pixel:pixelArray ){
  pixel.setRed(255 - pixel.getRed());
  pixel.setGreen(255 - pixel.getGreen());
  pixel.setBlue(0);
}//end for loop
```

5.221

In particular, the code in Listing 4 (p. 880) sets the blue color components to 0 and inverts the red and green color components for every pixel in the picture.

One of several approaches

There are several ways to do this, and this is only one of those ways. This approach makes use of a method named **getPixels** that is defined in the **SimplePicture** class and inherited into the **Picture** class.

Very useful when...

This approach is particularly useful when you want to perform the same action on every pixel in an image. The advantage is that you don't have to worry about horizontal and vertical coordinates with this approach. Access to all of the pixels is provided in a one-dimensional array.

javadocs description of the getPixels method

The javadocs description of this method is shown in Image 5 (p. 880) .

Image 5: javadocs description of the getPixels method.

```
Method to get a one-dimensional array of
Pixels for this simple picture.

Returns: a one-dimensional array of Pixel
objects starting with y=0 to y=height-1
and x=0 to x=width-1.
```

5.222

What is a Pixel object?

An object of Ericson's **Pixel** class encapsulates an individual pixel from an image. Image 6 (p. 881) shows the javadocs description of the **Pixel** class.

Image 6: javadocs description of the Pixel class.

```
    Class that references a pixel in a picture.

A pixel has an x and y location in a picture.

A pixel knows how to get and set the red, green,
blue, and alpha values in the picture.

A pixel also knows how to get and set the color
using a Color object.
```

5.223

Many methods available

The **Pixel** class defines a large number of methods. Once you have a reference to a **Pixel** object, you can manipulate the underlying pixel encapsulated in that object in a variety of ways.

Get the pixels in the image

Recall that a reference to the **Picture** object that encapsulates our image is stored in the variable named **pic** . *(See Listing 2 (p. 878) .)*

Listing 4 (p. 880) begins by calling the **getPixels** method on that reference.

All of the pixels in the image are returned in a one-dimensional array.

A reference to the array is stored in a local reference variable of type **Pixel[]** named **pixelArray** .

A for-each loop

A special kind of **for** loop *(often called a for-each loop)* is used to access and process each pixel in the array. *(A conventional* ***for*** *loop could also be used here.)*

During each iteration of the loop...

The three statements inside the loop modify the color values of a single pixel.

The first two statements invert the red and green color values by subtracting the values from 255.

The third statement in the loop sets the blue color value to zero.

When the loop terminates...

Every pixel in the image will have been modified as described above.

Not a reversible process

Because the blue color values were set to zero, the image has now been modified in an irreversible manner.

A reversible process

However, if the blue color values had also been inverted, the process would be reversible.

All that would be necessary to recover the original image would be to invert all of the pixels again.

An important process

Color inversion is a very important process in many areas of computing that involve images. The process is:

- Computationally cheap
- Very fast
- Usually visually obvious
- Totally reversible

Often used to highlight selected images

For example, many software program invert all of the colors in an image when it is selected for some purpose, such as copying to the clipboard. Then the colors are restored to their original values when the image is deselected.

Next to redeye correction, color inversion is probably the most commonly used color modification algorithm in use in modern image processing.

Display again and terminate

The variable named **pic** still contains a reference to the original **Picture** object. However, the image that is encapsulated in that object has been significantly modified.

Listing 5 (p. 882) calls the **explore** method again, creating and displaying another **PictureExplorer** object that encapsulates a copy of the **Picture** object with the modified image.

Listing 5: Display again and terminate.

```
        pic.explore();

  }//end run method
}//end class Prob02Runner
```

5.224

The result is shown in Image 2 (p. 876) .

The end of the program

Listing 5 (p. 882) also signals the end of the **run** method and the end of the **Prob02Runner** class.

Return control to main

The **run** method terminates and returns control to the **main** method in Listing 1 (p. 878) .

The code in the **main** method calls a *getter* method to get a reference to the **Picture** object.

The reference is passed to the **println** method, which displays the information about the **Picture** object in the last line of Image 3 (p. 877) .

The program terminates

Then the **main** method terminates, at which time the program terminates and returns control to the operating system.

5.3.7.4.1 Run the program

I encourage you to copy the code from Listing 6 (p. 884) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

You can download a copy of the raw image file here [81] .

5.3.7.5 Summary

In this module, you learned how to invert images and how to display images in **PictureExplorer** objects.

5.3.7.6 What's next?

Y ou will learn how to implement a space-wise linear color-modification algorithm in the next module.

[81] http://cnx.org/content/m44203/latest/Prob02.jpg

5.3.7.7 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 02 [82]
  - Part01 [83]
  - Part02 [84]
  - Part03 [85]
  - Part04 [86]

5.3.7.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Image Processing Algorithms, Image Inversion, and PictureExplorer Objects
- File: Java3004.htm
- Published: 07/29/12
- Revised: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.7.9 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 6 (p. 884) below.

[82] http://www.youtube.com/playlist?list=PL713DB9A1FF4B92DF
[83] http://www.youtube.com/watch?v=SVq_IN4TsTs
[84] http://www.youtube.com/watch?v=vcVLr8Z1mo4
[85] http://www.youtube.com/watch?v=M1Nns7vYTiM
[86] http://www.youtube.com/watch?v=Qw-yzEGuFJU

Listing 6: Complete program listing.

```
    /*File Prob02 Copyright 2008 R.G.Baldwin
**********************************************************/

public class Prob02{//the driver class
  public static void main(String[] args){
    Prob02Runner obj = new Prob02Runner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Prob02
//========================================================//

class Prob02Runner{
  private Picture pic = new Picture("Prob02.jpg");

  public Prob02Runner(){//constructor
    System.out.println("Dick Baldwin");
  }//end constructor
  //-------------------------------------------------------//

  //Accessor method
  public Picture getPicture(){return pic;}
  //-------------------------------------------------------//

  //This method is where the action is.
  public void run(){
    //Display the raw picture.
    pic.addMessage("Dick Baldwin",10,20);
    pic.explore();

    //Set the blue color components to 0 and invert the
    // red and green color components for every pixel in
    // the picture.
    Pixel[] pixelArray = pic.getPixels();
    for(Pixel pixel:pixelArray ){
      pixel.setRed(255 - pixel.getRed());
      pixel.setGreen(255 - pixel.getGreen());
      pixel.setBlue(0);
    }//end for loop

    //Display the modified picture
    pic.explore();
  }//end run method
}//end class Prob02Runner
```

5.225

-end-

5.3.8 Java3004r Review[87]

5.3.8.1 Table of Contents

5.3.8.2 Preface

This module contains review questions and answers keyed to the module titled Image Java3004: Processing Algorithms, Image Inversion, and PictureExplorer Objects [88] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.8.3 Questions

5.3.8.3.1 Question 1 .

True or False? A color value is inverted by subtracting the value from 256.

Answer 1 (p. 895)

5.3.8.3.2 Question 2

True or False? The code in Listing 1 (p. 886) instantiates a new object of the **PictureExplorer** class.

Listing 1. Question 2.

```
  class Prob02Runner{
private Picture pic = new Picture("Prob02.jpg");

public Prob02Runner(){//constructor
  System.out.println("Dick Baldwin");
}//end constructor
//-----------------------------------------------------//

//Accessor method
public Picture getPicture(){return pic;}
```

5.226

Answer 2 (p. 894)

[87] This content is available online at <http://cnx.org/content/m45763/1.4/>.
[88] http://cnx.org/content/m44203

5.3.8.3.3 Question 3

True or False? The code in Listing 2 (p. 887) causes an image to be displayed in the format shown in Image 1 (p. 888) .

Listing 2. Question 2.

```
public void run(){

pic.addMessage("Dick Baldwin",10,20);

pic.explore();
```

5.227

Image 1. Question 3.

5.228

Answer 3 (p. 893)

5.3.8.3.4 Question 4

True or False? The availability of the **explore** method and the **PictureExplorer** class is very important in at least two respects:

- The **explore** method makes it easy to display copies of an image at various stages during the processing of the image. Once the **PictureExplorer** object is created and displayed, it won't be effected by subsequent changes to the image.
- The availability of a **PictureExplorer** object makes it easy to manually analyze the colors of the individual pixels in an image encapsulated in that object.

Answer 4 (p. 893)

5.3.8.3.5 Question 5

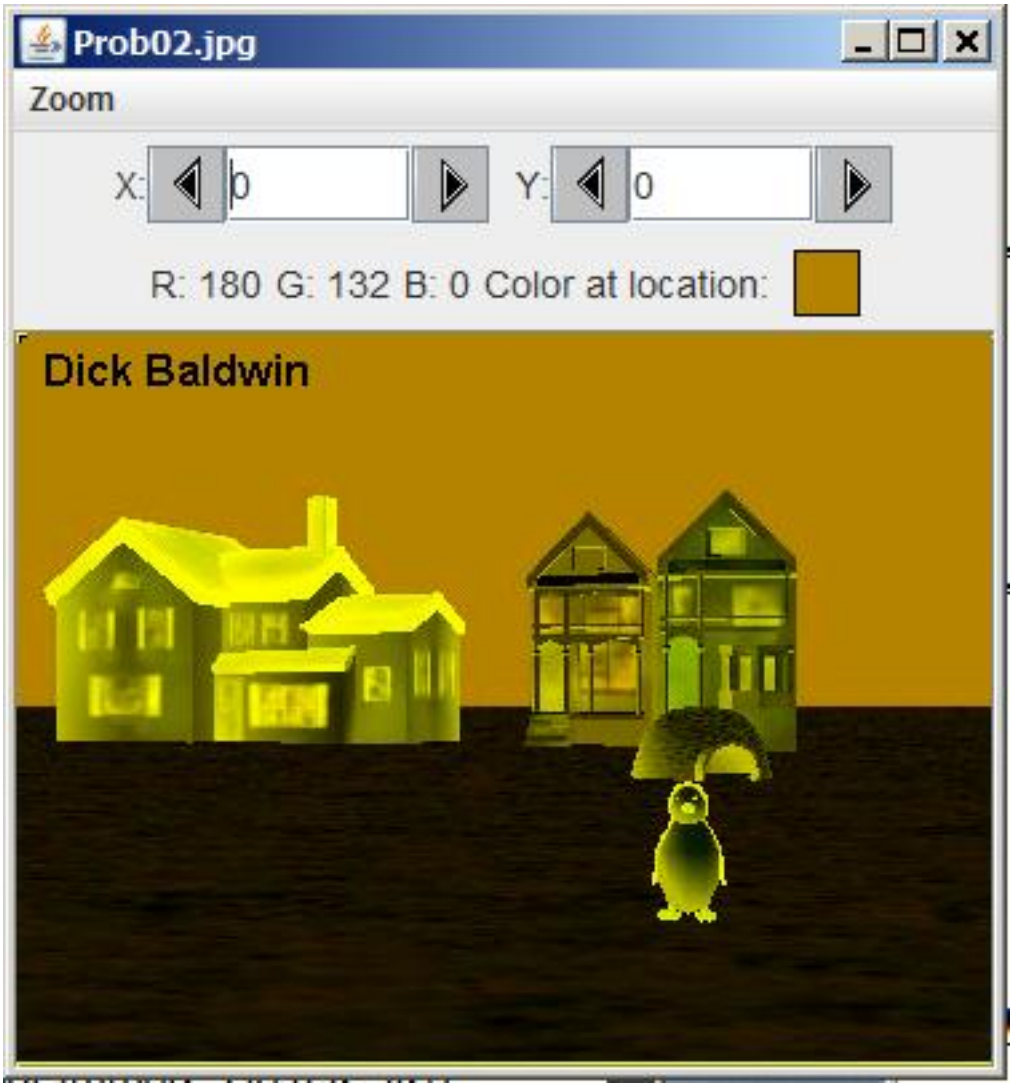
True or False? The algorithm shown in Listing 3 (p. 889) can be used to cause the image shown in Image 2 (p. 894) to be transformed into the image shown in Image 3 (p. 890) .

Listing 3. Question 5.

```
    Pixel[] pixelArray = pic.getPixels();

for(Pixel pixel:pixelArray ){
  pixel.setRed(255 - pixel.getRed());
  pixel.setGreen(255 - pixel.getGreen());
  pixel.setBlue(0);
}//end for loop
```

5.229

Image 3. Question 5.



5.230

Answer 5 (p. 893)

5.3.8.3.6 Question 6

True or False? An object of Ericson's **Pixel** class encapsulates an individual pixel from an image.
Answer 6 (p. 893)

5.3.8.3.7 Question 7

True or False? The **Pixel** class defines a single method that can be called to invert the color of the pixel.
Answer 7 (p. 893)

5.3.8.3.8 Question 8

True or False? The **getPixels** method belonging to a **Picture** object returns references to all of the **Pixel** objects encapsulated in the picture in a two-dimensional array where the dimensions of the array represent the horizontal and vertical coordinates of each pixel.
Answer 8 (p. 893)

5.3.8.3.9 Question 9

True or False? The **for** loop shown in Listing 3 (p. 889) is of a type that is often referred to as a ***for-each*** loop.
Answer 9 (p. 893)

5.3.8.3.10 Question 10

True or False? The three statements inside the loop in Listing 3 (p. 889) modify the color values of a single pixel.
Answer 10 (p. 893)

5.3.8.3.11 Question 11

True or False? If the colors of all the pixels in an image are inverted, the process is completely reversible. All that is necessary to recover the original image is to invert all of the pixels again.
Answer 11 (p. 893)

5.3.8.4 Images

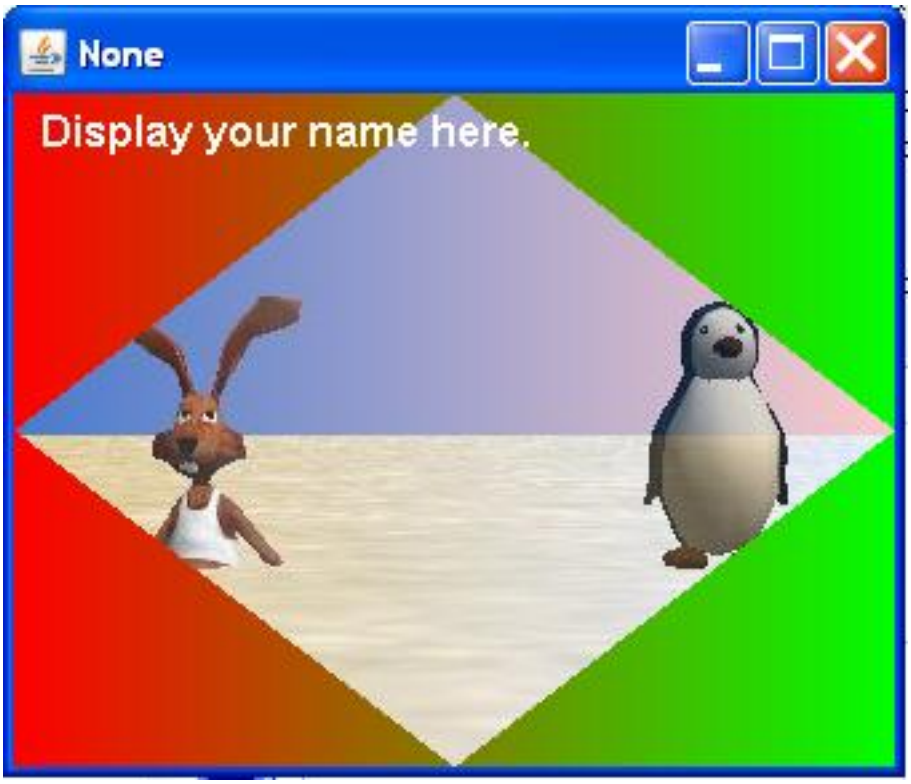
- Image 1 (p. 888) . Question 3.
- Image 2 (p. 894) . Answer 3.
- Image 3 (p. 890) . Question 5.

5.3.8.5 Listings

- Listing 1 (p. 886) . Question 2.
- Listing 2 (p. 887) . Question 2.
- Listing 3 (p. 889) . Question 5.

What is the meaning of the following two images?
This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.8.6 Answers

5.3.8.6.1 Answer 11

True.
Back to Question 11 (p. 891)

5.3.8.6.2 Answer 10

True.
Back to Question 10 (p. 891)

5.3.8.6.3 Answer 9

True.
Back to Question 9 (p. 891)

5.3.8.6.4 Answer 8

False. The **getPixels** method belonging to a **Picture** object returns references to all of the **Pixel** objects encapsulated in the picture in a one-dimensional array.
Back to Question 8 (p. 891)

5.3.8.6.5 Answer 7

False. The **Pixel** class defines a large number of methods. Once you have a reference to a **Pixel** object, you can manipulate the underlying pixel encapsulated in that object in a variety of ways.
Back to Question 7 (p. 891)

5.3.8.6.6 Answer 6

True.
Back to Question 6 (p. 890)

5.3.8.6.7 Answer 5

True.
Back to Question 5 (p. 888)

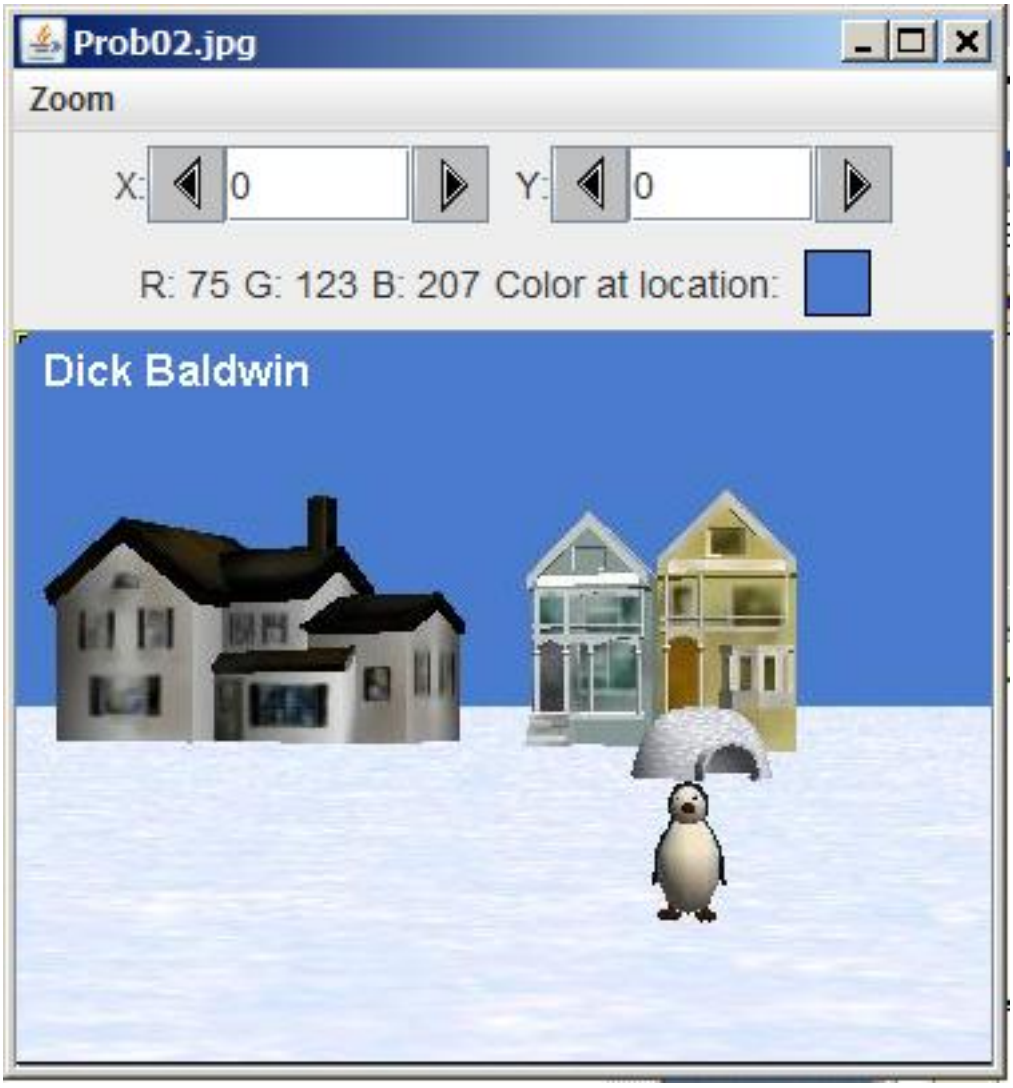
5.3.8.6.8 Answer 4

True.
Back to Question 4 (p. 888)

5.3.8.6.9 Answer 3

False. The code in Listing 2 (p. 887) causes an image to be displayed in the format shown in Image 2 (p. 894) .

Image 2. Answer 3.



5.231

Back to Question 3 (p. 887)

5.3.8.6.10 Answer 2

False. The code in Listing 1 (p. 886) instantiates a new object of the **Picture** class.

Back to Question 2 (p. 886)

5.3.8.6.11 Answer 1

False. A color value is inverted by subtracting the value from **255** .

Back to Question 1 (p. 886)

5.3.8.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3004r Review: Image Processing Algorithms, Image Inversion, and PictureExplorer Objects
- File: Java3004r.htm
- Published: 02/10/13
- Revised: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.9 Java3004s Slides[89]

5.3.9.1 Table of Contents

5.3.9.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3004: Image Processing Algorithms, Image Inversion, and PictureExplorer Objects [90] .

Click here [91] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

[89] This content is available online at <http://cnx.org/content/m45622/1.4/>.
[90] http://cnx.org/content/m44203
[91] http://cnx.org/content/m45622/latest/a0-Index.htm

5.3.9.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3004s Slides: Image Processing Algorithms, Image Inversion, and Picture-Explorer Objects
- File: Java3004s.htm
- Published: 01/05/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.10 Java3006: Implementing a space-wise linear color-modification algorithm.[92]

5.3.10.1 Table of Contents

[92] This content is available online at <http://cnx.org/content/m44204/1.6/>.

5.3.10.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [93] .

5.3.10.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.10.2.1.1 Images

- Image 1 (p. 899) . The raw image.
- Image 2 (p. 900) . The modified image.
- Image 3 (p. 901) . Text output on the command-line screen.

5.3.10.2.1.2 Listings

- Listing 1 (p. 901) . The driver class named Prob03.
- Listing 2 (p. 902) . Beginning of the class named Prob03Runner.
- Listing 3 (p. 902) . The beginning of the run method.
- Listing 4 (p. 903) . Beginning of the for loop.
- Listing 5 (p. 904) . Compute the column number and scale factors.
- Listing 6 (p. 904) . Apply the scale factors.
- Listing 7 (p. 905) . Display the modified image.
- Listing 8 (p. 907) . Complete program listing.

5.3.10.3 Preview

The program that I will explain in this module is designed to be used as a test of the student's understanding of programming using Java and Ericson's media library.

The student is provided an image file named **Prob03.jpg** along with a written specification of a space-wise linear image modification algorithm.

Implement the algorithm

The primary purpose of the test is to determine if the student can implement the algorithm and also satisfy some requirements for text output on the command line screen. Among other things, this requires that the student be able to:

- Create a **Picture** object from an image file.
- Write an *accessor* method to return a reference to the **Picture** object.
- Modify the pixels in the picture according to the specified algorithm.
- Display the raw picture and the modified picture in **PictureExplorer** objects by calling the **explore** method on the **Picture** object before and after it is modified.

The algorithm

Scale the blue and green color components by a scale factor that is less than or equal to 1.0. The green scale factor:

[93] http://cnx.org/content/m44148/latest/

- Is equal to 1.0 on the left side of the image
- Is equal to 0.0 on the right side of the image
- Decreases linearly with distance going from left to right across the image.

The blue scale factor

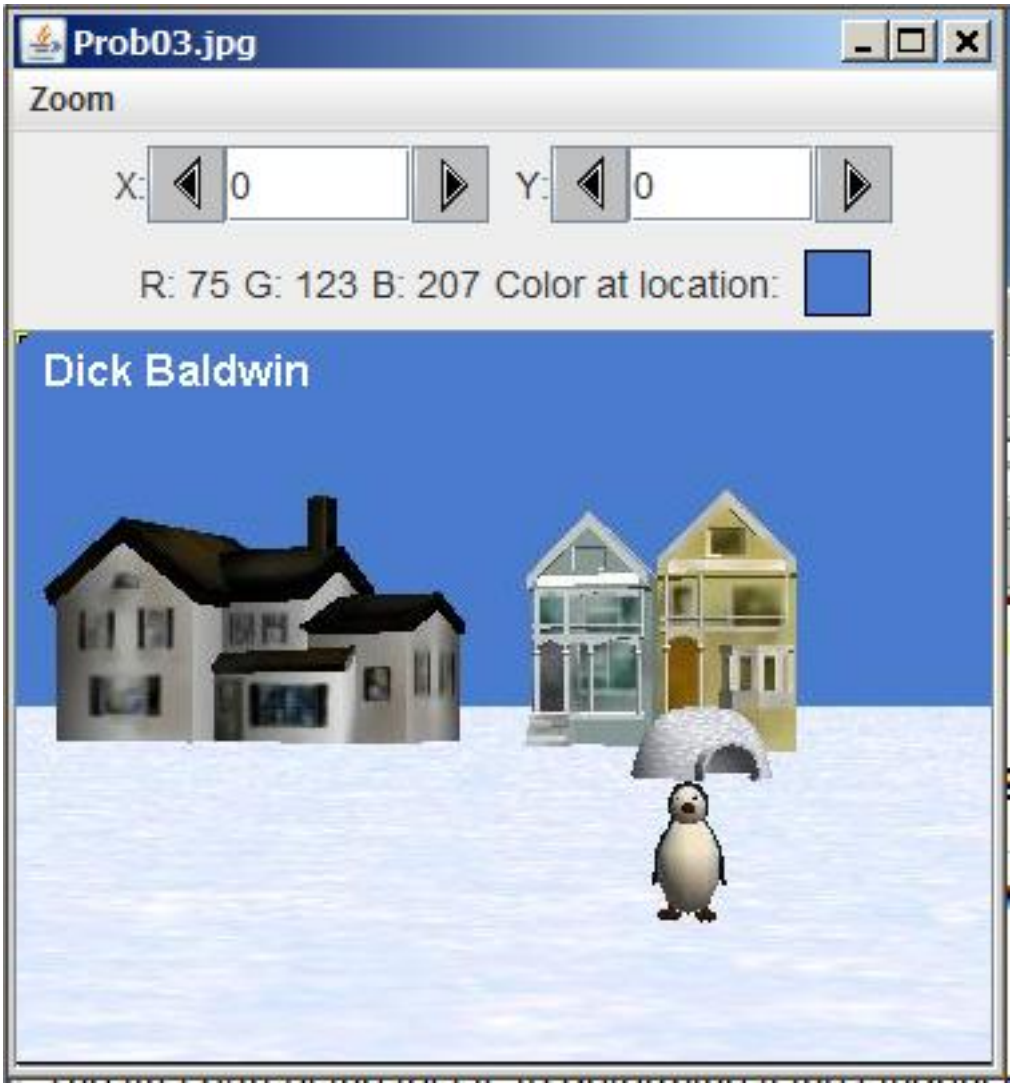
- Is 0.0 on the left side of the image
- Is 1.0 on the right side of the image
- Increases linearly with distance going from left to right across the image.

Do not scale the red color component.

The program output

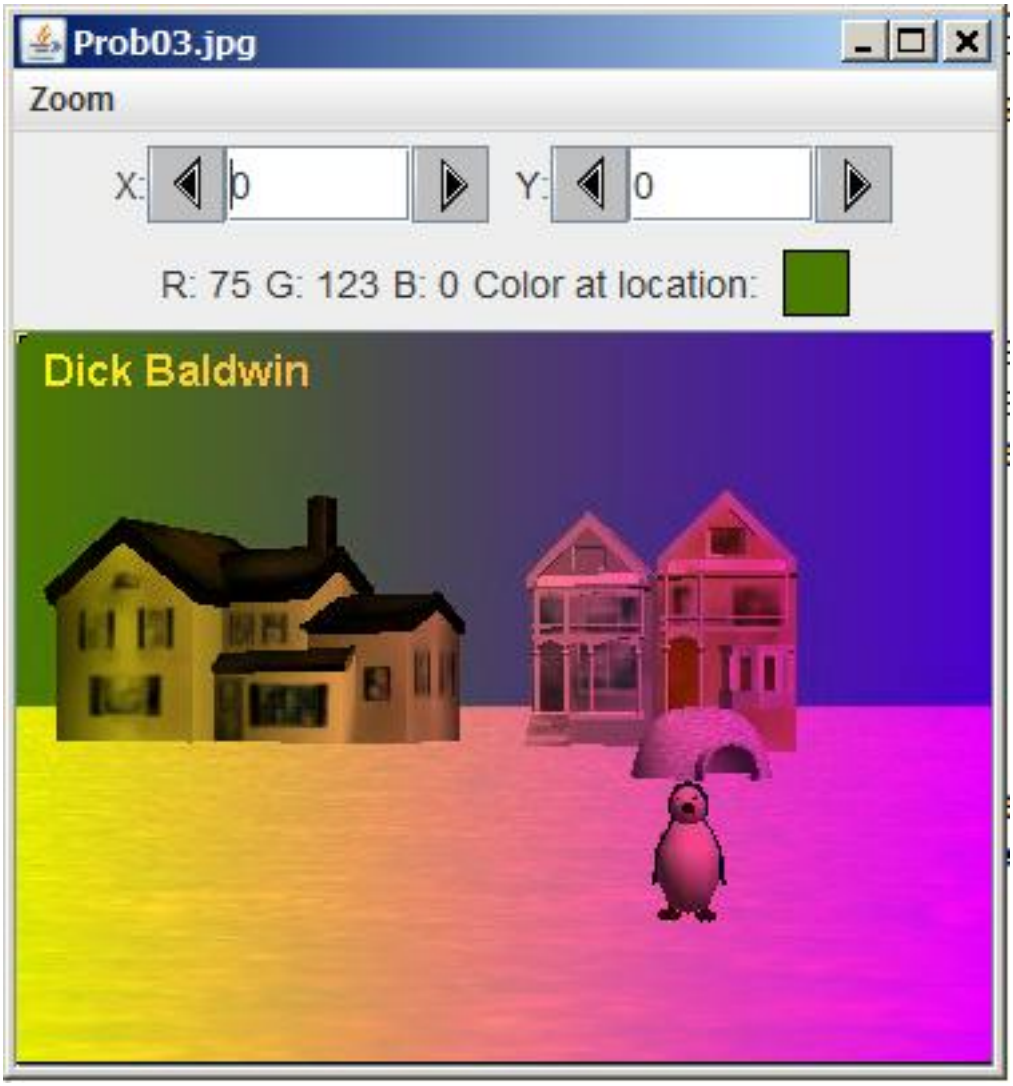
The program produces the images shown in Image 1 (p. 899) and Image 2 (p. 900) and produces the output text shown in Image 3 (p. 901) on the command line screen.

Image 1: The raw image.



5.232

Image 2: The modified image.



5.233

Image 3: Text output on the command-line screen.

```
    java version "1.6.0_14"
Java(TM) SE Runtime Environment (build 1.6.0_14-b08)
Java HotSpot(TM) Client VM (build 14.0-b16,
mixed mode, sharing)
javac 1.6.0_14

Dick Baldwin
Picture, filename Prob03.jpg height 274 width 365
```

5.234

The required output on the command-line screen is shown by the last two lines of text in Image 3 (p. 901) . The remaining text in Image 3 (p. 901) is produced by the system during the compilation and execution process.

5.3.10.4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in Listing 8 (p. 907) near the end of the module.

I will begin with the driver class named **Prob03** , which is shown in its entirety in Listing 1 (p. 901) .

Listing 1: The driver class named Prob03.

```
    public class Prob03{
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Prob03
```

5.235

There is nothing in Listing 1 (p. 901) that I haven't explained in earlier modules. Therefore, no explanation of the code in Listing 1 (p. 901) should be required.

Beginning of the class named Prob03Runner

The class definition for the class named **Prob03Runner** begins in Listing 2 (p. 902) .

Listing 2: Beginning of the class named Prob03Runner.

```
  class Prob03Runner{
//Instantiate the Picture object.
private Picture pic = new Picture("Prob03.jpg");

public Prob03Runner(){//constructor
  System.out.println("Dick Baldwin");
}//end constructor
//-----------------------------------------------------//

//Accessor method
public Picture getPicture(){return pic;}
```

5.236

Once again, there is nothing in Listing 2 (p. 902) that I haven't explained before. I included it here simply for the sake of continuity.

The beginning of the run method

The **run** method begins in Listing 3 (p. 902) . The **run** method is where most of the interesting action takes place.

Listing 3: The beginning of the run method.

```
  public void run(){
pic.addMessage("Dick Baldwin",10,20);
//Display a PictureExplorer object.
pic.explore();

//Get an array of Pixel objects.
Pixel[] pixels = pic.getPixels();

//Declare working variables
Pixel pixel = null;
int green = 0;
int blue = 0;
int width = pic.getWidth();
double greenScale = 0;
double blueScale = 0;
```

5.237

Much of what you see in Listing 3 (p. 902) has been explained in earlier modules. However, Listing 3 (p. 902) does deserve a few comments.

Display the raw image

The call to the **explore** method produces the output shown in Image 1 (p. 899) .

Get an array of Pixel data

The call to the **getPixels** method in Listing 3 (p. 902) returns a reference to a one-dimensional array object. The elements in the array are references to **Pixel** objects, where each **Pixel** object represents a single pixel in the image. I will explain the organization of the pixel data later [94] .

Get the width of the image

The call to the **getWidth** method in Listing 3 (p. 902) returns an **int** value that specifies the width of the image in pixels. This value will be used later to compute the column to which each pixel belongs.

Local variables

Listing 3 (p. 902) declares six local variables. The purpose of these variables should become clear during the explanation of the code that implements the algorithm.

Implementation of the algorithm

The algorithm [95] is implemented by the code in a conventional **for** loop, which begins in Listing 4 (p. 903) .

Listing 4: Beginning of the for loop.

```
  for(int cnt = 0;cnt < pixels.length;cnt++){

pixel = pixels[cnt];

green = pixel.getGreen();

blue = pixel.getBlue();
```

5.238

The loop iterates through the array of **Pixel** data, modifying the colors in one pixel during each iteration.

The length property of the array object

Every array object in Java contains a **length** property that contains the number of elements in the array. The value of this property is used in the conditional clause in the **for** loop in Listing 4 (p. 903) to establish when the end of the array has been reached in order to terminate the loop.

Get reference to the next Pixel object

The first statement inside the **for** loop in Listing 4 (p. 903) gets a reference to a **Pixel** object from the next array element. That reference is stored in the local variable of type **Pixel** named **pixel** that was declared in Listing 3 (p. 902) .

Get the red and green color values for the current pixel

Having gotten a reference to the **Pixel** object, the next statement calls the **getGreen** method on that reference to get and save the value of the green color component in the current pixel.

Similarly, the statement following that one gets and saves the value of the blue color component in the current pixel.

Both values are returned as type **int** , and can range in value from 0 up to and including 255.

Objective is to scale the green and blue color values

[94] http://cnx.org/content/m44204/latest/Java3006old.htm#Organization_of_the_pixel_data
[95] http://cnx.org/content/m44204/latest/Java3006old.htm#The_algorithm

Recall that the objective is to scale the green and blue color values on a column by column basis, going from left to right across the image shown in Image 1 (p. 899) in order to produce the output image shown in Image 2 (p. 900) .

Organization of the pixel data

The pixel data is stored in the array on a row by row basis. In other words, the first **width** elements contain references to pixels in the first row of pixels going from left to right across the screen. The next **width** elements contain references to pixels in the second row of pixels, etc.

Compute the column number and scale factors

Listing 5 (p. 904) uses the modulus operator to compute the column number for each **Pixel** object.

Listing 5: Compute the column number and scale factors.

```
    //Compute the column number and use it to compute
// the scale factor.
int col = cnt%width;

greenScale = (double)(width - col)/width;
blueScale = (double)(col)/width;
```

5.239

An exercise for the student

Knowing the column number in which the pixel is located, the next step is to compute the green and blue scale factors necessary to satisfy the algorithm.

I will leave it as an exercise for the student to think about how the expressions contained in the last two statements in Listing 5 (p. 904) cause the two scale factors to vary linearly from left to right across the image in accordance with the requirements of the algorithm. *(Think about the equation of a straight line from your high school math classes.)*

Apply the scale factors

The **Pixel** class contains methods named **setRed** , **setGreen** , and **setBlue** that can be called to set the color values for the pixel represented by a **Pixel** object.

Listing 6 (p. 904) computes new values for the red and green components based on the existing color values for the pixel and the scale factors computed in Listing 5 (p. 904) .

Listing 6: Apply the scale factors.

```
    pixel.setGreen((int)(green * greenScale));
  pixel.setBlue((int)(blue * blueScale));
 }//end for loop
```

5.240

Then Listing 6 (p. 904) calls the **setGreen** and **setBlue** methods on the **Pixel** object to set the green and blue color values to the newly computed values.

The end of the for loop

Listing 6 (p. 904) also signals the end of the **for** loop that began in Listing 4 (p. 903) .

Display the modified image

Finally, Listing 7 (p. 905) calls the **explore** method again to display the image shown in Image 2 (p. 900) .

Listing 7: Display the modified image.

```
        pic.explore();

  }//end run method
}//end class Prob03Runner
```

5.241

The end of the program

Listing 7 (p. 905) also signals the end of the **run** method and the end of the **Prob03Runner** class.

Return control to main

The **run** method terminates and returns control to the **main** method shown in Listing 1 (p. 901) .

The code in the **main** method calls a *getter* method to get a reference to the **Picture** object.

The reference is passed to the **println** method, which displays the information about the **Picture** object in the last line of Image 3 (p. 901) .

The program terminates

Then the **main** method terminates, at which time the program terminates and returns control to the operating system.

5.3.10.4.1 Run the program

I encourage you to copy the code from Listing 8 (p. 907) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

You can download a copy of the required input image file here [96] .

5.3.10.5 Summary

In this module, I showed you how to implement an algorithm that causes the green and blue color values in an image to change in a linear fashion going from left to right across the image.

5.3.10.6 What's next?

You will learn more about abstract methods, abstract classes, and overridden methods in the next lesson. Very importantly, you will learn more about overriding the toString method.

5.3.10.7 Online video links

Select the following links to view online video lectures on the material in this module.

[96] http://cnx.org/content/m44204/latest/Prob03.jpg

- ITSE 2321 Lecture 03 [97]
  - Part01 [98]
  - Part02 [99]
  - Part03 [100]

5.3.10.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Implementing a space-wise linear color-modification algorithm
- File: Java3006.htm
- Published: 07/30/12
- Revised: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.10.9 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 8 (p. 907) below.

[97] http://www.youtube.com/playlist?list=PL43014B9ED4419642
[98] http://www.youtube.com/watch?v=pdqKvAuzkhg
[99] http://www.youtube.com/watch?v=aXoSD7oauLQ
[100] http://www.youtube.com/watch?v=1iRiXhTPmMU

Listing 8: Complete program listing.

```
    /*File Prob03 Copyright 2008 R.G.Baldwin
**********************************************************/

public class Prob03{
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Prob03
//=========================================================//

class Prob03Runner{
  //Instantiate the Picture object.
  private Picture pic = new Picture("Prob03.jpg");

  public Prob03Runner(){//constructor
    System.out.println("Dick Baldwin");
  }//end constructor
  //-------------------------------------------------------//

  //Accessor method
  public Picture getPicture(){return pic;}
  //-------------------------------------------------------//

  //This method is where the action is.
  public void run(){
    pic.addMessage("Dick Baldwin",10,20);
    //Display a PictureExplorer object.
    pic.explore();

    //Get an array of Pixel objects.
    Pixel[] pixels = pic.getPixels();

    //Declare working variables
    Pixel pixel = null;
    int green = 0;
    int blue = 0;
    int width = pic.getWidth();
    double greenScale = 0;
    double blueScale = 0;

    //Scale the blue and green color components according
    // to the algorithm given above.
    //Do not scale the red component.
    for(int cnt = 0;cnt < pixels.length;cnt++){
      //Get blue and green values for current pixel.
      pixel = pixels[cnt];
      green = pixel.getGreen();
      blue = pixel.getBlue();
```



```
      //Compute the column number and use it to compute
      // the scale factor.
      int col = cnt%width;
      greenScale = (double)(width - col)/width;
```

-end-

5.3.11 Java3006r Review[101]

5.3.11.1 Table of Contents

5.3.11.2 Preface

This module contains review questions and answers keyed to the module titled Java3006: Implementing a space-wise linear color-modification algorithm [102] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.11.3 Questions

5.3.11.3.1 Question 1 .

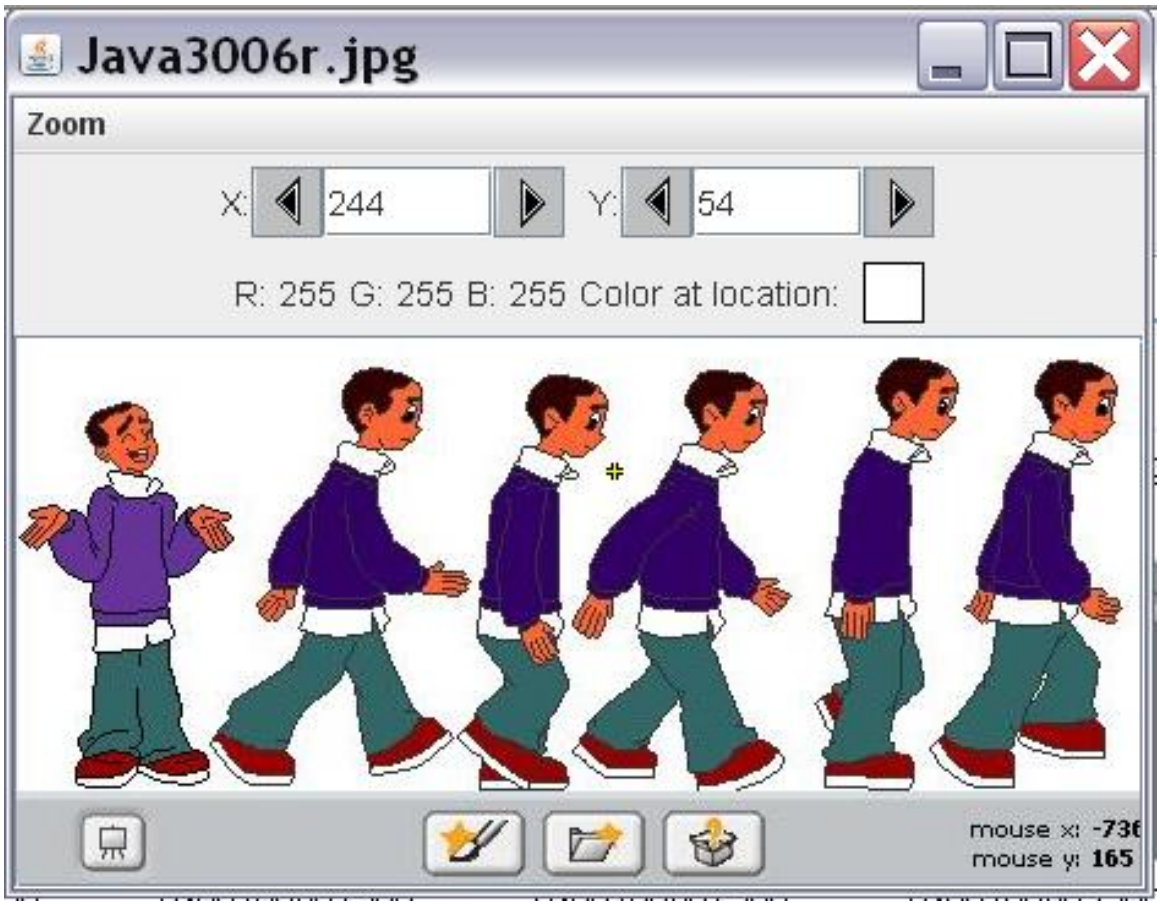
True or False? The program shown in Listing 1 (p. 910) will transform the image shown in Image 1 (p. 911) into the image shown in Image 2 (p. 912) *(or to an image that looks very similar to the image shown in Image 2* (p. 912)).

[101] This content is available online at <http://cnx.org/content/m45768/1.2/>.
[102] http://cnx.org/content/m44204

Listing 1. Source code for Java3006r.

```
/*File Java3006r Copyright 2013 R.G.Baldwin
************************************************************/

public class Java3006r{
  public static void main(String[] args){
    Java3006rRunner obj = new Java3006rRunner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Java3006r
//============================================================//

class Java3006rRunner{
  //Instantiate the Picture object.
  private Picture pic = new Picture("Java3006r.jpg");

  public Java3006rRunner(){//constructor
    System.out.println("Dick Baldwin");
  }//end constructor
  //----------------------------------------------------------//

  //Accessor method
  public Picture getPicture(){return pic;}
  //----------------------------------------------------------//

  //This method is where the action is.
  public void run(){
    pic.addMessage("Dick Baldwin",10,20);
    //Display a PictureExplorer object.
    pic.explore();

    //Get an array of Pixel objects.
    Pixel[] pixels = pic.getPixels();

    //Declare working variables
    Pixel pixel = null;
    int green = 0;
    int blue = 0;
    int width = pic.getWidth();
    double greenScale = 0;
    double blueScale = 0;

    //Scale the blue and green color components according
    // to the algorithm given above.
    //Do not scale the red component.
    for(int cnt = 0;cnt < pixels.length;cnt++){
      //Get blue and green values for current pixel.
      pixel = pixels[cnt];
      green = pixel.getGreen();
      blue = pixel.getBlue();

      //Compute the column number and use it to compute
      // the scale factor.
      int col = cnt%width;
```

Image 1. The image from Java3006r.jpg.



5.244

Image 2. Possible output image.

5.245

Answer 1 (p. 915)

5.3.11.3.2 Question 2

True or False? The output shown in Image 2 (p. 912) was produced by calling the **show** method of Ericson's **Picture** class.

Answer 2 (p. 915)

5.3.11.3.3 Question 3

True or False? A call to Ericson's **getPixels** method returns a reference to a one-dimensional array object. The elements in the array are references to **Pixel** objects, where each **Pixel** object represents a single pixel in an image.

Answer 3 (p. 915)

5.3.11.3.4 Question 4

True or False? A call to the **getWidth** method of Ericson's **Picture** class returns a **double** value that specifies the width of the image in inches.

Answer 4 (p. 915)

5.3.11.3.5 Question 5

True or False? Every array object in Java contains a **length** property that contains the number of elements in the array.

Answer 5 (p. 915)

5.3.11.3.6 Question 6

True or False? Having gotten a reference to a **Pixel** object, the **getGreen** method can be called on that reference to get the value of the red color component in the current pixel.

Answer 6 (p. 915)

5.3.11.3.7 Question 7

True or False? Red, green, and blue color values range from 0 to 256.

Answer 7 (p. 915)

5.3.11.4 Images

- Image 1 (p. 911) . The image from Java3006r.jpg.
- Image 2 (p. 912) . Possible output image.

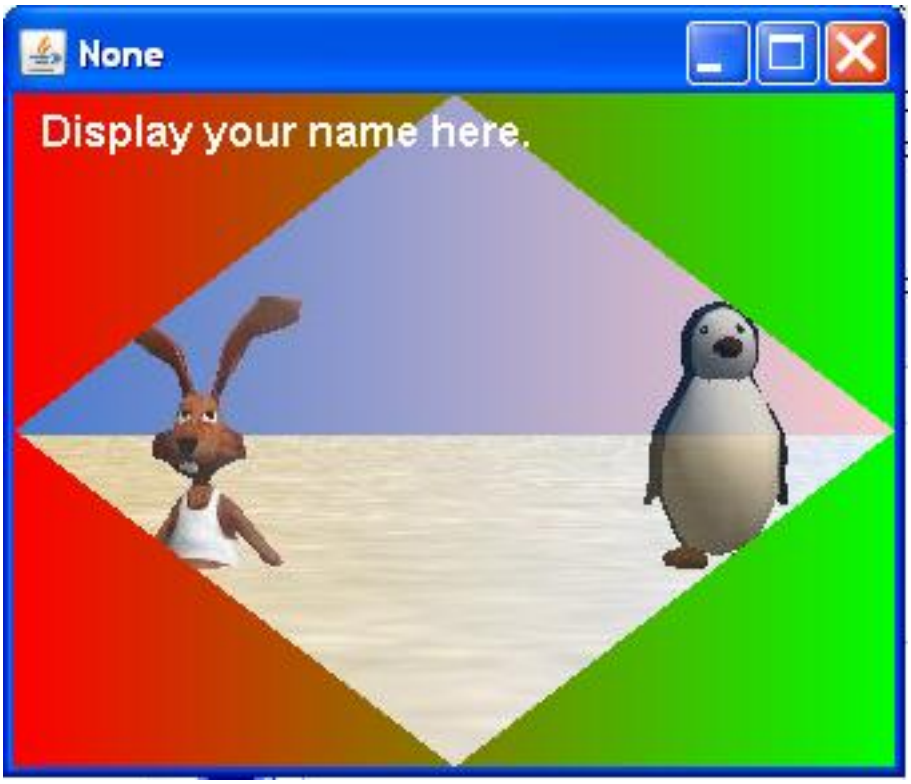
5.3.11.5 Listings

- Listing 1 (p. 910) . Source code for Java3006r.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.11.6 Answers

5.3.11.6.1 Answer 7

False. Red, green, and blue color values range from 0 to **255** .
Back to Question 7 (p. 913)

5.3.11.6.2 Answer 6

False. Having gotten a reference to a **Pixel** object, the **getGreen** method can be called on that reference to get the value of the green color component in the current pixel.
Back to Question 6 (p. 913)

5.3.11.6.3 Answer 5

True.
Back to Question 5 (p. 913)

5.3.11.6.4 Answer 4

False. A call to the **getWidth** method of Ericson's **Picture** class returns an **int** value that specifies the width of the image in pixels.
Back to Question 4 (p. 913)

5.3.11.6.5 Answer 3

True.
Back to Question 3 (p. 912)

5.3.11.6.6 Answer 2

False. The output shown in Image 2 (p. 912) was produced by calling the **explore** method of Ericson's **Picture** class.
Back to Question 2 (p. 912)

5.3.11.6.7 Answer 1

True.
Back to Question 1 (p. 909)

5.3.11.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3006r Review: Implementing a space-wise linear color-modification algorithm
- File: Java3006r.htm
- Published: 02/12/13
- Revised: 02/18/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.12 Java3006s Slides[103]

5.3.12.1 Table of Contents

5.3.12.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3006: Implementing a space-wise linear color-modification algorithm [104] .

Click here [105] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.12.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3006s Slides: Implementing a space-wise linear color-modification algorithm
- File: Java3006s.htm
- Published: 01/06/13

[103] This content is available online at <http://cnx.org/content/m45623/1.3/>.
[104] http://cnx.org/content/m44204
[105] http://cnx.org/content/m45623/latest/a0-Index.htm

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.13 Java3008: Abstract Methods, Abstract Classes, and Overridden Methods[106]

5.3.13.1 Table of Contents

5.3.13.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.3.13.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.13.2.1.1 Images

[106] This content is available online at <http://cnx.org/content/m44205/1.5/>.

5.3.13.2.1.2 Listings

5.3.13.3 Preview

The program that I will explain in this module produces no graphics and does not require the use of Ericson's media library.

OOP concepts

The program illustrates the following OOP concepts:

- Extending an abstract class.
- Parameterized constructor.
- Defining an abstract method in the superclass and overriding it in a subclass.
- Overridden **toString** method.

Program specifications

Write a program named **Prob04** that uses the class definition shown in Listing 1 (p. 919) to produce the output on the command-line screen shown in Image 1 (p. 918) .

Image 1: Program output on the command line screen.

```
    Prob04
Dick
Baldwin
95
95
```

5.246

Pseudo random data

Because the program generates and uses a pseudo random data value each time it is run, the actual values displayed in the last two lines of Image 1 (p. 918) will differ from one run to the next. However, in all cases, the two values must match.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob04** given below.

5.3.13.4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in Listing 5 (p. 926) near the end of the module.

I will begin with the driver class named **Prob04** , which is shown in its entirety in Listing 1 (p. 919) .

Listing 1: Source code for class Prob04.

```
    import java.util.*;

abstract class Prob04{
  public static void main(String[] args){
    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Prob04 objRef = new Prob04MyClass(randomNumber);

    System.out.println(objRef);
    System.out.println(objRef.getData());
    System.out.println(randomNumber);
  }//end main

  //Declare the signature of an abstract class.
  public abstract int getData();

}//end class Prob04
```

5.247

The import directive

The import directive at the beginning of Listing 1 (p. 919) is required because the program requires access to the **Random** class and the **Date** class, both of which are defined in the **java.util** package.

Lazy programming practice

It would be better programming practice to provide two explicit import directives, one for the **Random** class and the other for the **Date** class. However, if you are lazy like I apparently was when I wrote this program, you can use the wildcard character *(*)* to *import* all of the classes in a package.

An *abstract* method

I'm going to begin by skipping down to the second line from the bottom in Listing 1 and explain the declaration of the *abstract* method named **getData** .

Purpose of an abstract method

The purpose of an abstract method declaration is to establish the signature of a method that must be overridden in every *(non-abstract)* subclass of the class in which the abstract method is declared.

An incomplete method

As you can see the abstract method has no body. Therefore, it is incomplete, has no behavior, and cannot be executed.

An abstract method must be overridden in a subclass in order to be useful.

Override in different ways

The same abstract method can be overridden in different ways in different subclasses. In other words, the behavior of the overridden version can be tailored to *(appropriate for)* the class in which it is overridden.

A guarantee

The existence of an abstract method in a superclass guarantees that every *(non-abstract)* subclass of that superclass will have a **concrete** *(executable)* version of a method having that same signature.

An *abstract* class

The class named **Prob04** is declared *abstract* in Listing 1 (p. 919) .

Any class can be declared abstract. The consequence of declaring a class abstract is that it is not possible to instantiate an object of the class.

Must be declared abstract...

More importantly, a class must be declared abstract if it contains one or more abstract method declarations. The idea here is that it must not be possible to instantiate objects containing incomplete *(non-executable)* methods.

The main method

As you have seen in previous modules, the driver class for every Java application must contain a method named **main** with a signature matching that shown in Listing 1 (p. 919) .

A pseudo-random number generator

I will leave it as an exercise for the student to go to the javadocs and read up on the class named **Random** , along with the class named **Date** and the method named **getTime** .

Why pseudo-random?

I refer to this as a pseudo-random number generator because the sequence will probably repeat after an extremely large number of values has been generated.

An object of the class Random

Briefly, however, the first statement in the **main** method in Listing 1 (p. 919) instantiates an object that will return a pseudo-random number each time certain methods are called on the object.

Seeding the generator

The value passed as a parameter to the **Random** constructor represents the current time and guarantees that the series of pseudo-random values returned by the methods will be different each time the program is run. This is commonly known as *seeding* the generator.

Get and save a pseudo random value

The next statement in Listing 1 (p. 919) 1 calls the **nextInt** method on the generator object to get and save the next value of type **int** in the pseudo-random sequence.

Cast to type byte

This value is cast to type **byte** , which discards all but the eight least significant bits of the **int** value. When it is stored in the variable named **randomNumber** of type **int** , the sign is extended through the most significant 24 bits and it becomes a value of type int that is guaranteed to be of relatively small magnitude.

Why cast to byte?

I cast the random value to type byte simply to cause the values that are displayed to be smaller and easier to compare visually.

Instantiate an object of type Prob04MyClass

The next statement in Listing 1 (p. 919) instantiates an object of the class named **Prob04MyClass** , passing the random value as a parameter to the constructor. At this point, I will put the explanation of the class named **Prob04** on temporary hold and explain the class named **Prob04MyClass** , which begins in Listing 2 (p. 921) .

Listing 2: Beginning of the class named Prob04MyClass.

```
  class Prob04MyClass extends Prob04{
private int data;

public Prob04MyClass(int inData){//constructor
  System.out.println("Prob04");
  System.out.println("Dick");
  data = inData;
}//end constructor
```

5.248

Extends the abstract class named Prob04

First note that the class named **Prob04MyClass** extends the abstract class named **Prob04** .

Among other things, this means that either this class must override the abstract method named **getData** that was declared in the superclass, or this class must also be declared abstract.

Does it override getData?

Seeing that this class isn't declared abstract, we can surmise at this point that it does override the abstract method named **getData** . We will see more about this later.

Beginning of the class named Prob04MyClass

The class definition in Listing 2 (p. 921) begins by declaring a private instance variable of type **int** named **data** . Note that it does not initialize the variable. Therefore, the value is automatically initialized to an **int** value of zero.

The constructor

Then Listing 2 (p. 921) defines the constructor for the class. The first two statements in the constructor cause the first two lines of text shown in Image 1 (p. 918) to be displayed on the command line screen.

Save the incoming parameter value

The last line in the constructor saves the incoming value in the instance variable named **data** , overwriting the default value of zero that it finds there.

This statement is more in keeping with the intended usage of a constructor than the first two statements. The primary purpose of a constructor is to assist in the initialization of the *state of an object* , which depends on the values stored in its variables.

Override the abstract getData method

Listing 3 (p. 922) overrides the abstract **getData** method declared in the abstract superclass named **Prob04** and inherited into the subclass named **Prob04MyClass** .

Listing 3: Override the abstract getData method.

```
  public int getData(){//overridden abstract method
 return data;
}//end getData()
```

5.249

Very simple behavior

Although the overridden version of the method simply returns a copy of the value stored in the private instance variable named **data** , it is concrete [107] and can be executed. We will see later that it is called in the **main** method of the driver class named **Prob04** in Listing 1 (p. 919) .

Override the toString method

The ultimate superclass of every class is the predefined system class named **Object** . The **Object** class defines eleven methods with default behavior, including the method named **toString** .

Listing 4 (p. 922) overrides the inherited **toString** method, overriding the default behavior of the method insofar as objects of the class named **Prob04MyClass** are concerned.

Listing 4: Override the toString method.

```
  public String toString(){//overridden method
 return "Baldwin";
}//end overloaded toString()

}//end class Prob04MyClass
```

5.250

Default behavior of the toString method

If the **toString** method had not been overridden in the **Prob04MyClass** class, calling the **toString** method on an object of the class would return a string similar to that shown in Image 2.

Image 2: Default behavior of the toString method .

```
Prob04MyClass@42e816
```

5.251

[107] http://cnx.org/content/m44205/latest/Java3008old.htm#concrete

Image 2 (p. 922) shows the default behavior of the **toString** method as defined in the **Object** class. For this program, only the six hexadecimal digits at the end would change from one run to the next.

More on the default behavior of the toString method

Furthermore, if the **toString** method had not been overridden in the **Prob04MyClass** class, the output produced by the program on the command line screen would be similar to that shown in Image 3 (p. 923) instead of that shown in Image 1 (p. 918) .

Image 3: More on the default behavior of the toString method.

```
    Prob04
Dick
Prob04MyClass@42e816
-34
-34
```

5.252

Compare to see the difference

If you compare Image 3 (p. 923) with Image 1 (p. 918) , you will see that the difference results from the fact that the overridden version of the **toString** method in Listing 4 (p. 922) returns *"Baldwin"* as a string rather than returning the default string shown in Image 2 (p. 922) .

The end of the class named Prob04MyClass

Listing 4 (p. 922) signals the end of the class definition for the class named **Prob04MyClass** . Therefore, it is time to return to the explanation of the driver class shown in Listing 1 (p. 919) .

Display information about the object

When the **Prob04MyClass** constructor returns, Listing 1 (p. 919) calls the **println** method passing a reference to the new object as a parameter.

Many overloaded *(not overridden)* versions of println

There are many overloaded versions of the **println** method, each of which requires a different type of incoming parameter or parameters.

For example, different overloaded versions of the method know how to receive incoming parameters of each of the different primitive types, convert them to characters, and display the characters on the screen.

An incoming parameter of type Object

There is also an overloaded version of the **println** method that requires an incoming parameter of type **Object** . That is the version of the method that is executed when the reference to this object is passed to the method in Listing 1 (p. 919) .

One object, several types

Recall that the reference to this object can be treated as its true type, or as the type of any superclass. Therefore, the reference can be treated as any of the following types:

- **Prob04MyClass**
- **Prob04**
- **Object**

Will satisfy type requirement...

Because it can be treated as type **Object** , it will satisfy the type requirement for the overloaded version of the **println** method that requires an incoming parameter of type **Object** .

Call the toString method

The first thing that this version of the **println** method does is to call the **toString** method on the incoming reference. Then it displays the string value returned by the **toString** method on the screen.

In this case, the overridden **toString** method returns the string *"Baldwin"* , which is what you see displayed in Image 1 (p. 918) .

Runtime polymorphism

This is a clear example of an OOP concept known as *runtime polymorphism* .

Runtime polymorphism is much too complicated to explain in this module. However, I explain it in detail in my online OOP modules and I strongly recommend that you study it there until you thoroughly understand it.

A critical concept

It is critical that you understand runtime polymorphism if you expect to go further in Java OOP.

It is almost impossible to write a useful Java application without making heavy use of runtime polymorphism, because that is the foundation of the event driven Java graphical user interface system.

Call the overridden getData method

The next statement in Listing 1 (p. 919) calls the overridden **getData** method and displays the return value.

As you saw earlier, this method returns a copy of the random value that was received and saved by the constructor for the **Prob04MyClass** class in Listing 2 (p. 921) .

Display the original random value

Finally, the last statement in the **main** method in Listing 1 (p. 919) displays the contents of the instance variable named **randomNumber** . This variable contains the random value that was passed to the constructor for the **Prob04MyClass** earlier in Listing 1 (p. 919) .

The two values must match

Therefore, the final two statements in the **main** method in Listing 1 (p. 919) display the same random value. This is shown in the command line screen output in Image 1 (p. 918) .

The program terminates

After displaying this value, the **main** method terminates causing the program to terminate.

5.3.13.5 Run the program

I encourage you to copy the code from Listing 5 (p. 926) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

5.3.13.6 Summary

You have learned about abstract methods, abstract classes, and overridden methods in this module. Very importantly, you have learned about overriding the **toString** method.

5.3.13.7 What's next?

You will learn more about indirection, array objects, and casting in the next module.

5.3.13.8 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 04 [108]
  - Part01 [109]
  - Part02 [110]

[108] http://www.youtube.com/playlist?list=PL6C202D624F8C5972
[109] http://www.youtube.com/watch?v=wReb-ZdxgwQ
[110] http://www.youtube.com/watch?v=AMe_hVVZ7CA

- Part03 [111]
- Part04 [112]

5.3.13.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Abstract Methods, Abstract Classes, and Overridden Methods
- File: Java3008.htm
- Published: 08/02/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.13.10 Complete program listings

A complete listing of the program discussed in this module is shown in Listing 5 (p. 926) below.

[111] http://www.youtube.com/watch?v=DveltVjYqhQ
[112] http://www.youtube.com/watch?v=EPwoHu3O1ww

Listing 5: Complete program listing.

```
    /*File Prob04 Copyright 2001, R.G.Baldwin
Rev 12/16/08
*******************************************************/
import java.util.*;

abstract class Prob04{
  public static void main(String[] args){
    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Prob04 objRef = new Prob04MyClass(randomNumber);
    System.out.println(objRef);
    System.out.println(objRef.getData());
    System.out.println(randomNumber);
  }//end main

  //Declare the signature of an abstract class.
  public abstract int getData();

}//end class Prob04
//=======================================================//

class Prob04MyClass extends Prob04{
  private int data;

  public Prob04MyClass(int inData){//constructor
    System.out.println("Prob04");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){//overridden abstract method
    return data;
  }//end getData()

  public String toString(){//overridden method
    return "Baldwin";
  }//end overloaded toString()

}//end class Prob04MyClass
```

5.253

-end-

5.3.14 Java3008r Review[113]

5.3.14.1 Table of Contents

5.3.14.2 Preface

This module contains review questions and answers keyed to the module titled Java3008: Abstract Methods, Abstract Classes, and Overridden Methods [114] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.14.3 Questions

5.3.14.3.1 Question 1 .

True or False? The code in Listing 1 (p. 928) produces the output shown in Image 1 (p. 929) where the two numeric values shown are random but must always match.

[113] This content is available online at <http://cnx.org/content/m45773/1.1/>.
[114] http://cnx.org/content/m44205

Source code for Java3008r.java.

```
/*File Java3008r Copyright 2013, R.G.Baldwin
Rev 02/12/13
**********************************************************/
import java.util.*;

abstract class Java3008r{
  public static void main(String[] args){
    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Java3008r objRef = new Java3008rMyClass(randomNumber);
    System.out.println(objRef);
    System.out.println(objRef.getData());
    System.out.println(randomNumber);
  }//end main

  //Declare the signature of an abstract class.
  public abstract int getData();

}//end class Java3008r
//========================================================//

class Java3008rMyClass extends Java3008r{
  private int data;

  public Java3008rMyClass(int inData){//constructor
    System.out.println("Java3008r");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){//overridden abstract method
    return data;
  }//end getData()

}//end class Java3008rMyClass
```

5.254

Possible output from the code in Listing 1.

```
    Java3008r
Dick
Baldwin
-80
-80
```

5.255

Answer 1 (p. 933)

5.3.14.3.2 Question 2

True or False? The purpose of an abstract method declaration is to establish the signature of a method that must be overridden in a *(non-abstract)* subclass of the class in which the abstract method is declared.

Answer 2 (p. 933)

5.3.14.3.3 Question 3

True or False? When an abstract method is executed, it always exhibits default behavior defined in the class in which it is declared.

Answer 3 (p. 932)

5.3.14.3.4 Question 4

True or False? An abstract method must be overridden in a subclass in order to be executed.

Answer 4 (p. 932)

5.3.14.3.5 Question 5

True or False? An abstract method can be overridden once and once only and it must be overridden in the immediate subclass of the class in which it is declared.

Answer 5 (p. 932)

5.3.14.3.6 Question 6

True or False? The existence of an abstract method in a superclass guarantees that objects instantiated from every *(non-abstract)* subclass of that superclass will have a **concrete** *(executable)* version of a method having that same signature.

Answer 6 (p. 932)

5.3.14.3.7 Question 7

True or False? Any class can be declared abstract. The consequence of declaring a class abstract is that it is not possible to instantiate an object of the class.

Answer 7 (p. 932)

5.3.14.3.8 Question 8

True or False? A class must be declared abstract if it contains two or more abstract method declarations.
Answer 8 (p. 932)

5.3.14.3.9 Question 9

True or False? If a class inherits an abstract method, either the subclass must be declared abstract, or it must provide a concrete overridden version of the inherited abstract method.
Answer 9 (p. 932)

5.3.14.3.10 Question 10

True or False? The primary purpose of a constructor is to assist in the initialization of the *state of an object* , which depends on the values stored in its variables.
Answer 10 (p. 932)

5.3.14.3.11 Question 11

True or False? The default version of the toString method is defined in the class named **Class** .
Answer 11 (p. 932)

5.3.14.3.12 Question 12

True or False? There is an overloaded version of the **println** method that requires an incoming parameter of type **Object** . When that version of the method is called, it calls the **toString** method on the incoming object reference and displays the string that is returned by the **toString** method. If the **toString** method belonging to the object has not been overridden, the default version of the **toString** method will be executed and the string that will be displayed is the string returned by that default version. The **toString** method can be overridden to cause the string that is displayed to be more appropriate for the object. The **toString** method can be overridden only once in the class hierarchy.
Answer 12 (p. 932)

5.3.14.4 Images

- Image 1 (p. 929) . Possible output from the code in Listing 1.
- Image 2 (p. 933) . Output from code in Listing 1.

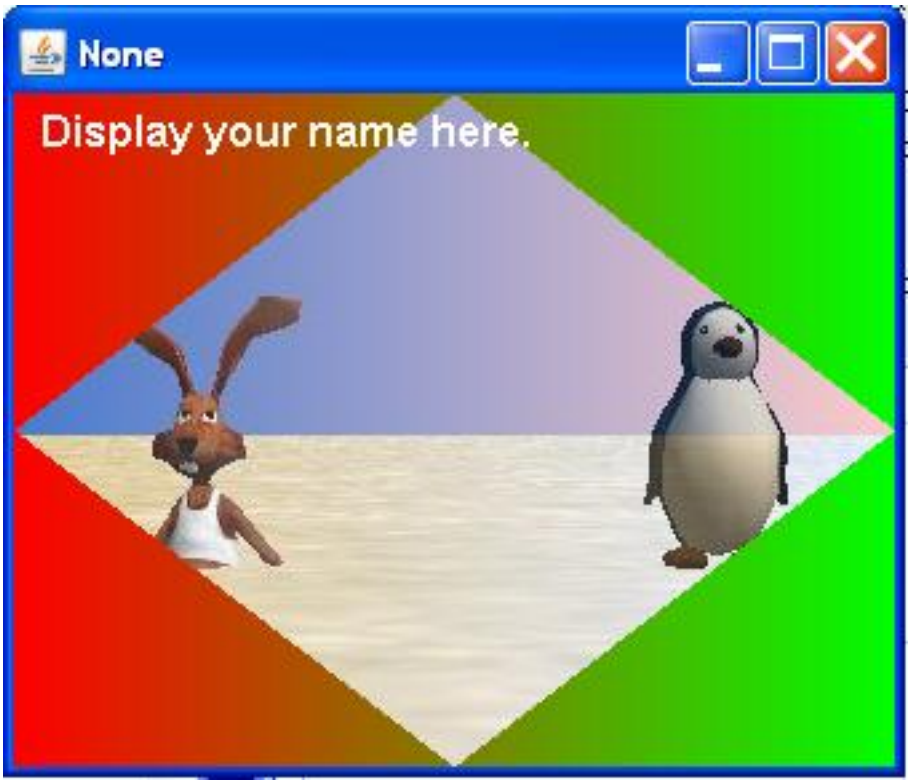
5.3.14.5 Listings

- Listing 1 (p. 928) . Source code for Java3008r.java.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.14.6 Answers

5.3.14.6.1 Answer 12

False. The **toString** method can be overridden by every class that inherits it.
Back to Question 12 (p. 930)

5.3.14.6.2 Answer 11

False. The default version of the toString method is defined in the class named **Object** .
Back to Question 11 (p. 930)

5.3.14.6.3 Answer 10

True.
Back to Question 10 (p. 930)

5.3.14.6.4 Answer 9

True.
Back to Question 9 (p. 930)

5.3.14.6.5 Answer 8

False. A class must be declared abstract if it contains one or more abstract method declarations.
Back to Question 8 (p. 930)

5.3.14.6.6 Answer 7

True.
Back to Question 7 (p. 929)

5.3.14.6.7 Answer 6

True.
Back to Question 6 (p. 929)

5.3.14.6.8 Answer 5

False. The same abstract method can be overridden in different ways in different subclasses. In other words, the behavior of the overridden version can be tailored to *(be appropriate for)* the class in which it is overridden.
Back to Question 5 (p. 929)

5.3.14.6.9 Answer 4

True.
Back to Question 4 (p. 929)

5.3.14.6.10 Answer 3

False. An abstract method has no body. Therefore, it is incomplete, has no behavior, and cannot be executed.
Back to Question 3 (p. 929)

5.3.14.6.11 Answer 2

True.

Back to Question 2 (p. 929)

5.3.14.6.12 Answer 1

False. Listing 1 (p. 928) produces the output shown in Image 2 (p. 933) except that the numeric values may vary from one run to the next. Note that the **toString** method is not overridden in Listing 1 (p. 928) .

Output from code in Listing 1.

```
    Java3008r
Dick
Java3008rMyClass@4f1d0d
27
27
```

5.256

Back to Question 1 (p. 927)

5.3.14.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3008r Review
- File: Java3008r.htm
- Published: 02/12/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.15 Java3008s Slides[115]

5.3.15.1 Table of Contents

5.3.15.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3008: Abstract Methods, Abstract Classes, and Overridden Methods [116] .

Click here [117] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.15.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3008s Slides: Abstract Methods, Abstract Classes, and Overridden Methods
- File: Java3008s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[115] This content is available online at <http://cnx.org/content/m45624/1.3/>.
[116] http://cnx.org/content/m44205
[117] http://cnx.org/content/m45624/latest/a0-Index.htm

5.3.16 Java3010: Indirection, Array Objects, and Casting[118]

5.3.16.1 Table of Contents

5.3.16.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.3.16.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.16.2.1.1 Images

5.3.16.2.1.2 Listings

5.3.16.3 Preview

The program that I will explain in this module produces no graphics and does not require the use of Ericson's media library.

OOP concepts

The program illustrates the following OOP concepts among others:

- Multiple levels of indirection
- A one-element array of type **Object**

[118] This content is available online at <http://cnx.org/content/m44206/1.5/>.

- Storing a reference to an object in an array element as type **Object**
- An anonymous object
- Passing a reference to a subclass object as type **Object**
- Downcasting an incoming object reference to access a method

Program specifications

Write a program named **Prob05** that uses the class definition shown in Listing 1 (p. 937) to produce an output similar to that shown in Image 1 (p. 936) on the command-line screen.

Image 1: Program output on command line screen.

```
    Prob05
Dick
Baldwin
-28
-28
```

5.257

A random value

Because the program generates and uses a random data value, the actual values displayed will differ from one run to the next. However, in all cases, the two values shown in Image 1 (p. 936) must match.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob05** which begins in Listing 1 (p. 937) .

5.3.16.4 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in Listing 6 (p. 943) near the end of the module.

I will begin with the driver class named **Prob05** , which begins in Listing 1 (p. 937) .

Listing 1: Beginning of the Prob05 class.

```
    import java.util.*;

class Prob05{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Object[] objRef = {new Prob05MyClassA(randomNumber)};
```

5.258

Everything in Listing 1 (p. 937) should be familiar to you except for the last statement, which I will explain shortly.

Characteristics of arrays in Java

Before explaining that statement, however, I will discuss some of the important characteristics of array objects in Java. A list of such characteristics follows in no particular order:

- All arrays in Java are one-dimensional arrays. *(Multidimensional arrays are created by creating tree structures of one-dimensional arrays.)*
- Each array in Java is encapsulated in a special type of object that I will refer to as an *array object* .
- As with all objects, an array object must be accessed using a reference to the array object.
- When the declared type of an array is one of the eight primitive types, the actual values are stored in the array elements in the array object.
- When the declared type of an array is the type of an object *(array object or ordinary object)*, references to the objects are stored in the array elements and the objects actually exist elsewhere in memory.
- As with instance variables, the elements in an array are typically initialized with the standard default values for the types involved *(zero, false, or null)* . That is not the case in this program however.
- The array that is encapsulated in an array object may have none, one, or more elements. *(Yes, it is possible for a Java array to have no elements, but that normally occurs only in special circumstances.)*

- The **length** or size of the array is established when the array object is instantiated and cannot be changed thereafter.
- Every array object contains a special property named **length** that contains the number of elements in an array. It is always possible to determine the number of elements in an array object at runtime by accessing the value of the **length** property for the array object.

A special instantiation syntax

There is a special syntax that allows for the instantiation of an array object and the initialization of the array elements in a single statement. *(I explain this in detail in my online OOP tutorial modules.)* The last statement in Listing 1 (p. 937) is an example of this syntax.

Briefly, the syntax consists of a comma separated list of element values *(expressions)* inside a pair of matching curly braces. The **length** of the array is determined by the number of values in the list. The type of the array is determined by the types of the elements in the list.

This syntax instantiates an array object of the correct **length** and populates the elements with the specified values.

A reference is returned

A reference to the array object is returned in much the same way that a constructor for an ordinary object returns a reference to the object.

As is always the case, if the reference is stored in a variable, the type of the reference must be *assignment compatible* with the type of the variable.

What is assignment compatible?

I recommend that you go to Google and search for the following keywords to learn more about this topic:

baldwin java "assignment compatible"

A one-element array

The last statement in Listing 1 (p. 937) instantiates an array object containing a one-element array. The array element is initialized with a reference to a new object of type **Prob05MyClassA** , which exists somewhere else in memory.

The value of a random number that was generated earlier in the **main** method is passed as a parameter to the constructor for the object of type **Prob05MyClassA** .

Save the reference to the array object

The reference to the array object is stored in the local reference variable named **objRef** of type **Object** . We know that the reference is *assignment compatible* with this reference variable because the **Object** type is completely generic. All non-primitive types are assignment compatible with type **Object** .

The class named Prob05MyClassA

At this point, I am going to put the explanation of the class named **Prob05** temporarily on hold and explain the class named **Prob05MyClassA** , which is shown in its entirety in Listing 2 (p. 938) .

Listing 2: The class named Prob05MyClassA.

```
   class Prob05MyClassA extends Prob05{
  private int data;

  public Prob05MyClassA(int inData){
    System.out.println("Prob05");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

}//end class Prob05MyClassA
```

5.259

Note that the class named **Prob05MyClassA** extends the class named **Prob05** , which is partially shown in Listing 1 (p. 937) .

Familiar code

All of the code in Listing 2 (p. 938) should be familiar to you because it is very similar to the code in the previous module. Therefore, no explanation of Listing 2 (p. 938) is warranted.

Save the incoming value

In summary, when the object of type **Prob05MyClassA** is instantiated, it saves the value of an incoming constructor parameter in a private instance variable.

Return the saved value

When the method named **getData** is called on a reference to the object, it returns a copy of that value.

A review

To review what I have already said, the array object that was instantiated in Listing 1 (p. 937) contains a reference to this object of type **Prob05MyClassA** in the only element of the one-element array.

The reference to the array object is stored in the reference variable named **objRef** .

Indirection at work

At this point, **objRef** contains a reference to an array object, one element of which contains a reference to an ordinary object, which is located somewhere else in memory. This is indirection.

The next statement in the main method

Returning now to the **main** method that began in Listing 1 (p. 937) , Listing 3 (p. 939) shows the next statement in the **main** method following the last statement in Listing 1 (p. 937) .

Listing 3: The next statement in the main method.

```
System.out.println(
  new Prob05MyClassB().getDataFromObj(objRef[0]));
```

5.260

What is an anonymous object?

An anonymous object is an object whose reference is not saved in a named reference variable.

Instantiate an anonymous object

Consider the parameter list of the **println** method shown in Listing 3 (p. 939) . A new object of the **Prob05MyClassB** class is instantiated in the parameter list. However, the reference to that object is not saved in a named reference variable. Instead, that reference is used to immediately call the method named **getDataFromObj** that belongs to the anonymous object.

The parameter that is passed...

Now consider the parameter that is passed to the method named **getDataFromObj** . The expression inside that parameter list extracts the contents of the zeroth element in the array object that is referred to by the contents of the variable named **objRef** .

And those contents are...

That element contains a reference to an object of the class **Prob05MyClassA** *(see Listing 1 (p. 937))* .

Therefore, a reference to an object of type **Prob05MyClassA** is passed as a parameter to the method named **getDataFromObj** .

The class named Prob05MyClassB

It is time to take a look at the class in which the **getDataFromObj** method is defined.

The class named **Prob05MyClassB** is shown in its entirety in Listing 4 (p. 940) .

Listing 4: The class named Prob05MyClassB.

```
    class Prob05MyClassB{

  Prob05MyClassB(){
    System.out.println("Baldwin");
  }//end constructor

  public int getDataFromObj(Object refToObj){
    return ((Prob05MyClassA)refToObj).getData();
  }//end getDataFromObj()

}//end class Prob05MyClassB
```

5.261

Extends the Object class

Note that this class does not extend the class named **Prob05** . In fact, it doesn't explicitly extend any class. This means that it extends the class named **Object** by default because every class is a subclass of the class named **Object** .

The constructor

The constructor for this class is inconsequential. It simply displays my last name when the object is instantiated, producing part of the output text shown in Image 1 (p. 936) .

The getDataFromObj method

The interesting part of Listing 4 (p. 940) is the definition of the method named **getDataFromObj** .

As we saw before, this method receives a reference to an object of type **Prob05MyClassA** *(see Listing 3 (p. 939))* . However, this reference is not received as the true type of the object. Instead, it is received as type **Object** , which is the ultimate superclass of the class named **Prob05MyClassA** .

The objective of the method

The objective is to call the method named **getData** on the incoming reference. However, the **Object** class doesn't know anything about a method named **getData** because the **Object** class neither defines nor inherits a method having that signature. Instead, the **getData** method is defined in the class named **Prob05MyClassA** , which is the true type of the object.

A cast is required

Therefore, it is necessary to convert the type of the reference back to its true type using a cast operator before that reference can be used to call the method named **getData** . *(The cast operator is shown in Listing 4 (p. 940) .)*

The returned value

The **getData** method returns a copy of the value that was passed as a constructor parameter when the object was instantiated. *(See Listing 2 (p. 938) .)* Recall that the value was the original random value. *(See Listing 1 (p. 937) .)*

Referring back to Listing 4 (p. 940) , that is the value that is returned from the call to the **getDataFromObj** method in Listing 3 (p. 939) , which cause the value to be displayed as the first numeric value in Image 1 (p. 936) .

The end of the main method

Returning once more to the **main** method and picking up where we left off in Listing 3 (p. 939) , Listing 5 (p. 941) shows the final statement in the **main** method.

Listing 5: The end of the main method.

```
        System.out.println(randomNumber);

  }//end main
}//end class Prob05
```

5.262

This statement simply displays the original random value that was passed to the constructor for the **Prob05MyClassA** in Listing 1 (p. 937) . This statement displays the second numeric value shown as the last line of text in Image 1 (p. 936) .

The end of the program

At this point, the **main** method terminates causing the program to terminate.

5.3.16.5 Run the program

I encourage you to copy the code from Listing 6 (p. 943) , compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

5.3.16.6 Summary

You learned about the following OOP concepts, among others in this module.

- Multiple levels of indirection
- A one-element array of type **Object**
- Storing a reference to an object in an array element as type **Object**
- An anonymous object
- Passing a reference to a subclass object as type **Object**
- Downcasting an incoming object reference to access a method

5.3.16.7 What's next?

You will learn how to use nested loops to process pixels on a row and column basis in the next module.

5.3.16.8 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 05 [119]
  - Part01 [120]
  - Part02 [121]
  - Part03 [122]

[119] http://www.youtube.com/playlist?list=PL13622F7BA83F110C
[120] http://www.youtube.com/watch?v=Ow_XzlSrmsw
[121] http://www.youtube.com/watch?v=UiT_ZYtNqWo
[122] http://www.youtube.com/watch?v=fCiMM4ps3o4

5.3.16.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Indirection, Array Objects, and Casting
- File: Java3010.htm
- Published: 08/02/12
- Revised: 01/01/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.16.10 Complete program listings

A complete listing of the program discussed in this module is shown in Listing 6 (p. 943) below.

Listing 6: Complete program listing.

```
    /*File Prob05 Copyright 2001, R.G.Baldwin
Rev 12/16/08
**********************************************************/
import java.util.*;

class Prob05{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Object[] objRef = {new Prob05MyClassA(randomNumber)};

    System.out.println(
          new Prob05MyClassB().getDataFromObj(objRef[0]));

    System.out.println(randomNumber);

  }//end main
}//end class Prob05
//=======================================================//

class Prob05MyClassA extends Prob05{
  private int data;

  public Prob05MyClassA(int inData){
    System.out.println("Prob05");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

}//end class Prob05MyClassA
//=======================================================//

class Prob05MyClassB{

  Prob05MyClassB(){
    System.out.println("Baldwin");
  }//end constructor

  public int getDataFromObj(Object refToObj){
    return ((Prob05MyClassA)refToObj).getData();
  }//end getDataFromObj()

}//end class Prob05MyClassB
```

-end-

5.3.17 Java3010r Review[123]

5.3.17.1 Table of Contents

5.3.17.2 Preface

This module contains review questions and answers keyed to the module titled Java3010: Indirection, Array Objects, and Casting [124] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.17.3 Questions

5.3.17.3.1 Question 1 .

True or False? The code shown in Listing 1 (p. 946) produces the output shown in Image 1 (p. 947) where the numeric values are random and vary from one run to the next.

[123] This content is available online at <http://cnx.org/content/m45774/1.1/>.
[124] http://cnx.org/content/m44206

Listing 1. Question 1.

```
/*File Java3010ra Copyright 2013, R.G.Baldwin
Rev 02/14/13
**********************************************************/
import java.util.*;

class Java3010ra{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Object[] objRef = [new Java3010raMyClassA(randomNumber)];

    System.out.println(
          new Java3010raMyClassB().getDataFromObj(objRef[0]));

    System.out.println(randomNumber);

  }//end main
}//end class Java3010ra
//========================================================//

class Java3010raMyClassA extends Java3010ra{
  private int data;

  public Java3010raMyClassA(int inData){
    System.out.println("Java3010ra");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

}//end class Java3010raMyClassA
//========================================================//

class Java3010raMyClassB{

  Java3010raMyClassB(){
    System.out.println("Baldwin");
  }//end constructor

  public int getDataFromObj(Object refToObj){
    return ((Java3010raMyClassA)refToObj).getData();
  }//end getDataFromObj()

}//end class Java3010raMyClassB
```

Image 1. Question 1.

```
    Java3010ra
Dick
Baldwin
-37
-37
```

5.265

Answer 1 (p. 953)

5.3.17.3.2 Question 2

True or False? All array objects in Java contain one-dimensional array structures.
Answer 2 (p. 953)

5.3.17.3.3 Question 3

True or False? Each array in Java is encapsulated in an array object. An array object must be accessed using a reference to the array object.
Answer 3 (p. 953)

5.3.17.3.4 Question 4

True or False? When the declared type of an array is one of the eight primitive types, the actual values are stored in the array elements in the array object.
Answer 4 (p. 953)

5.3.17.3.5 Question 5

True or False? When the declared type of an array is the type of an object *(array object or ordinary object)* , those objects are stored in the array elements.
Answer 5 (p. 953)

5.3.17.3.6 Question 6

True or False? Unless code is written to do otherwise, the elements in a new array object are initialized with the standard default values for the types involved *(zero, true, or null)* .
Answer 6 (p. 953)

5.3.17.3.7 Question 7

True or False? The array structure that is encapsulated in an array object must have one, or more elements.
Answer 7 (p. 953)

5.3.17.3.8 Question 8

True or False? The **length** or size of the array is established when the array object is instantiated and cannot be changed thereafter.

Answer 8 (p. 953)

5.3.17.3.9 Question 9

True or False? Every array object contains a special property named **size** that contains the number of elements in an array. It is always possible to determine the number of elements in an array object at runtime by accessing the value of the **size** property for the array object.

Answer 9 (p. 953)

5.3.17.3.10 Question 10

True or False? There is a special syntax that allows for the instantiation of an array object and the initialization of the array elements in a single statement.

Answer 10 (p. 952)

5.3.17.3.11 Question 11

True or False? An anonymous class is an object whose reference is not saved in a named reference variable.

Answer 11 (p. 952)

5.3.17.3.12 Question 12

True or False? Every class that doesn't explicitly extend another class automatically extends the class named **Class** .

Answer 12 (p. 952)

5.3.17.3.13 Question 13

True or False? The code shown in Listing 3 (p. 949) produces the output shown in Image 3 (p. 950) where the numeric values are random and vary from one run to the next.

Listing 3. Question 13.

```
/*File Java3010rb Copyright 2013, R.G.Baldwin
Rev 02/14/13
**********************************************************/
import java.util.*;

class Java3010rb{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Object[] objRef = {new Java3010rbMyClassA(randomNumber)};

    System.out.println(
          new Java3010rbMyClassB().getDataFromObj(objRef[0]));

    System.out.println(randomNumber);

  }//end main
}//end class Java3010rb
//========================================================//

class Java3010rbMyClassA extends Java3010rb{
  private int data;

  public Java3010rbMyClassA(int inData){
    System.out.println("Java3010rb");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

}//end class Java3010rbMyClassA
//========================================================//

class Java3010rbMyClassB{

  Java3010rbMyClassB(){
    System.out.println("Baldwin");
  }//end constructor

  public int getDataFromObj(Object refToObj){
    return refToObj.getData();
  }//end getDataFromObj()

}//end class Java3010rbMyClassB
```

Image 3. Question 13.

```
    Java3010rb
Dick
Baldwin
120
120
```

5.267

Answer 13 (p. 952)

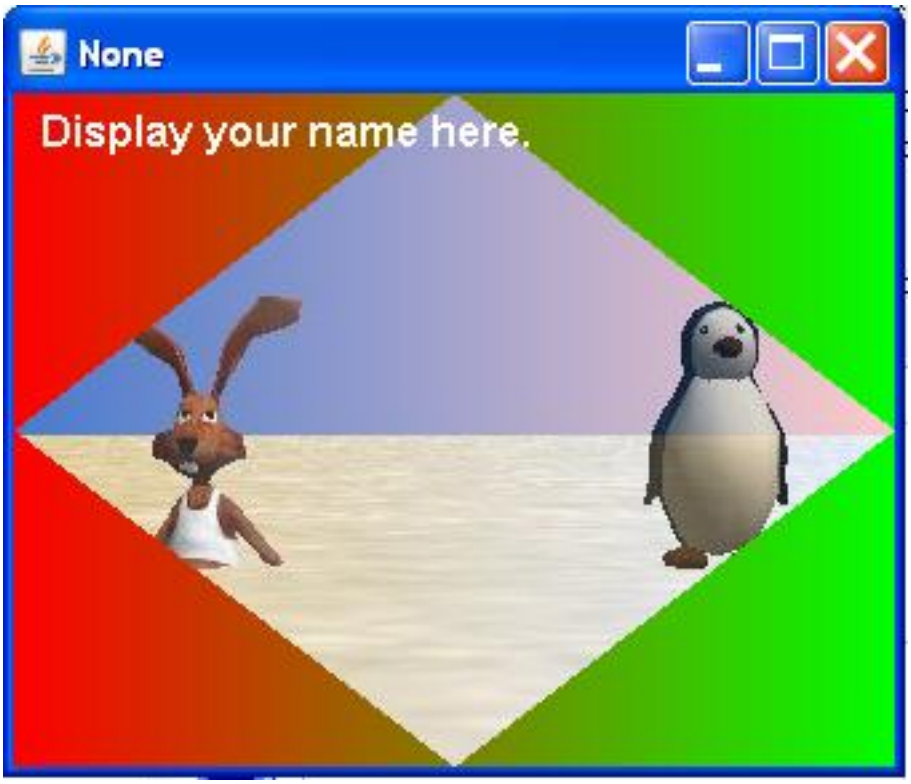
5.3.17.4 Images

5.3.17.5 Listings

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.17.6 Answers

5.3.17.6.1 Answer 13

False: The code shown in Listing 3 (p. 949) produces the compiler error shown in Image 4 (p. 952) . A cast is required to fix the problem.

Image 4. Answer 13.

```
   Java3010rb.java:47: error: cannot find symbol
   return (refToObj).getData();
                     ^
  symbol:   method getData()
  location: variable refToObj of type Object
1 error
```

5.268

Back to Question 13 (p. 948)

5.3.17.6.2 Answer 12

False. Every class that doesn't explicitly extend another class automatically extends the class named **Object** .

Back to Question 12 (p. 948)

5.3.17.6.3 Answer 11

False. An anonymous object is an object whose reference is not saved in a named reference variable. An anonymous **class** is something entirely different.

Back to Question 11 (p. 948)

5.3.17.6.4 Answer 10

True. Listing 2 (p. 952) shows an example of this syntax. *(Note the use of the curly brackets in Listing 2 (p. 952) as opposed to the use of square brackets in Listing 1 (p. 946) .)*

Listing 2. Answer 10.

```
Object[] objRef = {new Java3010raMyClassA(randomNumber)};
```

5.269

Back to Question 10 (p. 948)

5.3.17.6.5 Answer 9

False. Every array object contains a special property named **length** that contains the number of elements in an array. It is always possible to determine the number of elements in an array object at runtime by accessing the value of the **length** property for the array object.

Back to Question 9 (p. 948)

5.3.17.6.6 Answer 8

True.

Back to Question 8 (p. 948)

5.3.17.6.7 Answer 7

False. The array structure that is encapsulated in an array object may have none, one, or more elements.

Back to Question 7 (p. 947)

5.3.17.6.8 Answer 6

False. Unless code is written to do otherwise, the elements in a new array object are initialized with the standard default values for the types involved *(zero,* ***false*** *, or null)* .

Back to Question 6 (p. 947)

5.3.17.6.9 Answer 5

False. When the declared type of an array is the type of an object *(array object or ordinary object)*, references to the objects are stored in the array elements and the objects actually exist elsewhere in memory.

Back to Question 5 (p. 947)

5.3.17.6.10 Answer 4

True.

Back to Question 4 (p. 947)

5.3.17.6.11 Answer 3

True.

Back to Question 3 (p. 947)

5.3.17.6.12 Answer 2

True. Multidimensional arrays are created by creating tree structures of one-dimensional array objects.

Back to Question 2 (p. 947)

5.3.17.6.13 Answer 1

False. The program produces the compiler error shown in Image 2 (p. 954) .

Image2. Answer 1.

```
    Java3010ra.java:12: error: illegal start of expression
    Object[] objRef = [new Java3010raMyClassA(randomNumber)];
                      ^
Java3010ra.java:12: error: ';' expected
    Object[] objRef = [new Java3010raMyClassA(randomNumber)];
```

5.270

Back to Question 1 (p. 945)

5.3.17.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3010r Review
- File: Java3010.htm
- Published: 02/14/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.18 Java3010s Slides[125]

5.3.18.1 Table of Contents

[125] This content is available online at <http://cnx.org/content/m45625/1.2/>.

5.3.18.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3010: Indirection, Array Objects, and Casting [126] .

Click here [127] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.18.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3010s Slides
- File: Java3010s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.19 Java3012: Using Nested Loops to Process Pixels[128]

5.3.19.1 Table of Contents

[126] http://cnx.org/content/m44206
[127] http://cnx.org/content/m45625/latest/a0-Index.htm
[128] This content is available online at <http://cnx.org/content/m44207/1.6/>.

5.3.19.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [129] .

5.3.19.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.19.2.1.1 Images

5.3.19.2.1.2 Listings

5.3.19.3 Preview

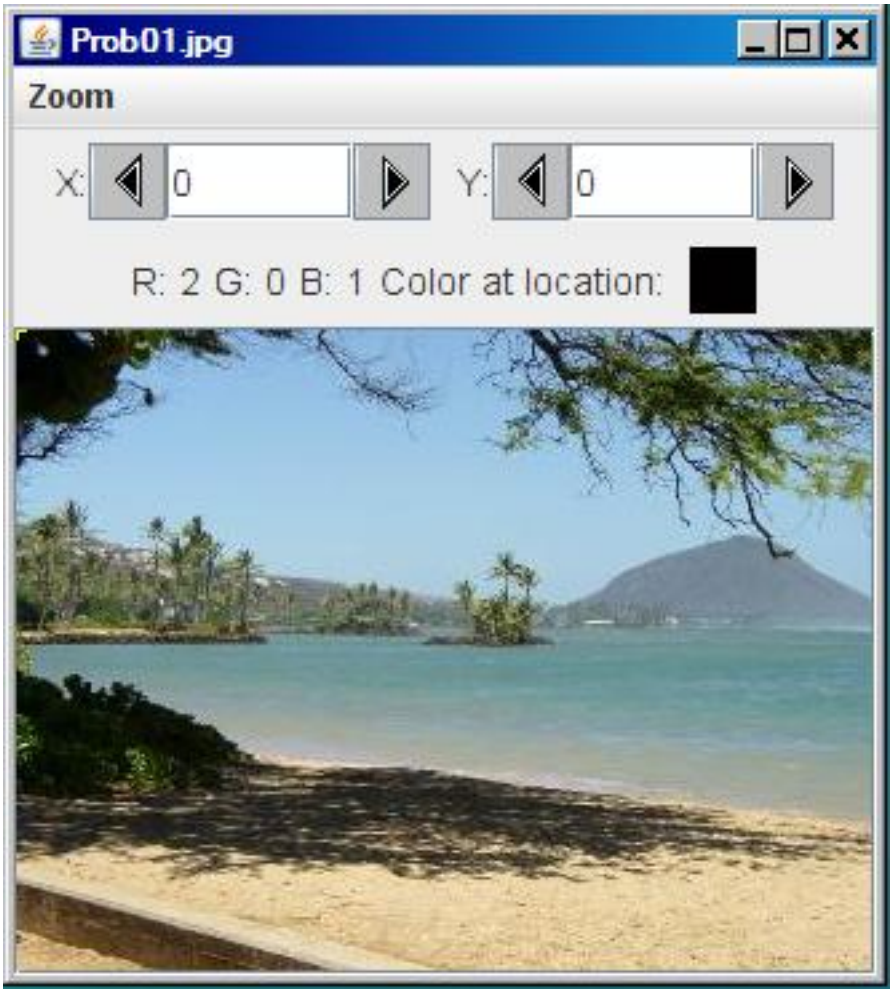
In this module, you will learn how to use nested **for** loops to process pixels on a row and column basis.

Program specifications

Write a program named **Prob01** that uses the class definition shown in Listing 1 (p. 959) along with Ericson's media library and the image file named **Prob01.jpg** to produce the graphic output images shown in Image 1 (p. 957) and Image 2 (p. 958) .

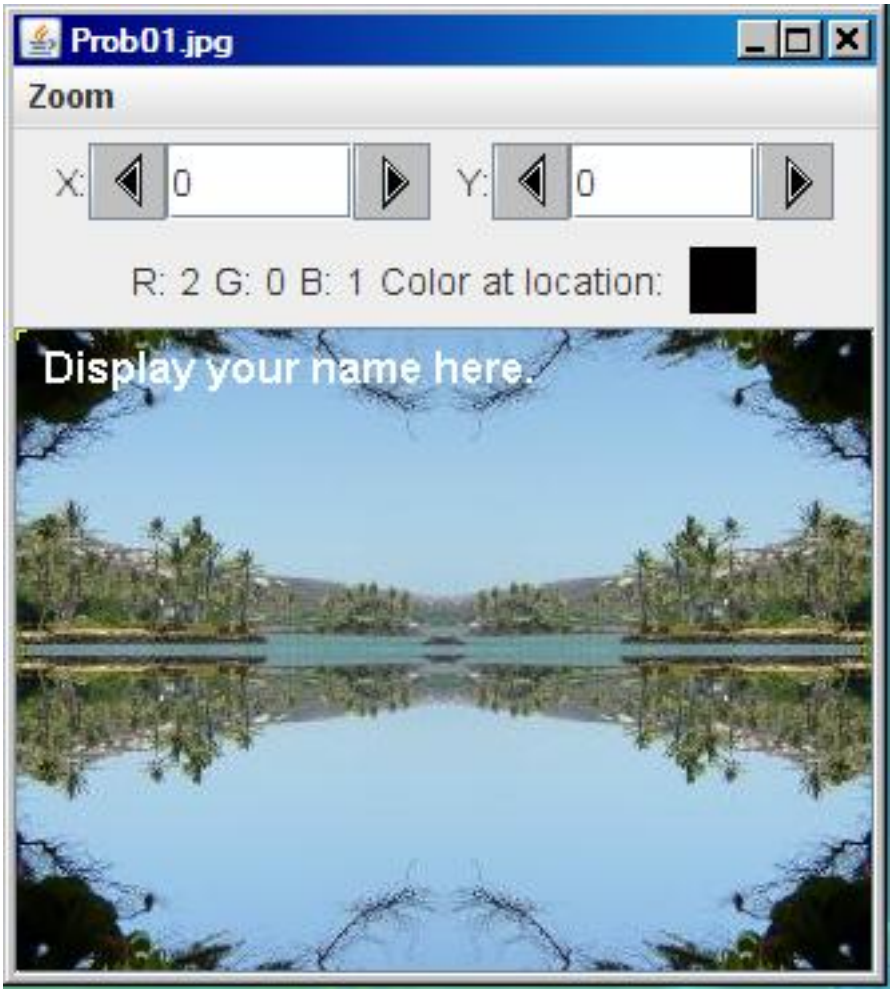
[129] http://cnx.org/content/m44148/latest/

Image 1: Raw image.



5.271

Image 2: Modified image.



5.272

Define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob01** given in Listing 1 (p. 959) .

Required text output

In addition to the two output images mentioned above, your program must display your name and the other line of text shown in Image 3 on the command-line screen.

Image 3: Required text output.

```
    Display your name here.
Picture, filename Prob01.jpg height 240 width 320
```

5.273

5.3.19.3.1 General background information

This program mirrors an image in such a way that the image in each quadrant is a mirror image of the image in the two adjacent quadrants as shown in Image 2 (p. 958) .

The top left quadrant is mirrored into the top right quadrant, and then the top half is mirrored into the bottom half.

Major evaluation areas

In order to successfully write this program, the student must be able to:

- Examine the input and output images and determine how the input image has been modified to produce the output image.
- Manipulate the individual pixels in the image to perform the required modifications.

5.3.19.4 Discussion and sample code

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 8 (p. 965) near the end of the module.

The driver class named Prob01

The driver class containing the **main** method is shown in Listing 1 (p. 959) .

Listing 1: The driver class named Prob01.

```
    public class Prob01{
  public static void main(String[] args){
    Picture pic = new Prob01Runner().run();
    System.out.println(pic);
  }//end main method
}//end class Prob01
```

5.274

There is nothing in Listing 1 (p. 959) that I haven't explained in earlier modules.

The **println** statement in Listing 1 (p. 959) causes the second line of text to be displayed in Image 3 (p. 959) .

The constructor for the class named Prob01Runner

The constructor for the class named **Prob01Runner** is shown in Listing 2 (p. 960) .

Listing 2: The constructor for the class named Prob01Runner .

```
 class Prob01Runner{
public Prob01Runner(){
  System.out.println("Display your name here.");
}//end constructor
```

5.275

The code in Listing 2 (p. 960) simply causes the first line of text in Image 3 (p. 959) to be displayed on the command line screen.

Beginning of the method named run

The code in the driver class in Listing 1 (p. 959) instantiates a new object of the **Prob01Runner** class and immediately calls the **run** method belonging to that object. The **run** method begins in Listing 3 (p. 960) .

Listing 3: Beginning of the method named run.

```
  public Picture run(){
 Picture pix = new Picture("Prob01.jpg");

 //Display the input picture.
 pix.explore();

 //Call the mirrorUpperQuads method to modify the top
 // half of the picture.
 pix = mirrorUpperQuads(pix);
```

5.276

A new Picture object

Listing 3 (p. 960) instantiates a new **Picture** object from an image file and saves a reference to that object in the local variable named **pix** .

Display the Picture object

Then Listing 3 (p. 960) calls the **explore** method on the reference producing the output image shown in Image 1 (p. 957) .

Modify top half of the picture

Finally, Listing 3 (p. 960) calls the method named **mirrorUpperQuads** to mirror the upper-left quadrant of the picture into the upper-right quadrant. A copy of a reference to the picture object is passed to the method and the value returned by the method is saved in the variable named **pix** . (*I will have more to say about this later.)*

Put the explanation of the run method on hold

I will put the explanation of the **run** method on hold temporarily and explain the method named **mirrorUpperQuads** .

Beginning of the mirrorUpperQuads method

The beginning of the **mirrorUpperQuads** method is shown in Listing 4 (p. 961) .

Listing 4: Beginning of the mirrorUpperQuads method.

```
  private Picture mirrorUpperQuads(Picture pix){
 Pixel leftPixel = null;
 Pixel rightPixel = null;

 int midpoint = pix.getWidth()/2;
 int width = pix.getWidth();
```

5.277

Note that the method receives a copy of a reference to the picture.

Declare working variables

The code in Listing 4 (p. 961) begins by declaring a pair of local working variables of type **Pixel** . These variables will be used to hold information about individual pixels.

Compute width and midpoint of the image

Then Listing 4 (p. 961) computes and saves the width and the horizontal midpoint of the image.

Mirror pixel colors around the midpoint

Listing 5 (p. 961) uses a pair of nested **for** loops to copy the pixel colors on the left of the midpoint to corresponding mirror-image pixels on the right side of the midpoint.

Listing 5: Mirror pixel colors around the midpoint.

```
    for(int row = 0;row < pix.getHeight()/2;row++){
   for(int col = 0;col < midpoint;col++){
     leftPixel = pix.getPixel(col,row);
     rightPixel = pix.getPixel(width-1-col,row);
     rightPixel.setColor(leftPixel.getColor());
   }//end inner loop
  }//end outer loop

  return pix;
}//end mirrorUpperQuads
```

5.278

Iterate on rows and columns

The outer loop in Listing 5 (p. 961) iterates down through each of the rows in the top half of the image.

The inner loop iterates across the left half of each row, copying the color of the pixels from the left half to the corresponding mirror-image pixels on the right half.

Return a reference to the modified object

Finally, Listing 5 (p. 961) returns a reference to the modified **Picture** object. The reference is assigned to the variable named **pix** in Listing 3 (p. 960) .

Superfluous but self-documenting code

Returning and storing a reference to the modified picture is superfluous and unnecessary. The code in Listing 3 (p. 960) already has a reference to the picture and that reference doesn't change just because the object to which it refers is modified.

However, I prefer this programming style because I consider it to be more self-documenting.

Remainder of the run method

Returning now to the **run** method, Listing 6 (p. 962) calls the method named **mirrorHoriz** to mirror the top half of the image into the bottom half. *(I will explain the **mirrorHoriz** method shortly.)*

Listing 6: Remainder of the run method.

```
        //Mirror the top half into the bottom half.
    pix = mirrorHoriz(pix);

    //Add your name and display the output picture.
    pix.addMessage("Display your name here.",10,20);

    pix.explore();

    return pix;

  }//end run
```

5.279

Display text on the image

Then Listing 6 (p. 962) calls the **addMessage** method on the reference to the picture to place the text near the upper-left corner as shown in Image 2 (p. 958) .

Display the modified image

After that, Listing 6 (p. 962) calls the **explore** method to display the modified image as shown in Image 2 (p. 958) .

Return a reference to the modified picture

Finally, Listing 6 (p. 962) returns the reference to the modified picture, which is saved in the variable named **pic** in Listing 1 (p. 959) .

As mentioned earlier, the variable named **pic** is passed to the **println** method in Listing 1 (p. 959) , causing the second line of text shown in Image 3 (p. 959) to be displayed on the command line screen.

The method named mirrorHoriz

Listing 7 (p. 963) shows the method named **mirrorHoriz** in its entirety. This method mirrors the top half of the picture into the bottom half.

Listing 7: The method named mirrorHoriz.

```
  private Picture mirrorHoriz(Picture pix){
Pixel topPixel = null;
Pixel bottomPixel = null;

int midpoint = pix.getHeight()/2;
int height = pix.getHeight();

for(int col = 0;col < pix.getWidth();col++){
  for(int row = 0;row < midpoint;row++){
    topPixel = pix.getPixel(col,row);
    bottomPixel =
              pix.getPixel(col,height-1-row);
    bottomPixel.setColor(topPixel.getColor());
  }//end inner loop
}//end outer loop

return pix;
}//end mirrorHoriz
//-----------------------------------------------------//

}//end class Prob01Runner
```

5.280

Very similar to an earlier method

This method is very similar to the method named **mirrorUpperQuads** that I explained in Listing 4 (p. 961) and Listing 5 (p. 961) . If you understood that explanation, you should have no difficulty understanding the code in Listing 7 (p. 963) without further explanation.

End of Prob01Runner class

Listing 7 (p. 963) also signals the end of the class named **Prob01Runner** .

5.3.19.4.1 Run the program

I encourage you to copy the code from Listing 8 (p. 965) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click here [130] to download the required input image file.

5.3.19.5 Summary

In this module, you learned how to use nested **for** loops to process pixels on a row and column basis.

5.3.19.6 What's next?

You will learn to crop, flip, and combine pictures in the next module.

[130] http://cnx.org/content/m44207/latest/Prob01.jpg

5.3.19.7 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 06 [131]
  - Part01 [132]
  - Part02 [133]

5.3.19.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Using Nested Loops to Process Pixels
- File: Java3012.htm
- Published: 07/31/12
- Revised: 02/14/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.19.9 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 8 (p. 965) below.

[131] http://www.youtube.com/playlist?list=PLB8C60363CB918BAA
[132] http://www.youtube.com/watch?v=JOhc503IPj8
[133] http://www.youtube.com/watch?v=vQBdVdqAxq4

Listing 8: Complete program listing.

```
    /*File Prob01 Copyright 2008 R.G.Baldwin
Revised 12/16/08
*******************************************************/

public class Prob01{
  public static void main(String[] args){
    Picture pic = new Prob01Runner().run();
    System.out.println(pic);
  }//end main method
}//end class Prob01
//=====================================================//

class Prob01Runner{
  public Prob01Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//
  public Picture run(){
    Picture pix = new Picture("Prob01.jpg");
    //Display the input picture.
    pix.explore();

    //Call the mirrorUpperQuads method to modify the top
    // half of the picture.
    pix = mirrorUpperQuads(pix);
    //Mirror the top half into the bottom half.
    pix = mirrorHoriz(pix);
    //Add your name and display the output picture.
    pix.addMessage("Display your name here.",10,20);
    pix.explore();
    return pix;

  }//end run
  //-----------------------------------------------------//

  //This method mirrors the upper-left quadrant of a
  // picture into the upper-right quadrant.
  private Picture mirrorUpperQuads(Picture pix){
    Pixel leftPixel = null;
    Pixel rightPixel = null;
    int midpoint = pix.getWidth()/2;
    int width = pix.getWidth();
    for(int row = 0;row < pix.getHeight()/2;row++){
      for(int col = 0;col < midpoint;col++){
        leftPixel = pix.getPixel(col,row);
        rightPixel =
                    pix.getPixel(width-1-col,row);
        rightPixel.setColor(leftPixel.getColor());
      }//end inner loop
    }//end outer loop
```



```
    return pix;
  }//end mirrorUpperQuads
  //-----------------------------------------------------//
```

-end-

5.3.20 Java3012r Review[134]

5.3.20.1 Table of Contents

5.3.20.2 Preface

This module contains review questions and answers keyed to the module titled Java3012: Using Nested Loops to Process Pixels [135] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.20.3 Questions

5.3.20.3.1 Question 1 .

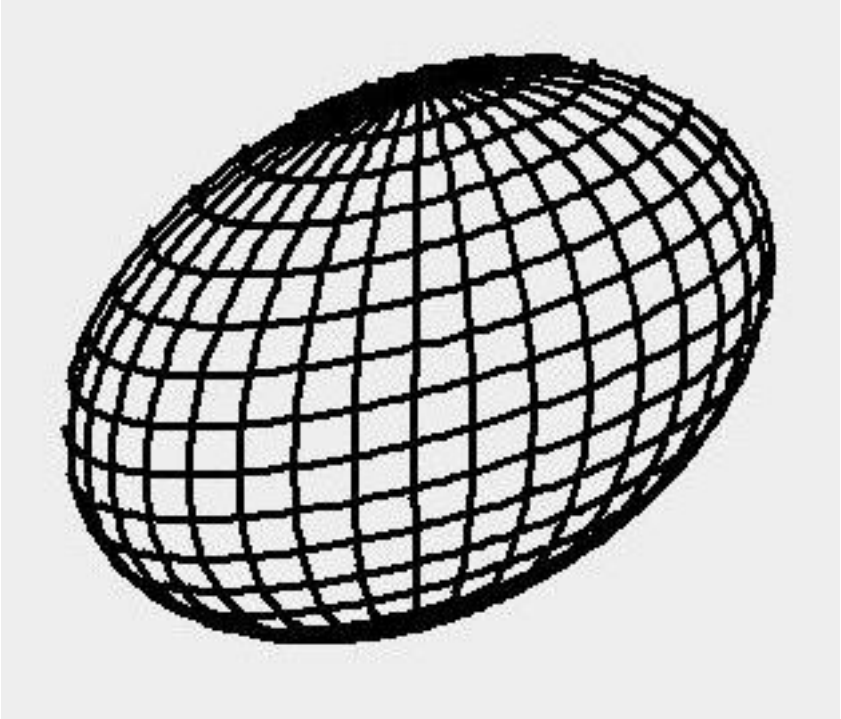
True or False? The code shown in Listing 1 (p. 968) will transform the image shown in Image 1 (p. 969) into the image shown in Image 2 (p. 970) .

[134] This content is available online at <http://cnx.org/content/m45775/1.1/>.
[135] http://cnx.org/content/m44207

Listing 1. Question 1.

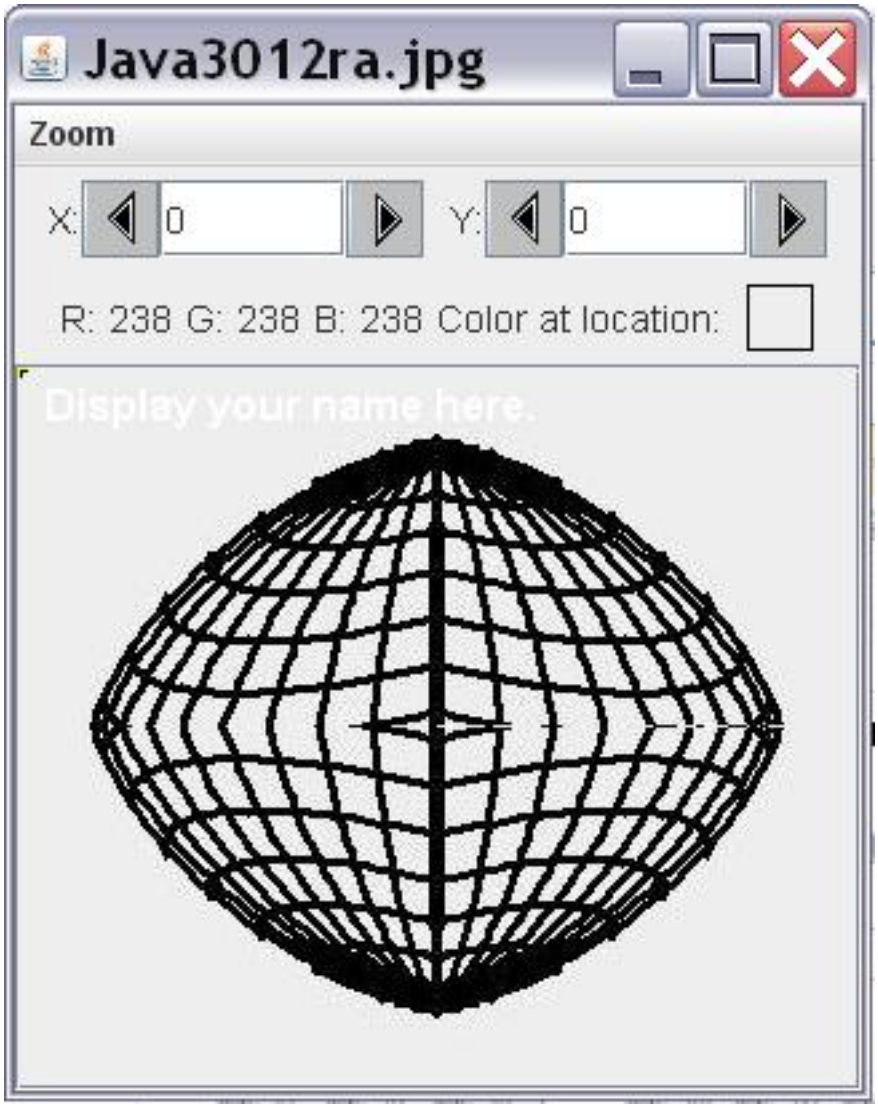
```
/*File Java3012ra Copyright 2008 R.G.Baldwin
Revised 02/14/13
*******************************************************/

public class Java3012ra{
  public static void main(String[] args){
    new Java3012raRunner().run();
  }//end main method
}//end class Java3012ra
//=======================================================//

class Java3012raRunner{
  public Java3012raRunner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//
  public void run(){
    Picture pix = new Picture("Java3012ra.jpg");

    //Call the mirrorUpperQuads method to modify the top
    // half of the picture.
    pix = mirrorUpperQuads(pix);
    //Mirror the top half into the bottom half.
    pix = mirrorHoriz(pix);
    //Add your name and display the output picture.
    pix.addMessage("Display your name here.",10,20);
    pix.explore();

  }//end run
  //-----------------------------------------------------//

  //This method mirrors the upper-left quadrant of a
  // picture into the upper-right quadrant.
  private Picture mirrorUpperQuads(Picture pix){
    Pixel leftPixel = null;
    Pixel rightPixel = null;
    int midpoint = pix.getWidth()/2;
    int width = pix.getWidth();
    for(int row = 0;row < pix.getHeight()/2;row++){
      for(int col = 0;col < midpoint;col++){
        leftPixel = pix.getPixel(col,row);
        rightPixel =
                    pix.getPixel(width-1-col,row);
        rightPixel.setColor(leftPixel.getColor());
      }//end inner loop
    }//end outer loop

    return pix;
  }//end mirrorUpperQuads
  //-----------------------------------------------------//

  //This method mirrors the top half of a picture into
  // the bottom half.
  private Picture mirrorHoriz(Picture pix){
```

Image 1. Image from the file named Java3012ra.jpg.



5.283

Image 2. Possible output produced by the program in Listing 1.



5.284

Answer 1 (p. 974)

5.3.20.3.2 Question 2

True or False? The code in Listing 2 (p. 971) instantiates an object of an anonymous class.

Listing 2. Question 2.

```
new Java3012raRunner().run();
```

5.285

Answer 2 (p. 974)

5.3.20.3.3 Question 3

True or False? The code in Listing 2 (p. 971) instantiates an anonymous object..
Answer 3 (p. 973)

5.3.20.3.4 Question 4

True or False? The statement shown in Listing 3 (p. 971) will return a reference to a **Pixel** object that represents a physical pixel located at a horizontal coordinate of **col** and a vertical coordinate of **row** .

Listing 3. Question 4.

```
leftPixel = pix.getPixels(col,row);
```

5.286

Answer 4 (p. 973)

5.3.20.4 Images

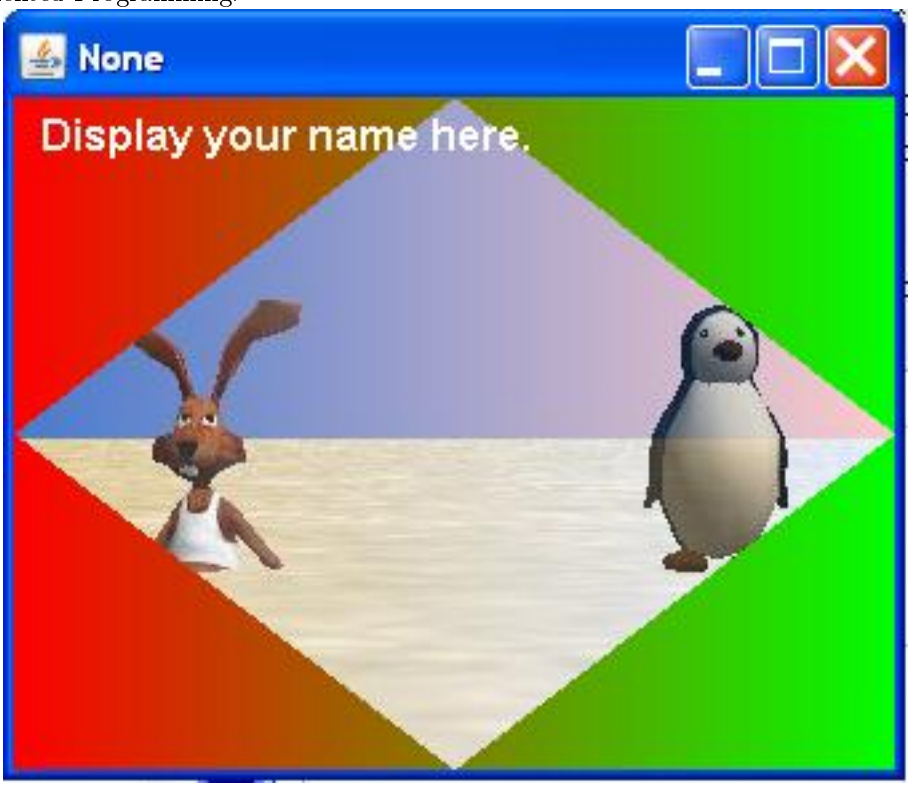
- Image 1 (p. 969) . Image from the file named Java3012ra.jpg.
- Image 2 (p. 970) . Possible output produced by the program in Listing 1.

5.3.20.5 Listings

- Listing 1 (p. 968) . Question 1.
- Listing 2 (p. 971) . Question 2.
- Listing 3 (p. 971) . Question 4.
- Listing 4 (p. 973) . Answer 4.

What is the meaning of the following two images?
This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.20.6 Answers

5.3.20.6.1 Answer 4

False. The code in Listing 3 (p. 971) has the wrong spelling for the method that returns the **Pixel** object. The correct spelling is shown in Listing 4 (p. 973) . You learned about the spelling *(with no parameters)* shown in Listing 3 (p. 971) in an earlier module.

Listing 4. Answer 4.

```
leftPixel = pix.getPixel(col,row);
```

5.287

Back to Question 4 (p. 971)

5.3.20.6.2 Answer 3

True. The object instantiated from the class named **Java3012raRunner** is an anonymous object because its reference is not saved in a named reference variable in the current scope. Anonymous classes and

anonymous objects are entirely different topics.

Back to Question 3 (p. 971)

5.3.20.6.3 Answer 2

False. A discussion of anonymous classes would be a somewhat advanced topic. That topic is not explained in the module named Java3012: Using Nested Loops to Process Pixels [136] .

Back to Question 2 (p. 970)

5.3.20.6.4 Answer 1

True.

Back to Question 1 (p. 967)

5.3.20.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3012r Review
- File: Java3012r.htm
- Published: 02/14/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.21 Java3012s Slides[137]

5.3.21.1 Table of Contents

[136] http://cnx.org/content/m44207

[137] This content is available online at <http://cnx.org/content/m45626/1.2/>.

5.3.21.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3012: Using Nested Loops to Process Pixels [138] .

Click here [139] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.21.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3012s Slides
- File: Java3012s.htm
- Published: 01/06/02

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.22 Java3014: Cropping, Flipping, and Combining Pictures[140]

5.3.22.1 Table of Contents

[138] http://cnx.org/content/m44207
[139] http://cnx.org/content/m45626/latest/a0-Index.htm
[140] This content is available online at <http://cnx.org/content/m44238/1.7/>.

5.3.22.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [141] .

5.3.22.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.22.2.1.1 Images

5.3.22.2.1.2 Listings

5.3.22.3 Preview

In this module, you will learn how to:

- Work directly with individual pixels and keep track of coordinate values.
- Copy a portion of one picture into a specific location in another picture.

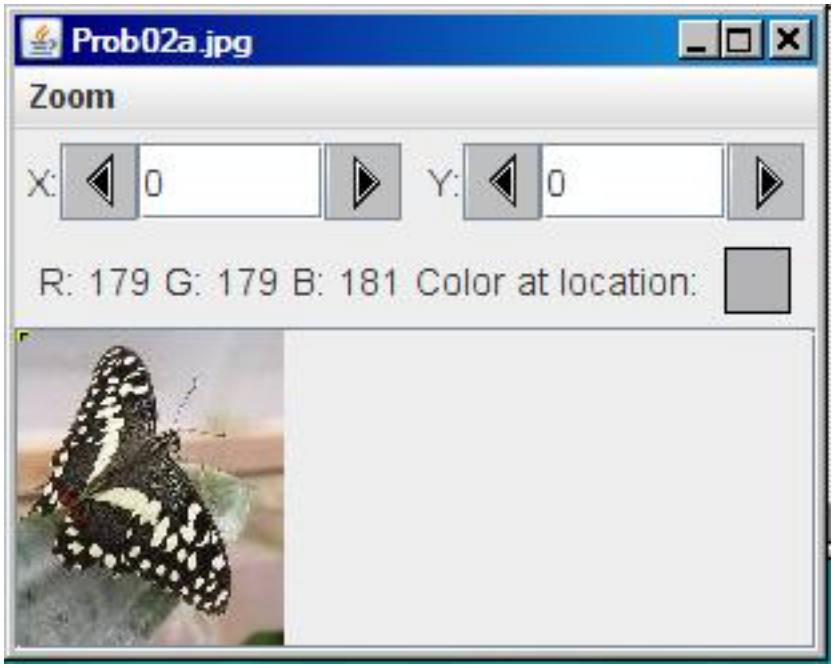
[141] http://cnx.org/content/m44148/latest/

- Crop and flip a picture.

Program specifications

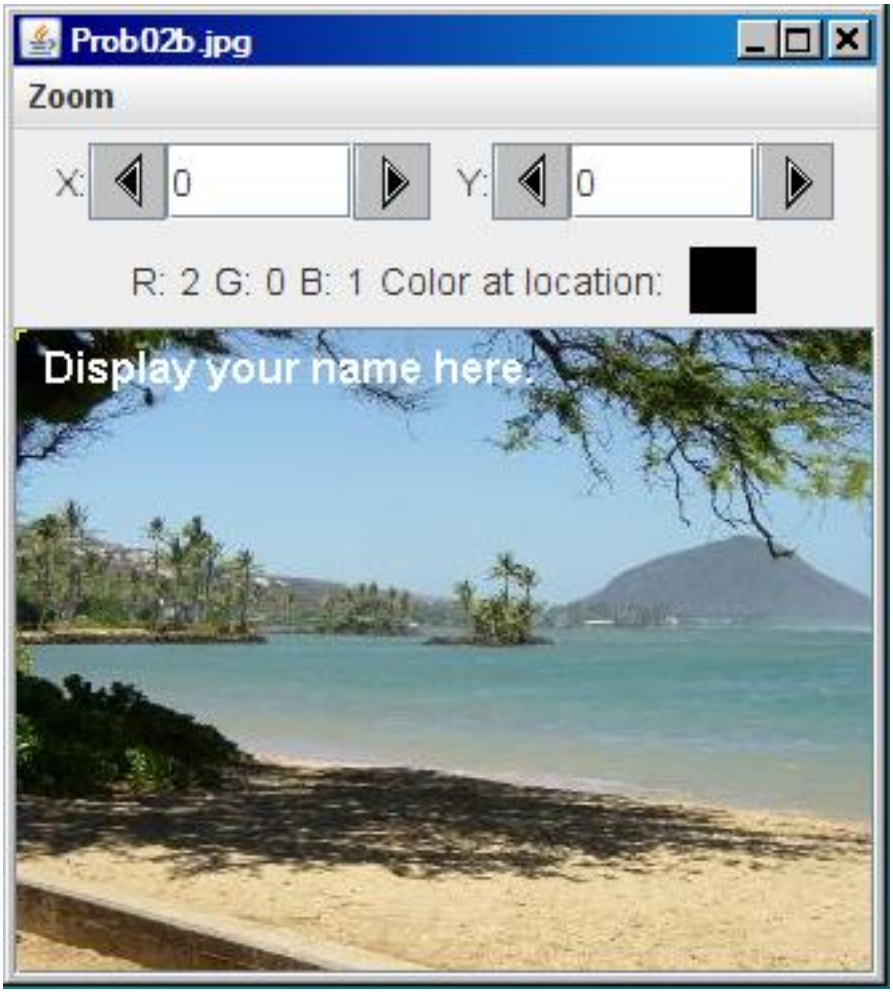
Write a program named **Prob02** that uses the class definition shown in Listing 1 (p. 981) and Ericson's media library along with the image files named **Prob02a.jpg** and **Prob02b.jpg** to produce the three graphic output images shown in Image 1 (p. 977) , Image 2 (p. 978) , and Image 3 (p. 979) .

Image 1: Raw butterfly image.



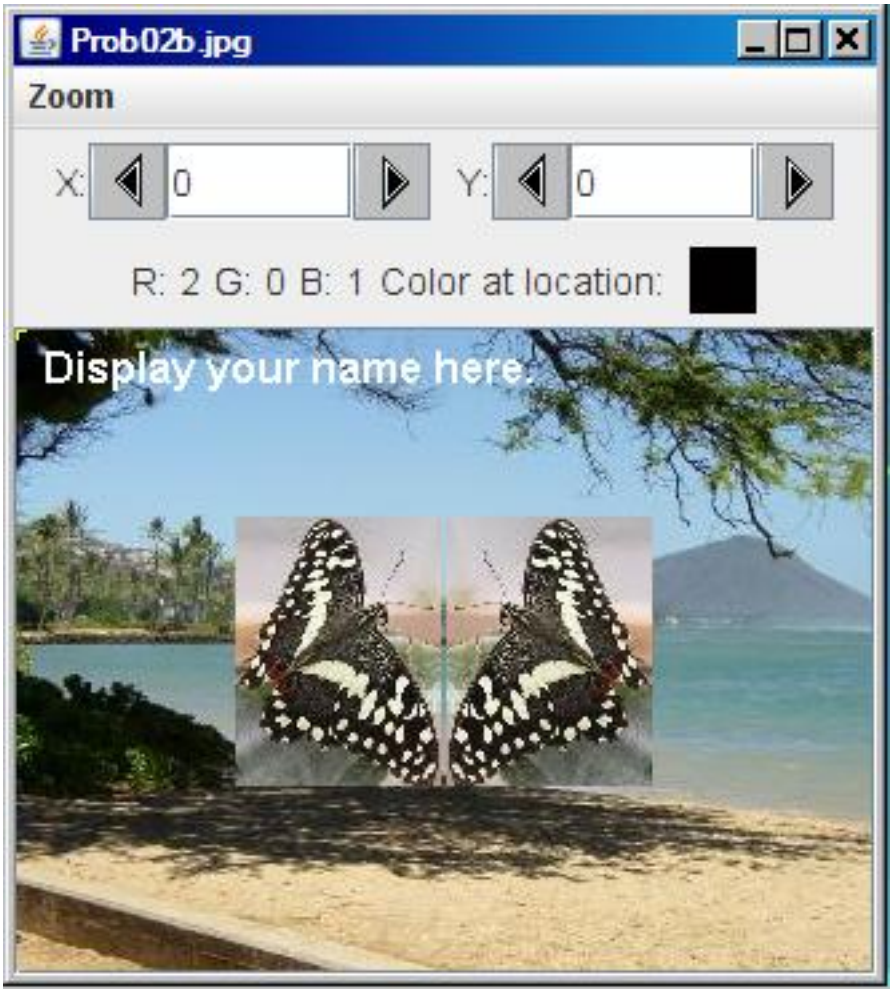
5.288

Image 2: Beach scene with student's name added.



5.289

Image 3: Composite image.



5.290

May define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob02** given in Listing 1 (p. 981) .

The facing butterfly images

The two facing images of the butterflies in the final output picture are separated by two pixels and those two images as a pair are centered in the picture of the beach.

Required text output

In addition to the three output images mentioned above, your program must display your name and the other three lines of text shown in Image 4 (p. 980) on the command-line screen:

Image 4: Required text output.

```
    Display your name here.
Picture, filename Prob02a.jpg height 118 width 100
Picture, filename Prob02b.jpg height 240 width 320
Picture, filename None height 101 width 77
```

5.291

5.3.22.4 General background information

This program copies a rectangular portion of a picture of a butterfly into a specific location in a picture of a beach.

The program also crops the butterfly picture to the same size as the portion that was copied into the beach picture and flips the cropped version to cause the butterfly to face left instead of facing right.

Then it copies the cropped and flipped image to a location two pixels to the right of the original copy of the butterfly in the beach image.

The two resulting images of the butterfly within the beach image are separated by two pixels, face one another, and are centered in the picture of the beach as shown in Image 3 (p. 979) .

Major evaluation areas

In order to successfully write this program, the student must, as a minimum be able to:

- Work directly with individual pixels and keep track of coordinate values.
- Copy a portion of one picture into a specific location in another picture.
- Crop and flip a picture.

5.3.22.5 Discussion and sample code

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 10 (p. 992) near the end of the module.

The driver class named Prob02

The driver class containing the **main** method is shown in Listing 1 (p. 981) .

Listing 1: The driver class named Prob02.

```
    import java.awt.Color;

public class Prob02{
  public static void main(String[] args){
    Picture[] pictures = new Prob02Runner().run();

    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob02
```

5.292

A reference to an array object

The call to the **run** method in Listing 1 (p. 981) may be new to you. This call expects to receive a reference to an array object of type **Picture[]** as a return value.

Save return value in variable named pictures

The return value from the **run** method is stored in the local reference variable named **pictures** .

Extract and print references to Picture objects

Then the reference variable is used to extract references to the individual **Picture** objects encapsulated in the array. Those references are passed to the **println** method causing the last three lines of text shown in Image 4 (p. 980) to be displayed on the command line screen.

Beginning of the Prob02Runner class

The class named **Prob02Runner** begins in Listing 2 (p. 981) , which shows the constructor for the class.

Listing 2: Beginning of the Prob02Runner class.

```
    class Prob02Runner{

  public Prob02Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
```

5.293

The constructor simply causes the student's name to be displayed on the command line screen, producing the first line of output text shown in Image 4 (p. 980) .

Beginning of the run method

The **run** method, that was called in Listing 1 (p. 981) begins in Listing 3 (p. 982) .

Listing 3: Beginning of the run method.

```
  public Picture[] run(){
Picture picA = new Picture("Prob02a.jpg");
picA.explore();

Picture picB = new Picture("Prob02b.jpg");
picB.addMessage("Display your name here.",10,20);
picB.explore();

Picture picC = cropAndFlip(picA,4,5,80,105);
```

5.294

Listing 3 (p. 982) instantiates two **Picture** objects from image files and displays them by calling the **explore** method on each **Picture** object. In addition, the student's name is added near the upper-left corner of the beach image. This code results in the images shown in Image 1 (p. 977) and Image 2 (p. 978) .

Call the cropAndFlip method

Then Listing 3 (p. 982) calls the **cropAndFlip** method passing the reference to the butterfly image of Image 1, along with some other information as parameters. The return value is stored in a new local reference variable of type **Picture** named **picC** .

Put discussion of the run method on hold

I will put the discussion of the **run** method on temporary hold at this point and explain the method named **cropAndFlip** , which begins in Listing 4 (p. 982) .

Beginning of the cropAndFlip method

The **cropAndFlip** method crops a picture to the specified coordinate values and flips it around a vertical line at its center.

Listing 4: Beginning of the cropAndFlip method.

```
private Picture cropAndFlip(Picture pic,
                            int x1,int y1,
                            int x2,int y2){
  Picture output = new Picture(x2-x1+1,y2-y1+1);

  int width = output.getWidth();

  Pixel pixel = null;
  Color color = null;
```

5.295

Incoming parameters

In addition to a reference to the picture to be processed, the method receives four incoming integer values as parameters. The parameters named **x1** and **y1** specify the coordinates of the upper-left corner of a rectangular area of the picture that is to be retained in the output.

The parameters named **x2** and **y2** specify the coordinates of the lower-right corner of the rectangular area of the picture that is to be retained in the output.

An empty Picture object

Listing 4 (p. 982) begins by creating an empty **Picture** object of the correct size to hold the cropped image. A reference to the empty picture is saved in the local reference variable named **output** .

Then Listing 4 (p. 982) gets and saves the width of the output picture.

Following this, Listing 4 (p. 982) declares two local working variables named **pixel** *(of type* ***Pixel*** *)* and **color** *(of type* ***Color*** *)* .

Process using nested loops

Listing 5 (p. 983) uses a pair of nested **for** loops to cause the output picture to be a cropped version of the picture received as an incoming parameter. The cropped image is flipped around its center.

Listing 5: Process using nested loops.

```
      for(int col = x1;col < (x2+1);col++){
    for(int row = y1;row < (y2+1);row++){
      color = pic.getPixel(col,row).getColor();
      pixel = output.getPixel(width-col+x1-1,row-y1);
      pixel.setColor(color);
    }//end inner loop
  }//end outer loop

  return output;
}//end cropAndFlip method
```

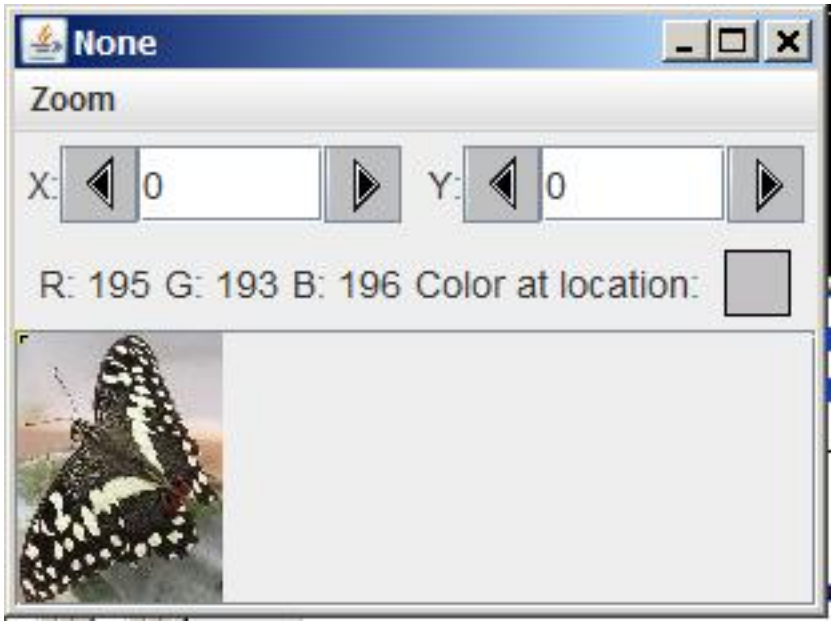
5.296

The code in Listing 5 (p. 983) copies the pixel colors of the selected pixels of the incoming image to the pixels of the output image, flipping the image around its center line in the process.

Cropped and flipped version of the butterfly image

If you display the picture referred to by **output** in Listing 5 (p. 983) , you will get the image shown in Image 5 (p. 984) .

Image 5: Cropped and flipped version of the butterfly image.



5.297

Compare with the original butterfly picture

If you compare Image 5 (p. 984) with Image 1 (p. 977) , you will see that pixels on the outer edges of Image 1 (p. 977) have been discarded and the resulting image has been flipped around its centerline.

End of the cropAndFlip method

Image 5 (p. 984) returns a reference to the new image and ends the method named **cropAndFlip** . The returned value is stored in the variable named **picC** in Listing 3 (p. 982) .

Original image not modified

Note that the code in the **cropAndFlip** method does not modify the original image of the butterfly. Instead, it extracts pixel data from the original image to produce a new image. When control returns to the **run** method in Listing 3 (p. 982) , a reference to the new image is stored in the variable named **picC** .

Call the copyPictureWithCrop method from the run method

Control has now returned to the **run** method, picking up where Listing 3 (p. 982) left off. The next statement in the **run** method is shown in Listing 6 (p. 984) .

Listing 6: Call the copyPictureWithCrop method from the run method.

```
copyPictureWithCrop(picA,picB,82,70,4,5,77,101);
```

5.298

Put the run method on hold again

Once again, I will put the **run** method on hold while I explain the method named **copyPictureWithCrop**, which begins in Listing 7 (p. 985).

Beginning of the method named copyPictureWithCrop

The first two incoming parameters named **source** and **dest** are references to a source picture and a destination picture.

When the method is called in Listing 6 (p. 984), the source picture is the original butterfly picture shown in Image 1 (p. 977) and the destination picture is the beach picture shown in Image 2 (p. 978).

Listing 7: Beginning of the method named copyPictureWithCrop.

```
  private void copyPictureWithCrop(
       Picture source,
       Picture dest,
       int xOff,
       int yOff,
       int xCoor,
       int yCoor,
       int width,
       int height){

//Confirm that source will fit in destination
if(((width+xOff) <= dest.getWidth()) &&
     ((height+yOff) <= dest.getHeight())){

  Pixel pixel = null;
  Color color = null;
```

Copy source to destination

The method named **copyPictureWithCrop** copies part of the source picture into the destination picture with an offset on both axes after first confirming that the part will fit. The method does nothing if the part won't fit.

The copy process causes selected pixel colors in the destination picture to be replaced by pixel colors from the source picture.

The offset values

The next two parameters named **xOff** and **yOff** in Listing 7 (p. 985) specify the location in the destination picture where the upper-left corner of the cropped source picture is to be located.

The statement in Listing 6 (p. 984) passes the values (82,70) for these two values. This is the location of the upper left corner of the left-most butterfly image in Image 3 (p. 979).

NOTE: **Not really cropped** For clarity, I will refer to this as a cropped source picture even though the program doesn't actually save a cropped version of the picture as was the case with the **cropAndFlip** method.

The program simply copies a rectangular portion of the source picture into the destination picture.

Upper-left cropping corner

The parameters named **xCoor** and **yCoor** in Listing 7 (p. 985) specify the upper-left corner of the rectangular area of pixels that is to be preserved when the source image is cropped.

Coordinate values of (4,5) are passed for these two values when the method is called in Listing 6 (p. 984) .

Same values as Listing 3

Note that these are the same two values that were passed for this purpose when the **cropAndFlip** method was called in Listing 3 (p. 982) .

Two ways to specify a rectangle

There are two commonly used ways to specify a rectangular area in programming. One way is to specify the coordinates of the upper-left and bottom right corners. This is the approach used in the **cropAndFlip** method in Listing 4 (p. 982) .

The other way is to specify the coordinates of the upper-left corner and then to specify the width and the height. This is the approach used in the **copyPictureWithCrop** method in Listing 7 (p. 985) .

The width and height parameters

The parameters named **width** and **height** in Listing 7 (p. 985) specify the width and height of the rectangular area of pixels that is to be preserved when the source picture is cropped.

If you compare the width and height parameter values passed in Listing 6 (p. 984) with the coordinate values passed in Listing 3 (p. 982) , you will see that the same rectangular area of the butterfly image is being preserved after cropping in both cases.

Confirm that the cropped image will fit

Listing 7 (p. 985) begins by confirming that the cropped rectangular area of the source picture will fit within the destination picture when placed at the specified location. If the conditional clause of the **if** statement returns true, then the code in the body of the statement will be executed. If not, control bypasses the body of the **if** statement and the source picture will not be copied into the destination picture.

Process using nested for loops

As was the case in Listing 4 (p. 982) , Listing 7 (p. 985) declares two working variables named **pixel** and **color** .

The variables named **pixel** and **color** are used along with various parameter values in the pair of nested **for** loops shown in Listing 8 (p. 987) to crop the source picture and to copy the cropped source picture into the destination picture at the specified location.

Listing 8: Process using nested loops.

```
        for(int col = 0;col < width;col++){
       for(int row = 0;row < height;row++){
        color = source.getPixel(
                    col + xCoor,row + yCoor).getColor();
        pixel = dest.getPixel(col+xOff,row+yOff);
        pixel.setColor(color);
       }//end inner loop
      }//end outer loop

    }//end if

  }//end copyPictureWithCrop method

}//end class Prob02Runner
```

5.300

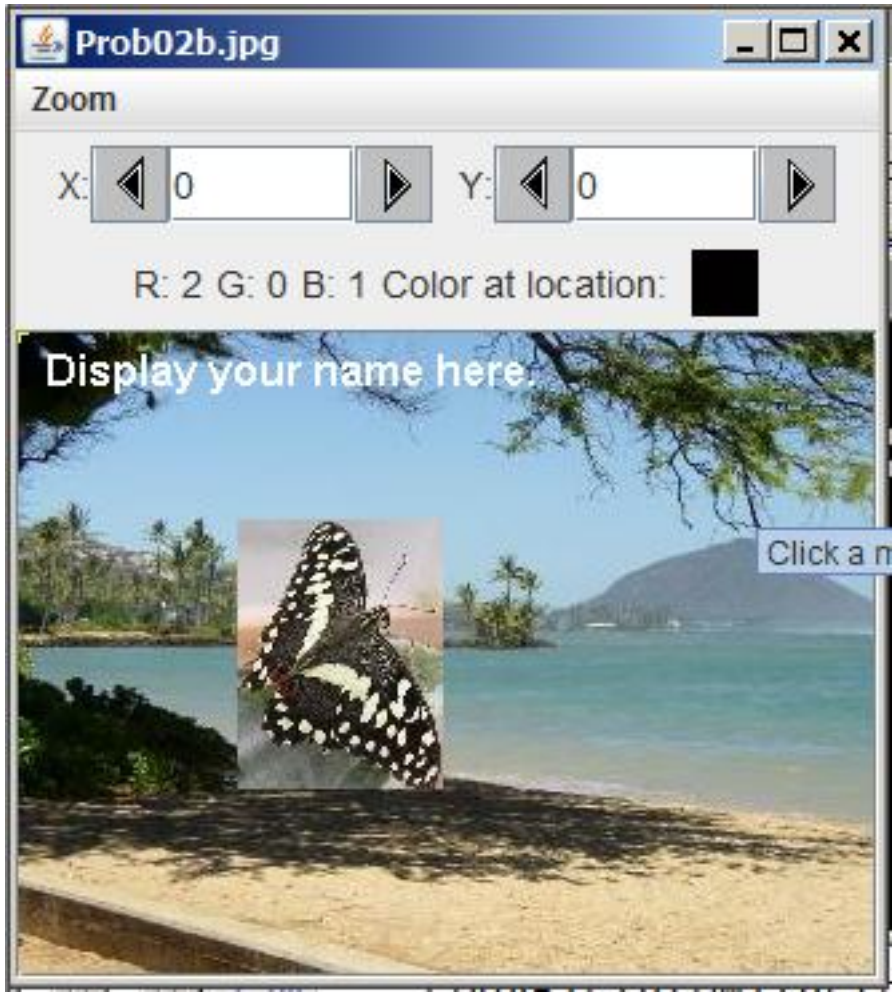
Not as complicated as it looks

Although the arithmetic operations involved in Listing 8 (p. 987) can be daunting, the code in Listing 8 (p. 987) is doing nothing more than replacing selected pixel colors in the destination picture with selected pixel colors from the source picture.

Partially complete version of the output picture.

If you were to display the destination picture before returning control back to the **run** method in Listing 8 (p. 987) , you would see the image shown in Image 6 (p. 988) .

Image 6: Partially complete version of the output picture.



5.301

At this point, only one cropped version of the butterfly image has been copied into the beach image.

Return control to the run method

The **copyPictureWithCrop** method terminates in Listing 8 (p. 987) and returns control to the **run** method, picking up where Listing 6 (p. 984) left off.

The remainder of the run method

The remainder of the **run** method is shown in Listing 9 (p. 989) .

Listing 9: The remainder of the run method.

```
      copyPictureWithCrop(picC,picB,161,70,0,0,77,101);

   picB.explore();

   Picture[] output = {picA,picB,picC};
   return output;
}//end run
```

5.302

Call the copyPictureWithCrop method again

Listing 9 (p. 989) begins by calling the **copyPictureWithCrop** method again. This time, however, the picture shown in Image 5 (p. 984) is passed as the source image with the same picture as before being passed as the destination image.

The offset coordinates

In this case, the offset coordinate values specify the upper-left corner of the right-most butterfly image in Image 3 (p. 979) .

The cropping parameters

The final four parameters that are passed in Listing 9 (p. 989) specify that the entire source picture is to be copied into the destination picture.

Display the destination picture

When the **copyPictureWithCrop** method returns, Listing 9 (p. 989) calls the **explore** method to display the current state of the destination picture. The result is shown in Image 3 (p. 979) .

A new array object

Finally, Listing 9 (p. 989) instantiates a new array object, populates it with references to three **Picture** objects, and returns control to the **main** method code in Listing 1 (p. 981) returning a reference to the array object in the process.

The code in Listing 1 (p. 981) saves the reference to the array object in the variable named **pictures** .

Pass Picture object references to println method

Then Listing 1 (p. 981) extracts and passes each of the three **Picture** object references to the **println** method causing the last three lines of text shown in Image 4 (p. 980) to be displayed on the command-line screen.

The second line of output text (**picA**) describes the raw butterfly image shown in Image 1 (p. 977) .

The third line of output text for (**picB**) describes the beach scene shown in Image 2 (p. 978) and Image 3 (p. 979) .

The last line of output text (**picC**) describes the cropped and flipped version of the butterfly image shown in Image 5 (p. 984) .

5.3.22.6 Run the program

I encourage you to copy the code from Listing 10 (p. 992) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click here [142] and here [143] to download the two required input image files.

[142] http://cnx.org/content/m44238/latest/Prob02a.jpg
[143] http://cnx.org/content/m44238/latest/Prob02b.jpg

5.3.22.7 Summary

In this module, you learned how to:

- Work directly with individual pixels and keep track of coordinate values.
- Copy a portion of one picture into a specific location in another picture.
- Crop and flip a picture.

5.3.22.8 What's next?

You will l earn to write a program to do green-screen processing in the next module.

5.3.22.9 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 07 [144]
  - Part01 [145]
  - Part02 [146]
  - Part03 [147]

5.3.22.10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Cropping, Flipping, and Combining Pictures
- File: Java3014.htm
- Published: 08/01/12
- Revised: 02/14/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[144] http://www.youtube.com/playlist?list=PL3D7CCC0D884E2EF4
[145] http://www.youtube.com/watch?v=AY1oMeuFWwY
[146] http://www.youtube.com/watch?v=IWNm1xWA7wQ
[147] http://www.youtube.com/watch?v=P0thqN0Fofs

5.3.22.11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 992) below.

Listing 10: Complete program listing.

```
    /*File Prob02 Copyright 2008 R.G.Baldwin
Revised 12/16/08
*******************************************************/
import java.awt.Color;

public class Prob02{
  public static void main(String[] args){
    Picture[] pictures = new Prob02Runner().run();

    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob02
//=======================================================//

class Prob02Runner{

  public Prob02Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
  //-------------------------------------------------------//

  public Picture[] run(){
    Picture picA = new Picture("Prob02a.jpg");
    picA.explore();
    Picture picB = new Picture("Prob02b.jpg");
    picB.addMessage("Display your name here.",10,20);
    picB.explore();

    Picture picC = cropAndFlip(picA,4,5,80,105);

    copyPictureWithCrop(picA,picB,82,70,4,5,77,101);
    copyPictureWithCrop(picC,picB,161,70,0,0,77,101);


    picB.explore();

    Picture[] output = {picA,picB,picC};
    return output;
  }//end run
  //-------------------------------------------------------//

  //Crops a picture to the specified coordinate values and
  // flips it around a vertical line at its center.
  private Picture cropAndFlip(Picture pic,int x1,int y1,
                              int x2,int y2){
    Picture output = new Picture(x2-x1+1,y2-y1+1);

    int width = output.getWidth();
    Pixel pixel = null;
    Color color = null;
    for(int col = x1;col < (x2+1);col++){
      for(int row = y1;row < (y2+1);row++){
        color = pic.getPixel(col,row).getColor();
```

-end-

5.3.23 Java3014r Review[148]

5.3.23.1 Table of Contents

5.3.23.2 Preface

This module contains review questions and answers keyed to the module titled Java3014: Cropping, Flipping, and Combining Pictures [149] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.23.3 Questions

5.3.23.3.1 Question 1 .

True or False? The code in Listing 1 (p. 995) combined with the images in Image 1 (p. 996) and Image 2 (p. 996) produces the output shown in Image 3 (p. 997) .

[148] This content is available online at <http://cnx.org/content/m45778/1.1/>.
[149] http://cnx.org/content/m44238

Listing 1. Question 1.

```
/*File Java3014ra Copyright 2013 R.G.Baldwin
Revised 02/15/13
*******************************************************/
import java.awt.Color;

public class Java3014ra{
  public static void main(String[] args){
    Picture[] pictures = new Java3014raRunner().run();
  }//end main method
}//end class Java3014ra
//=======================================================//

class Java3014raRunner{

  public Picture[] run(){
    Picture picA = new Picture("Prob02a.jpg");
    Picture picB = new Picture("Prob02b.jpg");
    Picture picC = cropAndFlip(picA,4,5,80,105);
    copyPictureWithCrop(picA,picB,130,10,4,5,77,101);
    copyPictureWithCrop(picC,picB,130,120,0,0,77,101);
    picB.explore();

    Picture[] output = {picA,picB,picC};
    return output;

  }//end run
  //-----------------------------------------------------//

  //Crops a picture to the specified coordinate values and
  // flips it around a vertical line at its center.
  private Picture cropAndFlip(Picture pic,int x1,int y1,
                              int x2,int y2){
    Picture output = new Picture(x2-x1+1,y2-y1+1);

    int width = output.getWidth();
    Pixel pixel = null;
    Color color = null;
    for(int col = x1;col < (x2+1);col++){
      for(int row = y1;row < (y2+1);row++){
        color = pic.getPixel(col,row).getColor();
        pixel = output.getPixel(width-col+x1-1,row-y1);
        pixel.setColor(color);
      }//end inner loop
    }//end outer loop

    return output;
  }//end crop and flip
  //-----------------------------------------------------//

  //Copies part of the source picture into the destination
  // picture with an offset on both axes after first
  // confirming that the part will fit. Does nothing if it
  // won't fit.
  private void copyPictureWithCrop(
```

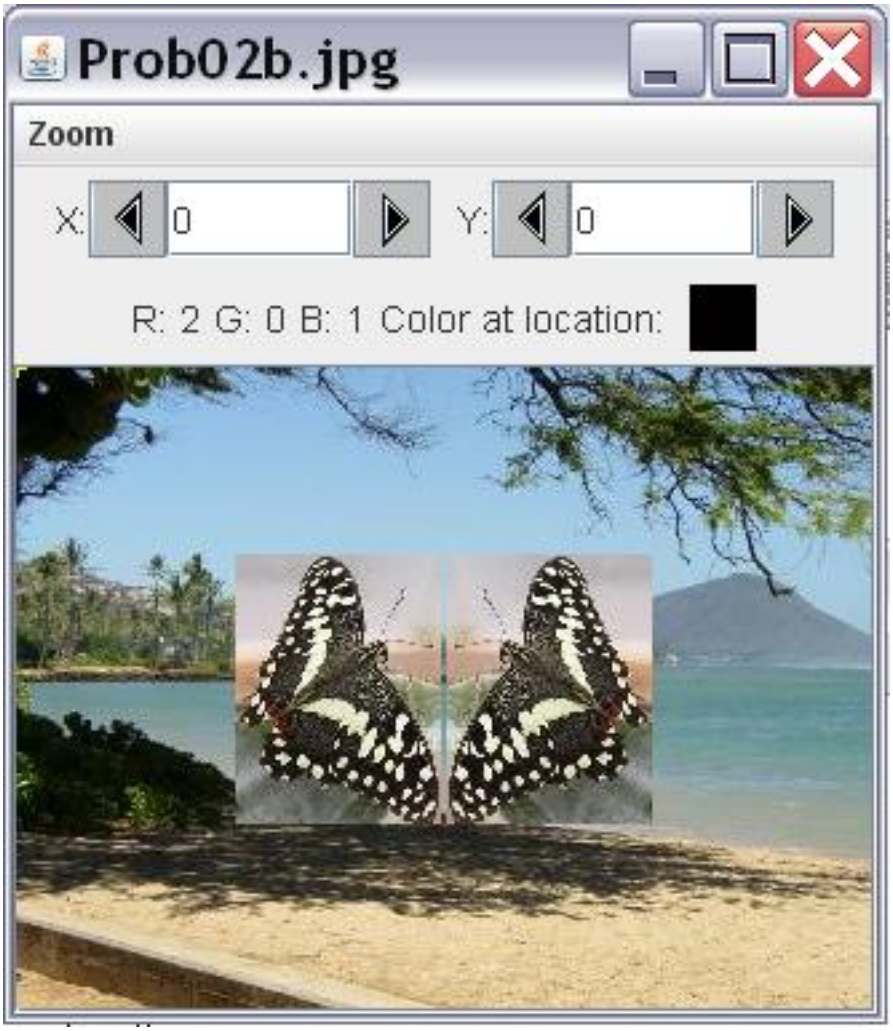
Image 1. Prob02a.jpg.

5.305

Image 2. Prob02b.jpg.

5.306

Image 3. Possible output image.



5.307

Answer 1 (p. 1000)

5.3.23.3.2 Question 2

True or False? The call to the run method in Listing 2 (p. 998) returns a reference to an object of the class **Picture** .

Listing 2. Question 2.

```
Picture[] pictures = new Java3014raRunner().run();
```

5.308

Answer 2 (p. 1000)

5.3.23.4 Images

- Image 1 (p. 996) . Prob02a.jpg.
- Image 2 (p. 996) . Prob02b.jpg.
- Image 3 (p. 997) . Possible output image.
- Image 4 (p. 1001) . Answer 1.

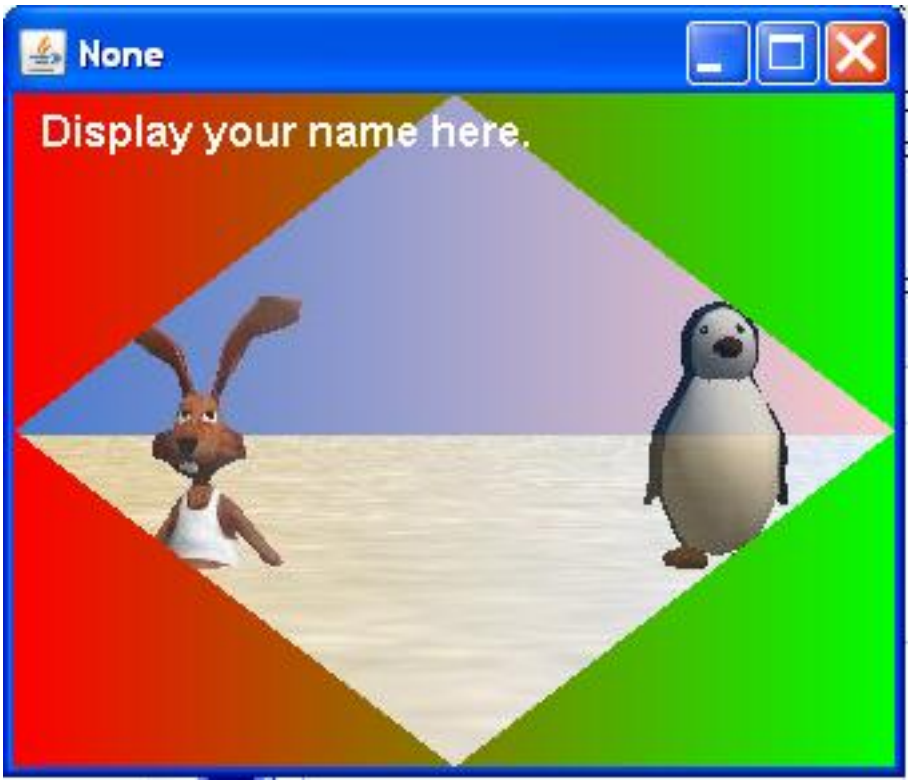
5.3.23.5 Listings

- Listing 1 (p. 995) . Question 1.
- Listing 2 (p. 998) . Question 2.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.23.6 Answers

5.3.23.6.1 Answer 2

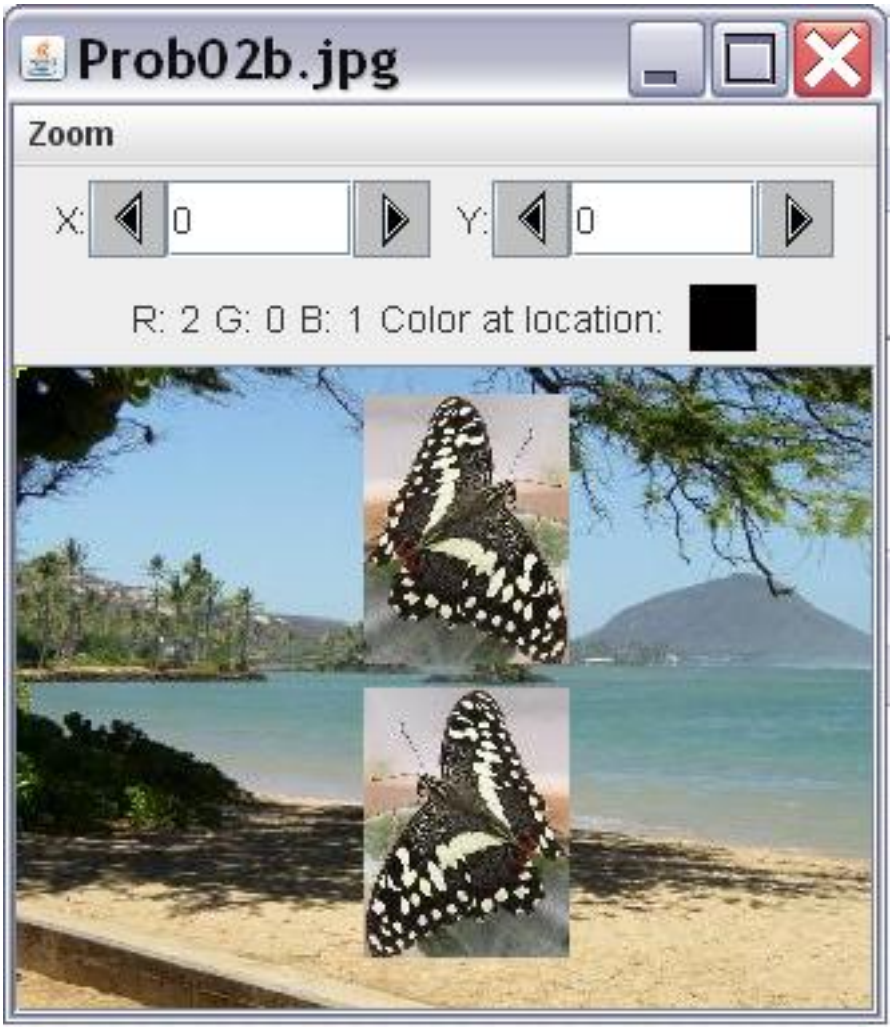
False. The call to the run method in Listing 2 (p. 998) returns a reference to an array object whose elements may or may not contain references to objects of the class **Picture** *(or some subclass of the class* ***Picture*** *)* . However, since array objects in Java may have a **length** of 0, without seeing the source code for the **run** method, it is impossible to know what is contained in the array object.

Back to Question 2 (p. 997)

5.3.23.6.2 Answer 1

False. The code in Listing 1 (p. 995) combined with the images in Image 1 (p. 996) and Image 2 (p. 996) produces the output shown in Image 4 (p. 1001) .

Image 4. Answer 1.



5.309

Back to Question 1 (p. 994)

5.3.23.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3014r Review
- File: Java3014r.htm
- Published: 02/15/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.24 Java3014s Slides[150]

5.3.24.1 Table of Contents

5.3.24.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3014: Cropping, Flipping, and Combining Pictures [151] .

Click here [152] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.24.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3014s Slides
- File: Java3014s.htm
- Published: 01/06/13

[150] This content is available online at <http://cnx.org/content/m45628/1.3/>.
[151] http://cnx.org/content/m44238
[152] http://cnx.org/content/m45628/latest/a0-Index.htm

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.25 Java3016: Green-Screen Processing[153]

5.3.25.1 Table of Contents

5.3.25.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [154] .

5.3.25.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

[153] This content is available online at <http://cnx.org/content/m44210/1.7/>.
[154] http://cnx.org/content/m44148/latest/

5.3.25.2.1.1 Images

- Image 1 (p. 1005) . Input image file Prob03a.bmp.
- Image 2 (p. 1006) . Input image file Prob03b.bmp.
- Image 3 (p. 1007) . Input image file Prob03c.bmp.
- Image 4 (p. 1008) . Input image file Prob03d.jpg.
- Image 5 (p. 1009) . Output picture.
- Image 6 (p. 1010) . Required output text.
- Image 7 (p. 1013) . Front view of the skater after cropping.

5.3.25.2.1.2 Listings

- Listing 1 (p. 1011) . The driver class named Prob03.
- Listing 2 (p. 1011) . Beginning of the class named Prob03Runner.
- Listing 3 (p. 1012) . Beginning of the run method.
- Listing 4 (p. 1014) . Remainder of the run method.
- Listing 5 (p. 1016) . The greenScreenDraw method.
- Listing 6 (p. 1019) . Complete program listing.

5.3.25.3 Preview

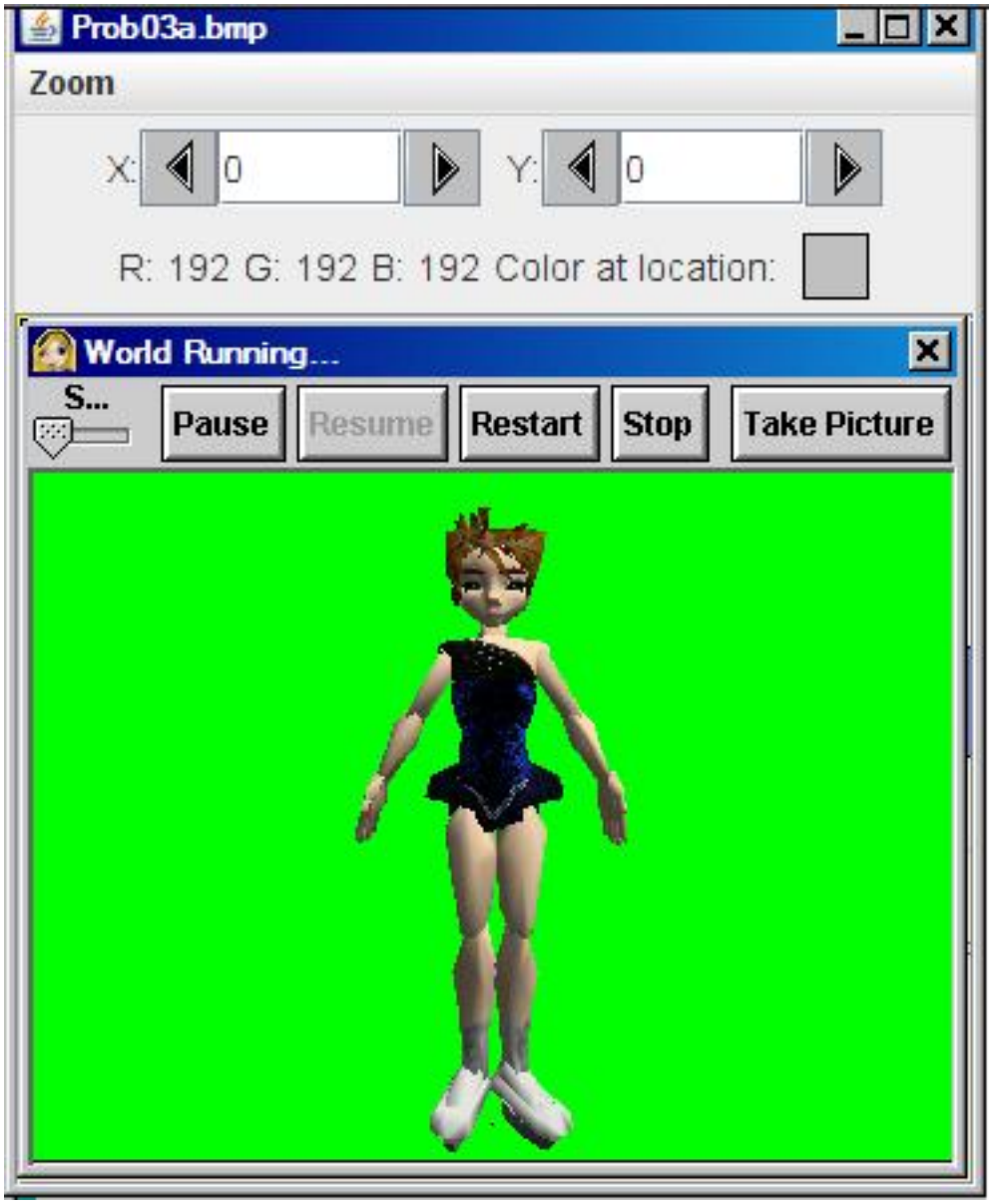
n this lesson, you will learn how to write a program to do *green-screen* processing to superimpose a source image onto a destination image while making the green background of the source image appear to be transparent.

Program specifications

Write a program named **Prob03** that uses the class definition shown in Listing 1 (p. 1011) and Ericson's media library along with the image files in the following list to produce the five graphic output images shown in Image 1 (p. 1005) through Image 5 (p. 1009) .

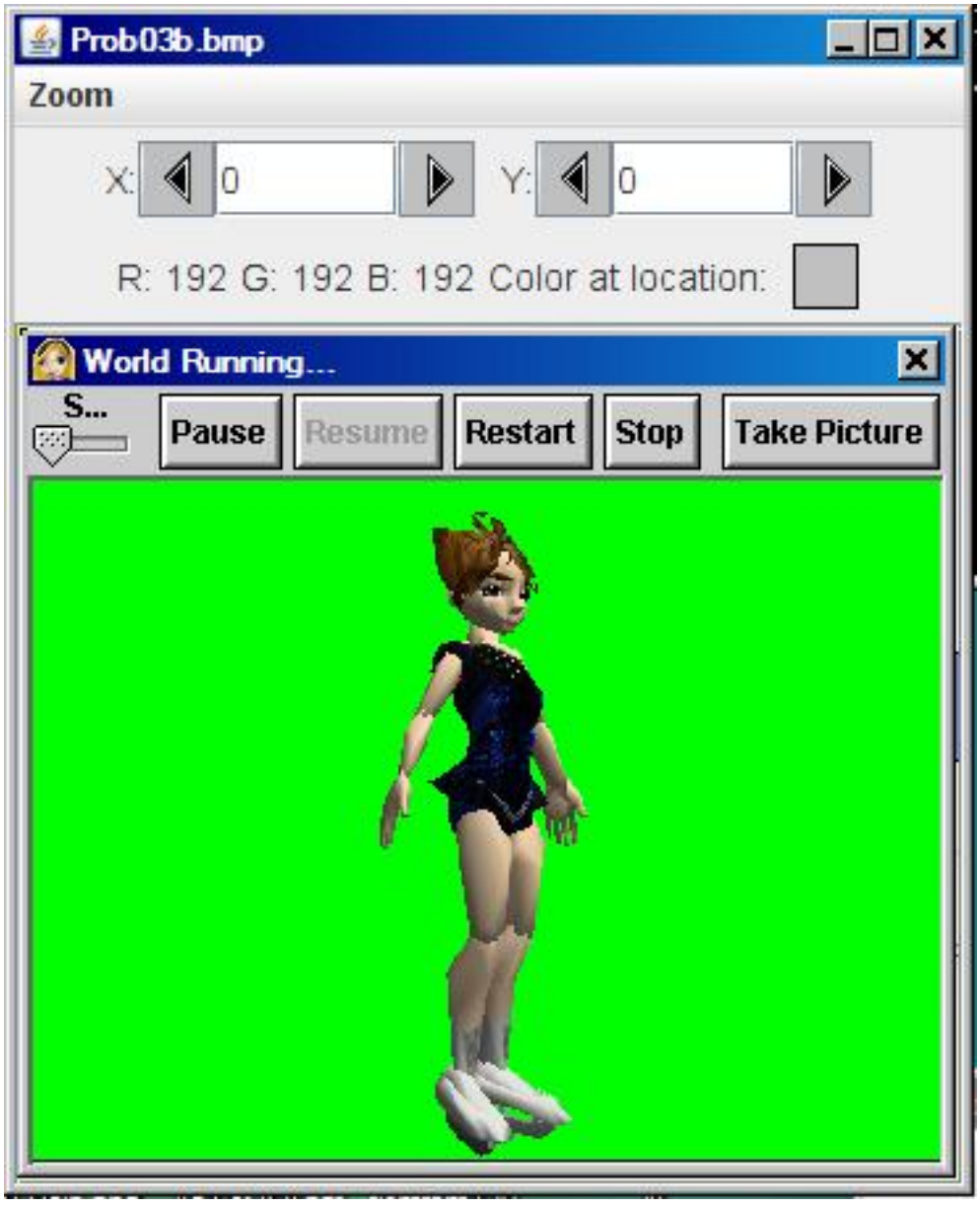
- Prob03a.bmp
- Prob03b.bmp
- Prob03c.bmp
- Prob03d.jpg

Image 1: Input image file Prob03a.bmp.



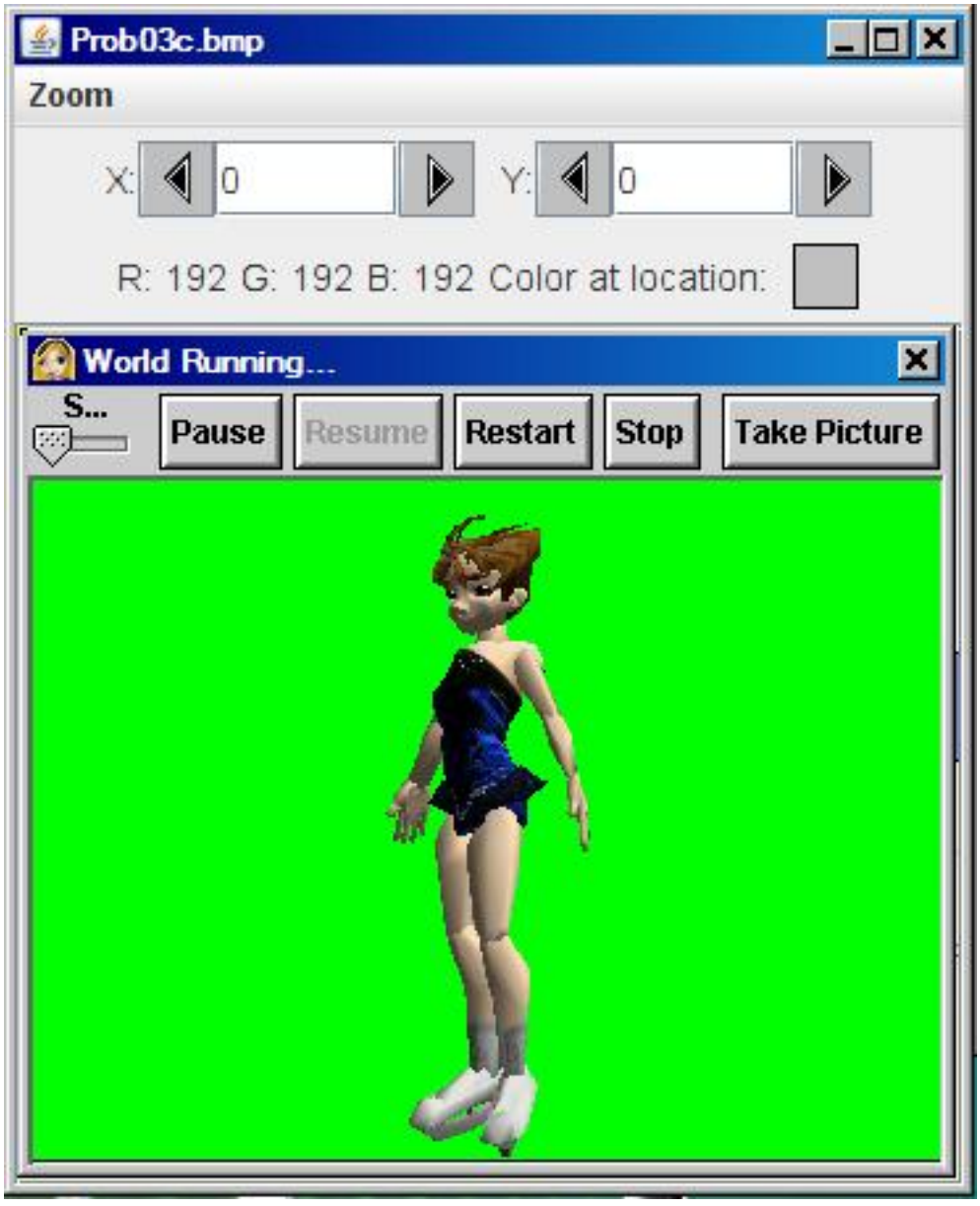
5.310

Image 2: Input image file Prob03b.bmp.



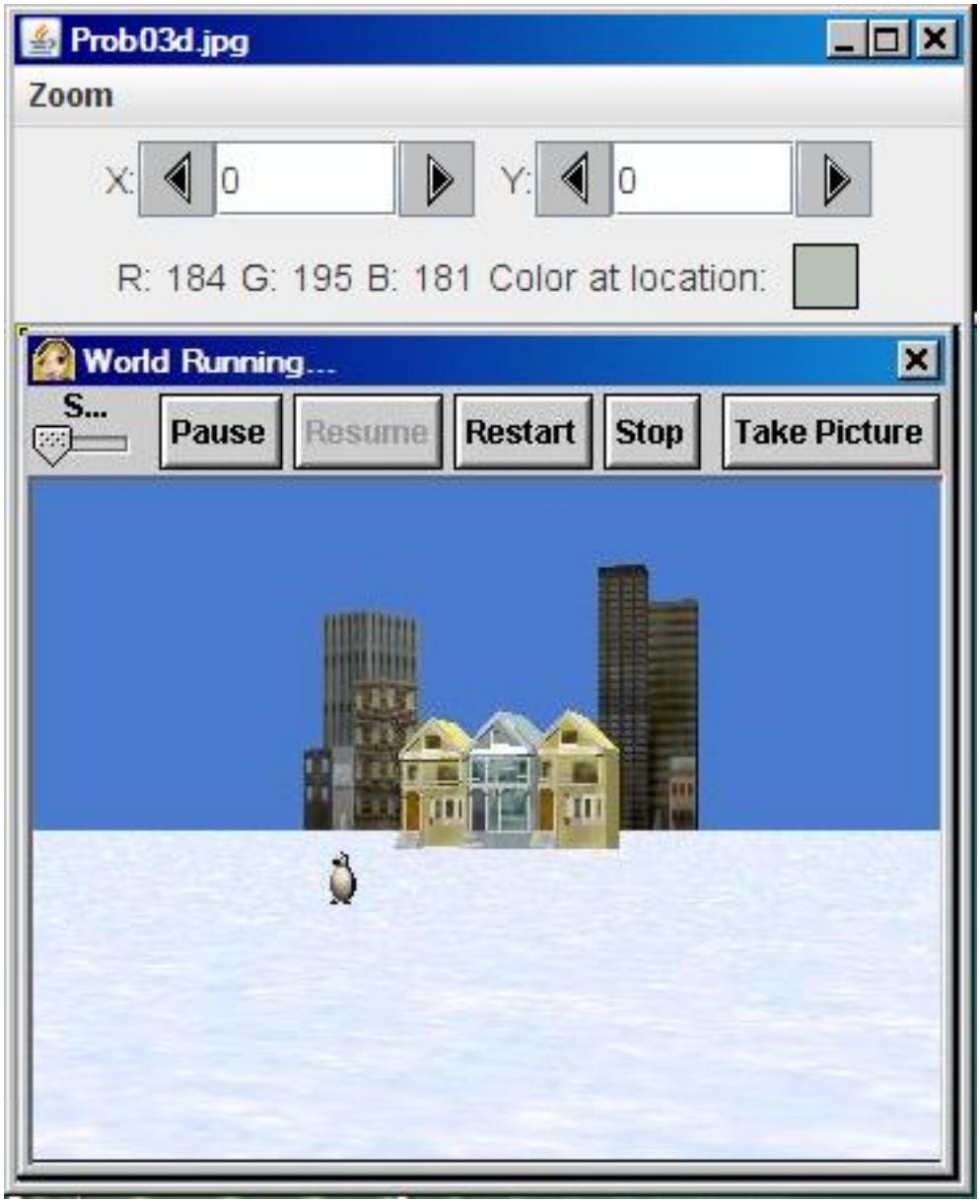
5.311

Image 3: Input image file Prob03c.bmp.



5.312

Image 4: Input image file Prob03d.jpg.



5.313

Image 5: Output picture.

5.314

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob03** given in Listing 1 (p. 1011) .

Required output text

In addition to the output images mentioned above, your program must display your name and one other line of text on the command-line screen as shown in Image 6 (p. 1010) .

Image 6: Required output text.

```
    Display your name here.
Picture, filename None height 256 width 344
```

5.315

5.3.25.4 General background information

This program receives three views of an ice skater in **bmp** files with a pure green background along with a **jpg** file containing a snow scene.

All four files show the Alice [155] runtime panel with the words ***World Running...*** and sever associated buttons. *(See Image 1 (p. 1005) .)*

Program behavior

The program performs the following actions:

- Crops the snow scene to remove the Alice runtime panel.
- Crops the three views of the skater to remove the Alice runtime panel along with excess blank green background.
- Scales two of the views of the skater to smaller sizes.
- Does green-screen processing to place the three views of the skater at different locations in the snow scene.
- Uses position along with size to create an optical illusion of a 3D scene of three ice skaters and a penguin standing at different locations on a frozen lake *(see Image 5 (p. 1009))* .

Programming skills required

In order to write this program, the student must be able to, as a minimum, write a green-screen processing method.

5.3.25.5 Discussion and sample code

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 6 (p. 1019) near the end of the lesson.

The driver class named Prob03

The driver class containing the **main** method is shown in Listing 1 (p. 1011) .

[155] http://www.alice.org/

Listing 1: The driver class named Prob03.

```
    import java.awt.Color;

public class Prob03{
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
  }//end main
}//end class Prob03
```

5.316

The **main** method in Listing 1 (p. 1011) instantiates a new object of the class named **Prob03Runner** and calls the method named **run** that belongs to that object.

When the **run** method returns, the program terminates.

Beginning of the class named Prob03Runner

The beginning of the class named **Prob03Runner** , and its constructor, is shown in Listing 2 (p. 1011) .

Listing 2: Beginning of the class named Prob03Runner.

```
    class Prob03Runner{

  public Prob03Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
```

5.317

The constructor simply displays the student's name on the command line screen producing the first line of text shown in Image 6 (p. 1010) .

Beginning of the run method

The beginning of the **run** method that is called in Listing 1 (p. 1011) is shown in Listing 3 (p. 1012) .

Listing 3: Beginning of the run method.

```
 public void run(){

//A view facing the front of the skater.
Picture front = new Picture("Prob03a.bmp");
front.explore();
front = crop(front,123,59,110,256);

//A view showing the right side of the skater.
Picture right = new Picture("Prob03b.bmp");
right.explore();
right = crop(right,123,59,110,256);

//A view showing the left side of the skater.
Picture left = new Picture("Prob03c.bmp");
left.explore();
left = crop(left,123,59,110,256);

//This will be the background for the new picture.
Picture snowScene = new Picture("Prob03d.jpg");
snowScene.explore();
snowScene = crop(snowScene,6,59,344,256);
```

5.318

The code in Listing 3 (p. 1012) instantiates, displays, and crops the four input images.

All four images must be cropped to remove the Alice runtime window. In addition, the three skater images are also cropped to remove excess blank green background material.

Image formats: bmp versus jpg

Note that the three views of the skater are extracted from **bmp** image files instead of **jpg** image files. This is necessary in order to preserve the pure green background color. Storing the images as **jpg** files would corrupt the background color in the low order bits making it more difficult to achieve the green-screen processing required by this program.

The method named crop

Listing 3 calls the **crop** method four times in succession. once for each of the four **Picture** objects instantiated from the image files.

I explained image cropping in an earlier module. The **crop** method used in this program is very similar to the methods that I explained in the earlier module, so I won't explain it again in this module. You can view the **crop** method in detail in Listing 6 (p. 1019) near the end of the module.

Five incoming parameters

The **crop** method requires five incoming parameters. The first parameter is a reference to the **Picture** object that is to be cropped. The remaining four integer parameters specify the rectangular area that is to be preserved after the picture is cropped.

The rectangular area to be preserved

The first two integers specify the column and row coordinates for the upper-left corner of the rectangular area that is to be preserved. The last two integers specify the width and the height of the rectangle that is

to be preserved.

Note that in all four cases, the height of the rectangular area that is to be preserved is 256 pixels. This will be important later on with respect to scaling two of the images.

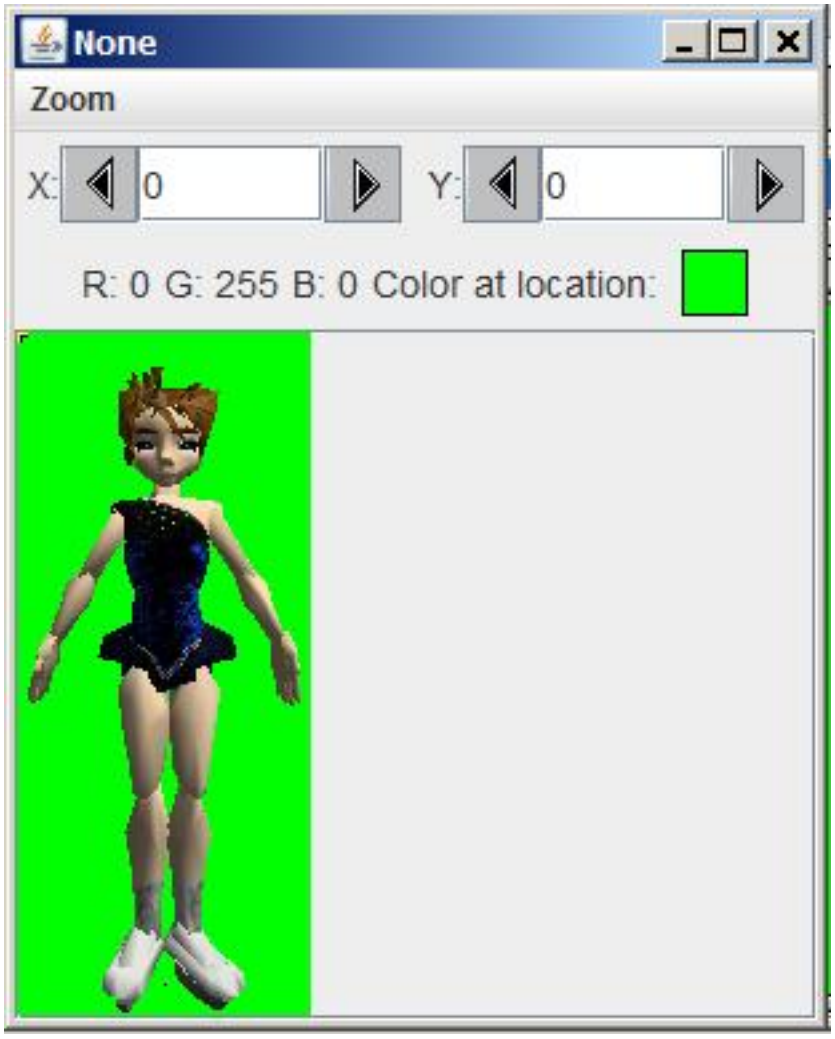
Returns a reference to a cropped picture

The **crop** method returns a reference to a **Picture** object that is a cropped version of the picture whose reference is passed to the method. In each case, the code in Listing 3 (p. 1012) replaces the reference to the original **Picture** object with the reference to the cropped **Picture** object.

Front view of the skater after cropping

If you were to display the **Picture** object referred to by the variable **front** after cropping, you would see the image shown in Image 7 (p. 1013) .

Image 7: Front view of the skater after cropping.



5.319

Transparent pixels

This is the image that appears as the center ice skater in Image 5 (p. 1009) after green-screen processing. Note that all of the green pixels in Image 7 (p. 1013) appear to be transparent in Image 5 (p. 1009) .

Remainder of the run method

Continuing with the **run** method, Listing 4 (p. 1014) calls the method named **greenScreenDraw** three times in succession to draw the three skater images at specific locations on the snow scene with green-screen processing.

Listing 4: Remainder of the run method.

```
    //Draw the front view of the skater on the snowScene
  // at full size.
  greenScreenDraw(front,snowScene,117,0);

  //Draw the left side view of the skater on the
  // snowScene at half size.
  left = left.getPictureWithHeight(256/2);
  greenScreenDraw(left,snowScene,55,64);

  //Draw the right side view of the skater on the
  // snowScene at one-third size.
  right = right.getPictureWithHeight(256/3);
  greenScreenDraw(right,snowScene,260,96);

  //Display students name on the final output and
  // display it.
  snowScene.addMessage("Display your name here.",10,15);
  snowScene.explore();
  System.out.println(snowScene);
}//end run method
```

5.320

Two skater images are scaled

In two cases in Listing 4 (p. 1014) , the method named **getPictureWithHeight** is called before calling the **greenScreenDraw** method. The **getPictureWithHeight** method is used to scale two of the images to a smaller size as shown in Image 5 (p. 1009) .

The getPictureWithHeight method

The **getPictureWithHeight** method is defined in Ericson's **SimplePicture** class and inherited into Ericson's **Picture** class.

The method requires a single integer input parameter, which specifies the height in pixels of a scaled output version of the picture object on which the method is called.

According to the documentation, the method can be used to create a new picture with the specified height. The aspect ratio of the width and height will stay the same.

Original height is 256 pixels

Referring back to the parameters that were passed to the **crop** method in Listing 3 (p. 1012) , you can see the height of all three cropped images is 256 pixels.

Scale by 1/2 and 1/3

Referring to the two calls to the **getPictureWithHeight** method in Listing 4 (p. 1014) , you can see that one of the cropped images was replaced with an image having a height of 256/ 2 or 128 pixels. The other cropped image was replaced with an image having a height of 256/3 or 85 pixels. You can see the effect of this scaling in Image 5 (p. 1009) .

The height of one of the images was not changed, which you can also see in Image 5 (p. 1009) .

Put the run method on hold

I will put the explanation of the **run** method on temporary hold at this point and explain the method named **greenScreenDraw** , which is shown in Listing 5 (p. 1016) .

Behavior of the greenScreenDraw method

The **greenScreenDraw** method requires four incoming parameters:

- A reference to a source image.
- A reference to a destination image
- The horizontal coordinate on the destination image where the upper-left corner of the source image will be drawn.
- The vertical coordinate on the destination image where the upper-left corner of the source image will be drawn.

Pure green pixels are required for transparency

The method draws the source image onto the destination image at the specification location. However, pixels having a pure green color are not drawn on the destination image. The effect is to make it appear that those portions of the source image with pure green pixels become totally transparent allowing the pixels belonging to the destination image to show through.

jpg image files are not satisfactory for this program

Picture objects created from jpg image files typically won't have a pure green background even if they had a pure green background before being compressed into the jpg format file. However, the bmp file format does not corrupt the pixel colors. Therefore, bmp images work well for this type of processing.

The greenScreenDraw method

The **greenScreenDraw** method is shown in its entirety in Listing 5 (p. 1016) .

Listing 5: The greenScreenDraw method.

```
  private void greenScreenDraw(
                       Picture source,
                       Picture dest,
                       //Place the upper-left corner
                       // of the source image at the
                       // following location in the
                       // destination image.
                       int destX,
                       int destY){
  int width = source.getWidth();
  int height = source.getHeight();
  Pixel pixel = null;
  Color color = null;

  for(int row = 0;row < height;row++){
    for(int col = 0;col < width;col++){
      color = source.getPixel(col,row).getColor();
      if(!(color.equals(Color.GREEN))){
        pixel = dest.getPixel(destX + col,destY + row);
        pixel.setColor(color);
      }//end if
    }//end inner loop
  }//end outer loop

}//end greenScreenDraw
```

5.321

Very similar to earlier methods

This method is very similar to other methods that I have explained in earlier modules that use nested **for** loops to draw one image onto another image.

The one new thing...

The only thing that is really new in Listing 5 (p. 1016) is the **if** statement that tests the color of source image pixels for a value of exactly **Color.GREEN** . If the color of the source pixel does not match that color exactly, it is drawn on the destination image replacing the pixel color previously at that location on the destination image.

If the source pixel color does exactly match that color, it is not drawn on the destination image thereby leaving the color of the corresponding destination pixel unchanged.

Listing 5 (p. 1016) signals the end of the **greenScreenDraw** method.

The weather forecast on television

This is roughly how the TV stations superimpose a human weather forecaster onto a giant animated weather map. The forecaster is photographed with a video camera standing in front of a green or blue screen. At the same time, an animated video of the weather map is also created.

Then each video frame of the forecaster is superimposed onto a video frame of the weather map. The green or blue pixels in the forecaster frame are not copied onto the weather map frame. This allows the

weather map pixels to show with the exception of those that are replaced by the pixels that comprise the human forecaster. *(The forecaster must be careful to avoid wearing clothing that matches the color of the green or blue screen.)*

Returning to the run method

When the third call to the **greenScreenDraw** method returns in Listing 4 (p. 1014) , the **run** method:

- Adds the student's name to the snow scene picture.
- Displays the snow scene picture (see Image 5 (p. 1009)).
- Displays information about the snow scene picture on the command line screen.

The end of the program

Then the **run** method in Listing 4 (p. 1014) returns control to the **main** method in Listing 1 (p. 1011) causing the program to terminate.

5.3.25.6 Run the program

I encourage you to copy the code from Listing 6 (p. 1019) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click the following links to download the required input images:

1. Prob03a.bmp [156]
2. Prob03b.bmp [157]
3. Prob03c.bmp [158]
4. Prob03d.jpg [159]

5.3.25.7 Summary

In this module, you learned how to write a program to do *green-screen* processing to superimpose a source image onto a destination image while making the green background of the source image appear to be transparent.

5.3.25.8 What's next?

You will learn how to darken, brighten, and tint the colors in a **Picture** object in the next module.

5.3.25.9 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 08 [160]
  - Part01 [161]
  - Part02 [162]
  - Part03 [163]

[156] http://cnx.org/content/m44210/latest/Prob03a.bmp
[157] http://cnx.org/content/m44210/latest/Prob03b.bmp
[158] http://cnx.org/content/m44210/latest/Prob03c.bmp
[159] http://cnx.org/content/m44210/latest/Prob03d.jpg
[160] http://www.youtube.com/playlist?list=PLF12CDA1C72ACC167
[161] http://www.youtube.com/watch?v=EW6ZEDGJi2w
[162] http://www.youtube.com/watch?v=JqK_42UnXoI
[163] http://www.youtube.com/watch?v=4bM6qElbxpc

5.3.25.10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Green-Screen Processing
- File: Java3016.htm
- Published: 08/01/12
- Revised: 02/15/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.25.11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 6 (p. 1019) below.

Listing 6: Complete program listing.

```
    /*File Prob03 Copyright 2008 R.G.Baldwin
Revised 12/17/08
*******************************************************/
import java.awt.Color;
public class Prob03{
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
  }//end main
}//end class Prob03
//=====================================================//

class Prob03Runner{

  public Prob03Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//

  public void run(){
    //Instantiate, display and crop the four input
    // images. They must be cropped to remove the Alice
    // runtime window material. The three skater images
    // are also cropped to remove excess blank green
    // background.

    //Note that the three views of the skater are bmp
    // images instead of jpg images in order to preserve
    // the pure green background color. Storing the
    // images as jpg files would corrupt the background
    // color in the low order bits.
    //A view facing the front of the skater.
    Picture front = new Picture("Prob03a.bmp");
    front.explore();
    front = crop(front,123,59,110,256);

    //A view showing the right side of the skater.
    Picture right = new Picture("Prob03b.bmp");
    right.explore();
    right = crop(right,123,59,110,256);

    //A view showing the left side of the skater.
    Picture left = new Picture("Prob03c.bmp");
    left.explore();
    left = crop(left,123,59,110,256);

    //This will be the background for the new picture.
    Picture snowScene = new Picture("Prob03d.jpg");
    snowScene.explore();
    snowScene = crop(snowScene,6,59,344,256);
```



```
    //Draw the front view of the skater on the snowScene
    // at full size.
    greenScreenDraw(front,snowScene,117,0);
```

-end-

5.3.26 Java3016r Review[164]

5.3.26.1 Table of Contents

5.3.26.2 Preface

This module contains review questions and answers keyed to the module titled Java3016: Green-Screen Processing [165] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.26.3 Questions

5.3.26.3.1 Question 1 .

Given Image 1 (p. 1023) , Image 2 (p. 1024) , Image 3 (p. 1025) , and Image 4 (p. 1026) , which of the following output images is produced by the code in Listing 1 (p. 1022) ?

A. Image 5 (p. 1027)
B. Image 6 (p. 1028)

[164] This content is available online at <http://cnx.org/content/m45781/1.1/>.
[165] http://cnx.org/content/m44210

Listing 1. Question 1.

```
/*File Java3016ra Copyright 2013 R.G.Baldwin
Revised 02/15/13
*******************************************************/
import java.awt.Color;
public class Java3016ra{
  public static void main(String[] args){
    new Java3016raRunner().run();
  }//end main
}//end class Java3016ra
//=====================================================//

class Java3016raRunner{

  public void run(){

    Picture snowScene = crop(
                new Picture("Prob03d.jpg"),6,59,344,256);

    greenScreenDraw(crop(new Picture("Prob03a.bmp"),
                         123,59,110,256),snowScene,117,0);
    greenScreenDraw(
          crop(new Picture("Prob03c.bmp"),123,59,110,256).
             getPictureWithHeight(256/2),snowScene,55,64);
    greenScreenDraw(
          crop(new Picture("Prob03b.bmp"),123,59,110,256).
            getPictureWithHeight(256/3),snowScene,260,96);

    snowScene.explore();

  }//end run method
  //-----------------------------------------------------//

  private void greenScreenDraw(Picture source,
                               Picture dest,
                               int destX,
                               int destY){
    Pixel pixel = null;
    Color color = null;

    for(int row = 0;row < source.getHeight();row++){
      for(int col = 0;col < source.getWidth();col++){
        color = source.getPixel(col,row).getColor();
        if((!(color.equals(Color.GREEN)))&&
          (!(color.getRed() < 40))){
          dest.getPixel(destX + col,destY + row).
                                          setColor(color);
        }//end if
      }//end inner loop
    }//end outer loop
  }//end greenScreenDraw
  //-----------------------------------------------------//

  private Picture crop(Picture pic,int startCol,
                                  int startRow,
```

Image 1. Prob03a.bmp.

5.324

Image 2. Prob03b.bmp.

5.325

Image 3. Prob03c.bmp.

5.326

Image 4. Prob03d.jpg.

5.327

Image 5. Possible output image.

5.328

Image 6. Possible output image.

5.329

Answer 1 (p. 1030)

5.3.26.4 Images

- Image 1 (p. 1023) . Prob03a.bmp.
- Image 2 (p. 1024) . Prob03b.bmp.
- Image 3 (p. 1025) . Prob03c.bmp.
- Image 4 (p. 1026) . Prob03d.jpg.
- Image 5 (p. 1027) . Possible output image.

- Image 6 (p. 1028) . Possible output image.

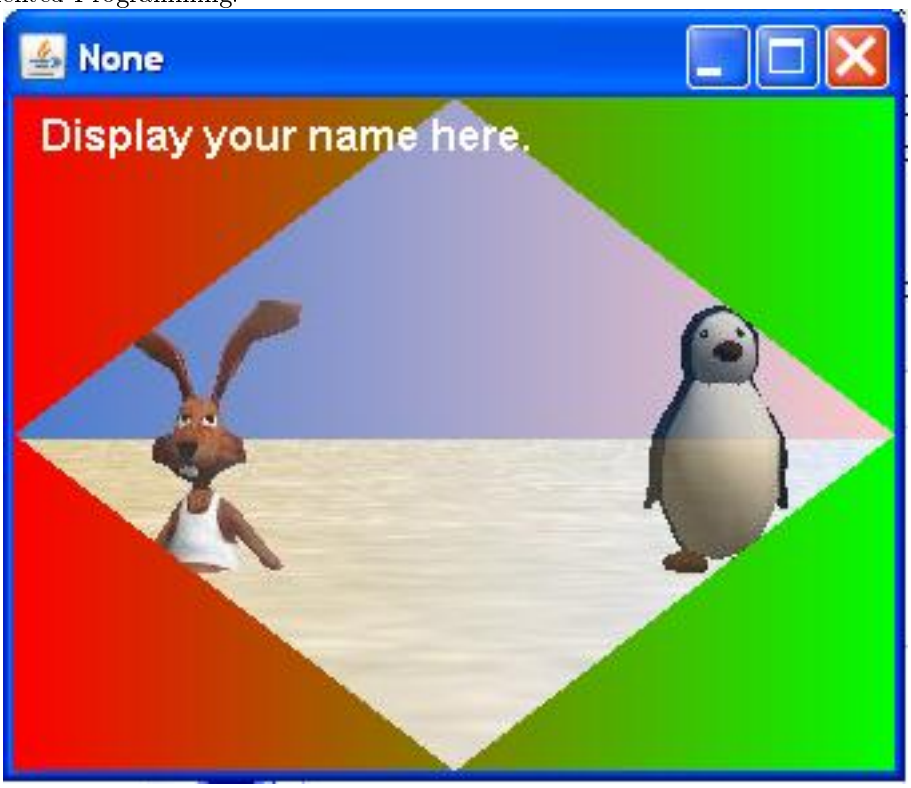
5.3.26.5 Listings

- Listing 1 (p. 1022) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.26.6 Answers

5.3.26.6.1 Answer 1

The code in Listing 1 (p. 1022) produces the output image shown in Image 6 (p. 1028) .
Back to Question 1 (p. 1021)

5.3.26.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3016r Review
- File: Java3016r.htm
- Published: 02/17/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.27 Java3016s Slides[166]

5.3.27.1 Table of Contents

5.3.27.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3016: Green-Screen Processing [167] .

Click here [168] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.27.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3016s Slides
- File: Java3016s.htm
- Published: 01/06/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[166] This content is available online at <http://cnx.org/content/m45637/1.2/>.
[167] http://cnx.org/content/m44210
[168] http://cnx.org/content/m45637/latest/a0-Index.htm

5.3.28 Java3018: Darkening, Brightening, and Tinting the Colors in a Picture[169]

5.3.28.1 Table of Contents

5.3.28.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [170] .

5.3.28.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.28.2.1.1 Images

5.3.28.2.1.2 Listings

[169] This content is available online at <http://cnx.org/content/m44234/1.5/>.
[170] http://cnx.org/content/m44148/latest/

- Listing 7 (p. 1043) . The else clause in the processing loop.
- Listing 8 (p. 1044) . Apply a red tint to the skater.
- Listing 9 (p. 1046) . The remainder of the run method.
- Listing 10 (p. 1049) . Complete program listing.

5.3.28.3 Preview

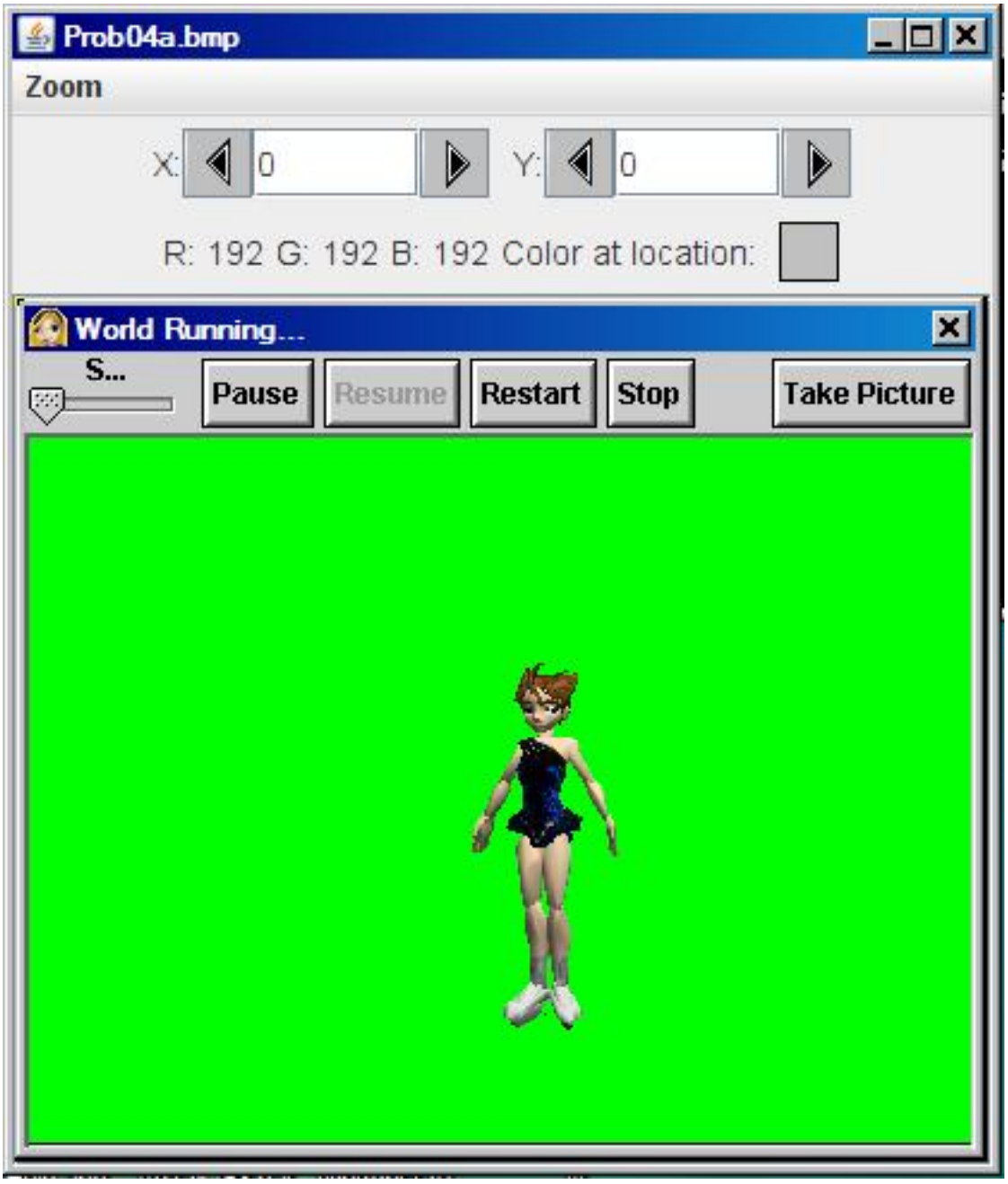
In this module, you will learn how to darken, brighten, and tint the colors in a **Picture** object.

Program specifications

Write a program named **Prob04** that uses the class definition shown in Listing 1 (p. 1038) and Ericson's media library along with the image files in the following list to produce the four graphic output images shown in Image 1 (p. 1034) through Image 4 (p. 1037) .

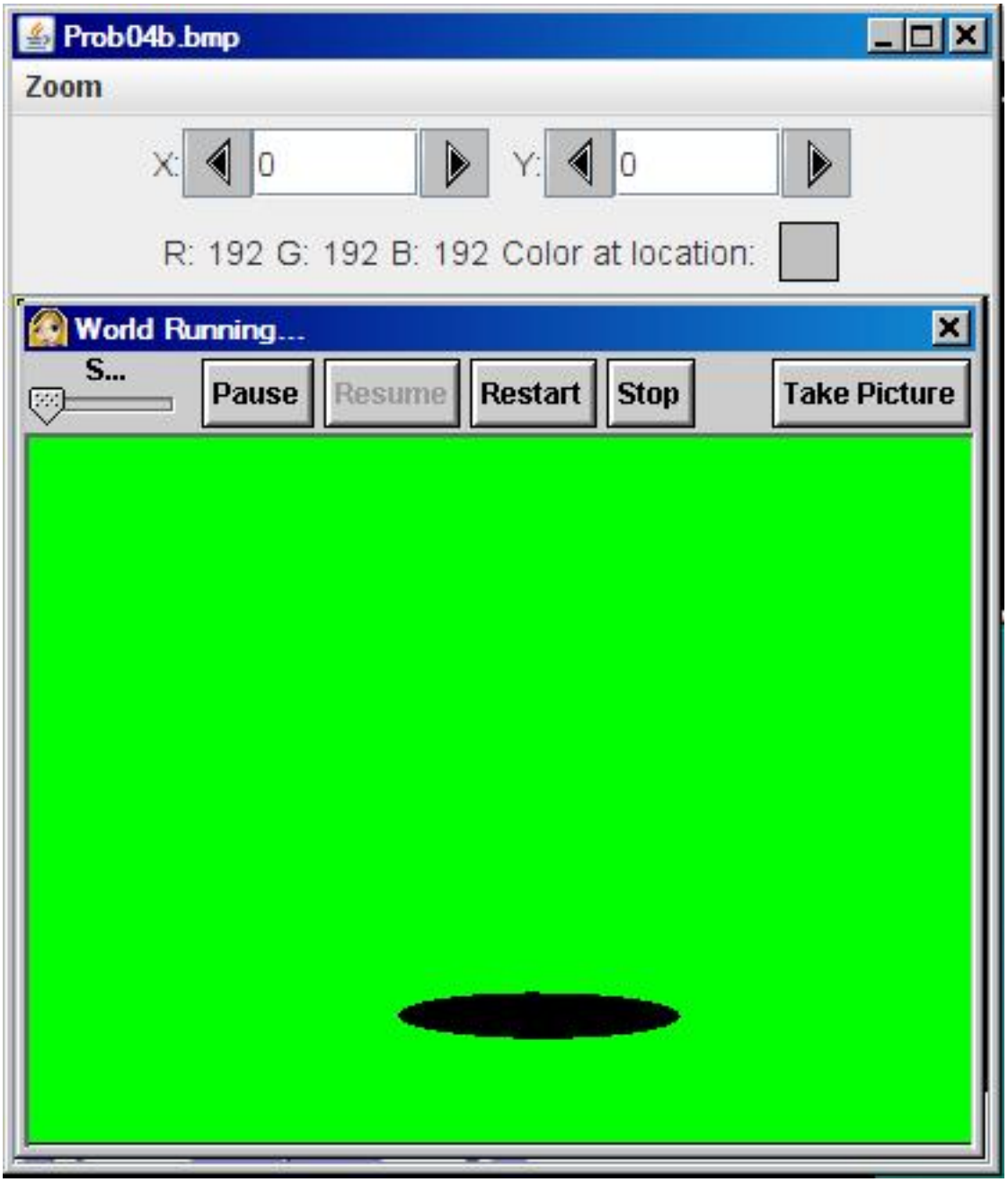
- Prob04a.bmp
- Prob04b.bmp
- Prob04c.jpg

Image 1: Input file Prob04a.bmp.



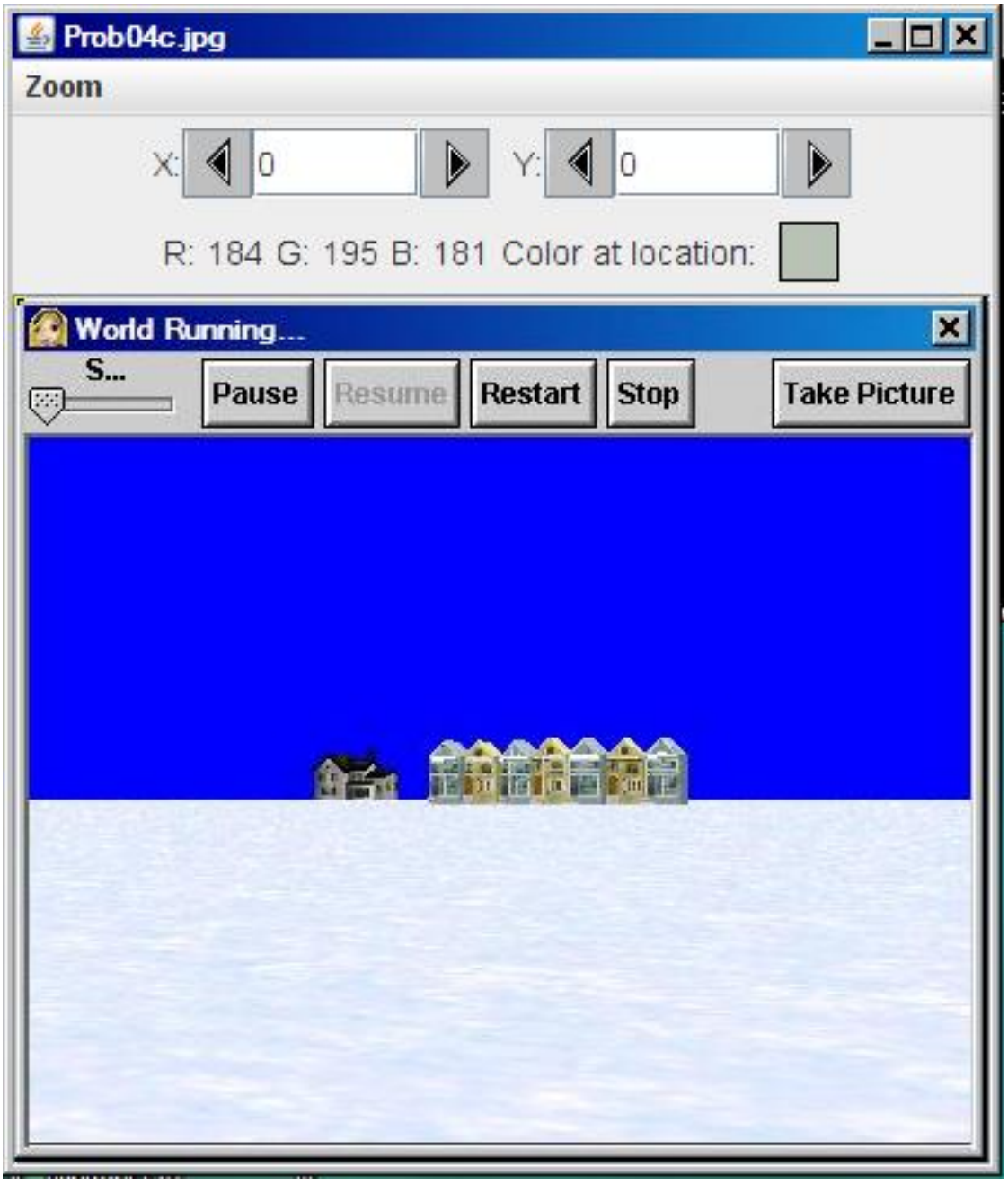
5.330

Image 2: Input file Prob04b.bmp.



5.331

Image 3: Input file Prob04c.jpg.



5.332

Image 4: Required output image.

5.333

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob04** shown in Listing 1 (p. 1038) .

Required text output

In addition to the four output images mentioned above, your program must display your name and the other line of text shown in Image 5 (p. 1038) on the command-line screen.

Image 5: Required text output.

```
    Display your name here.
Picture, filename None height 293 width 392
```

5.334

5.3.28.4 General background information

This program uses a black ellipse on a green background *(see Image 2 (p. 1035))* as a pattern to cause the pixels in a snow scene *(see Image 3 (p. 1036))* at the location of the ellipse to be given a red tint and causes all other pixels in the snow scene to be darkened *(see Image 4 (p. 1037))* .

The program also causes a red skater in a green background *(see Image 1 (p. 1034))* to be given a red tint and then drawn on the snow scene at the location of the red-tinted ellipse. The effect is that of a spotlight with a red tint shining on the skater *(see Image 4 (p. 1037))* .

5.3.28.5 Discussion and sample code

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 10 (p. 1049) near the end of the module.

The driver class named Prob04

The driver class containing the **main** method is shown in Listing 1 (p. 1038) .

Listing 1: The driver class named Prob04.

```
    import java.awt.Color;

public class Prob04{
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    obj.run();
  }//end main
}//end class Prob04
```

5.335

As has been the case in several earlier modules, the code in the **main** method instantiates an object of the class named **Prob04Runner** and calls the **run** method on that object.

When the **run** method returns, the **main** method terminates causing the program to terminate.

Beginning of the class named Prob04Runner

The class named **Prob04Runner** begins in Listing 2 (p. 1039) .

Listing 2: Beginning of the class named Prob04Runner.

```
  class Prob04Runner{

public Prob04Runner(){//constructor
  System.out.println("Display your name here.");
}//end constructor
```

5.336

Listing 2 (p. 1039) shows the constructor for the class, which simply displays the student's name on the command line screen as shown in Image 5 (p. 1038) .

Beginning of the run method

The beginning of the **run** method, which is called in Listing 1 (p. 1038) , is shown in Listing 3 (p. 1039) .

Listing 3: Beginning of the run method.

```
    public void run(){

  Picture skater = new Picture("Prob04a.bmp");
  skater.explore();
  skater = crop(skater,6,59,392,293);

  Picture hole = new Picture("Prob04b.bmp");
  hole.explore();
  hole = crop(hole,6,59,392,293);


  Picture snowScene = new Picture("Prob04c.jpg");
  snowScene.explore();
  snowScene = crop(snowScene,6,59,392,293);
```

5.337

Instantiate and display three Picture objects

The code in Listing 3 (p. 1039) instantiates **Picture** objects from the three image files and displays those pictures in Image 1 (p. 1034) , Image 2 (p. 1035) , and Image 3 (p. 1036) .

Crop the pictures

Note that the images in those three pictures contain the **Alice** [171] runtime window controls as a banner that reads **World Running...** and a line of buttons.

The method named crop

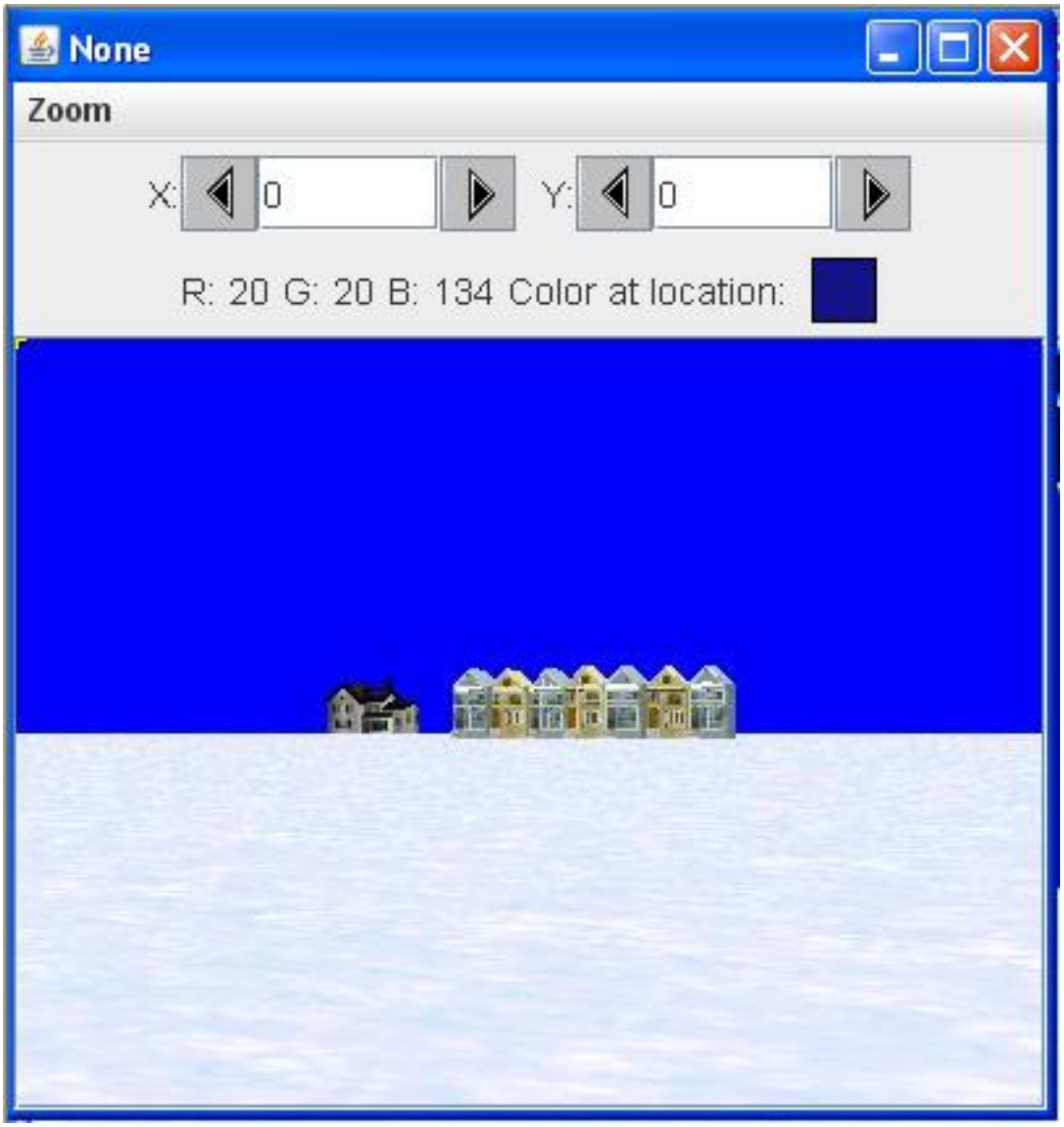
[171] http://www.alice.org/

Listing 3 (p. 1039) calls a method named **crop** on all three **Picture** objects to eliminate the **Alice** runtime controls. Note that the same rectangular area is preserved for all three images. Thus, all three images are the same size after cropping.

The cropped snow scene image

If you were to display the picture whose reference is stored in the variable named **snowScene** after the **crop** method returns, you would see the image shown in Image 6 (p. 1040) with the **Alice** runtime controls no longer visible.

Image 6: Cropped version of the snow scene image.



5.338

The crop method

The method named **crop** that was used to crop the three pictures is essentially the same as the cropping methods that I explained in earlier modules. Therefore, I won't repeat that explanation here. You can view the **crop** method in its entirety in Listing 10 (p. 1049) near the end of the module.

Darken the background of the snow scene

Listing 4 (p. 1041) calls the method named **darkenBackground** to make all of the pixels darker in the snow scene except for those in the location of the ellipse as shown in Image 4 (p. 1037) . The pixels at the location of the ellipse are given a red tint.

Listing 4: Darken the background of the snow scene.

```
darkenBackground(hole,snowScene);
```

5.339

Put the run method on temporary hold

I will put the explanation of the **run** method on hold at this point and explain the method named **darkenBackground** .

The method named darkenBackground

The method named **darkenBackground** receives references to two **Picture** objects as parameters. It uses the first picture as a *pattern* from which it determines which pixels in the second picture *(destination)* should be darkened.

The *pattern* and *destination* images

In this case, a cropped version of the image of the black ellipse shown in Image 2 (p. 1035) is the pattern. The cropped image of the snow scene shown in Image 6 (p. 1040) is the destination image whose pixels will be darkened.

Assumptions

The **darkenBackground** method assumes that the pattern image has a pure green background as shown in Image 2 (p. 1035) . It also assumes that the pattern and the destination have the same dimensions.

Behavior of the method

The **darkenBackground** method darkens every pixel in the destination that is at the location of a green pixel in the pattern.

The method applies a red tint to every pixel in the destination that is at the location of a non-green pixel in the pattern

Beginning of the darkenBackground method

The **darkenBackground** method begins in Listing 5 (p. 1042) .

Listing 5: Beginning of the darkenBackground method.

```
 private void darkenBackground(Picture pattern,
                               Picture dest){

Pixel[] patternPixels = pattern.getPixels();
Pixel[] destPixels = dest.getPixels();
Color color = null;
int red = 0;
int green = 0;
int blue = 0;
```

5.340

Get two arrays of pixel data

The **darkenBackground** method begins by calling the **getPixels** method on each of the picture objects to create a pair of array objects containing pixel data.

You learned how to use the **getPixels** method in an earlier module. Recall that this approach is useful when you don't need to be concerned about the locations of the pixels in an x-y coordinate system.

Arrays have the same length

Because the two pictures have the same dimensions, the two arrays have the same length.

A given array index specifies pixel data from the same location in both pictures.

Beginning of the processing loop

The method uses a **for** loop to traverse the two arrays of pixel data in parallel, using information from the *pattern* picture to make the color changes to the *destination* picture described above. The **for** loop begins in Listing 6 (p. 1042) .

Listing 6: Beginning of the processing loop.

```
  for(int cnt = 0;cnt < patternPixels.length;cnt++){
color = patternPixels[cnt].getColor();
if(color.equals(Color.GREEN)){
  //Darken corresponding pixel in the destination.
  color = destPixels[cnt].getColor();
  destPixels[cnt].setColor(color.darker());
```

5.341

Behavior of the processing loop

The loop begins by getting the color value of the next pixel in the pattern array. If the color of the pattern pixel is green, the code in Listing 6 (p. 1042) :

- Gets the color from the corresponding pixel in the destination array.

- Calls the method named **darker** , which is a method of the **Color** class in the standard Sun library, to produce a darker version of the pixel color.
- Replaces the pixel color in the destination array with the darker version of the color.

Compare images to see the results

If you compare Image 4 (p. 1037) with Image 3 (p. 1036) , you will see that *(ignoring the skater and the ellipse)* , all of the pixels in Image 4 (p. 1037) are darker versions of the colors in Image 3 (p. 1036) .

The brighter method

The **Color** class also provides a method named **brighter** which has the opposite effect. In particular, it can be used to brighten the color of a pixel.

These two methods are very useful for making a pixel darker or brighter without having to know anything about the actual color of the pixel.

The else clause in the processing loop

If the color of the pattern pixel *(tested in Listing 6 (p. 1042))* is not green, the code in the **else** clause in Listing 7 (p. 1043) is executed.

Listing 7: The else clause in the processing loop.

```
      }else{
    //Apply a red tint to the corresponding pixel in
    // the destination.
    color = destPixels[cnt].getColor();
    red = color.getRed();
    if(red*1.25 < 255){
      red = (int)(red * 1.25);
    }else{
      red = 255;
    }//end else
    green = (int)(color.getGreen() * 0.8);
    blue = (int)(color.getBlue() * 0.8);
    destPixels[cnt].setColor(new Color(red,green,blue));
  }//end else
 }//end for loop

}//end darkenBackground
```

5.342

The snow scene and the ellipse only...

At this point, only the images from Image 2 (p. 1035) *(the ellipse)* and Image 3 (p. 1036) *(the snow scene)* are being processed. The image of the skater in Image 1 (p. 1034) hasn't entered the picture yet.

Application of the ellipse pattern

The code in Listing 7 (p. 1043) is executed only if the pixel from the destination picture is at a location that matches one of the non-green *(black)* pixels in the ellipse in Image 2 (p. 1035) .

(The fact that the pixel is black is of no consequence. The only thing that matters is that it is not green.)

The objective of the else clause

The objective is to modify the pixel color in the destination picture at this location to give it a red tint as shown in Image 4 (p. 1037) .

Get, save, and modify the red color value

Listing 7 (p. 1043) begins by getting the color of the pixel from the current location in the destination picture. It extracts and saves the red color value of the pixel. Then, depending on the current value of the red color value, it either:

- Multiplies the red color value by a factor of 1.25, or
- Sets the red color value to the maximum allowable value of 255.

Decrease the color values for blue and green

Following this, it gets the green and blue color values and multiplies each of them by 0.8.

Replace the old pixel color with the new color

Finally, it replaces the pixel color with a new color using the modified values of red, green, and blue.

The texture is preserved

As you can see in Listing 4 (p. 1041) , this process causes the pixels in locations that match the ellipse to take on a red tint, but the texture of the image is not destroyed as it would be if the pixels had simply been replaced by pixels that all have the same color of pink.

The end of the darkenBackground method

Listing 7 (p. 1043) signals the end of the processing loop and the end of the **darkenBackground** method.

Apply a red tint to the skater

Returning to where we left off in the **run** method in Listing 4 (p. 1041) , the code in Listing 8 (p. 1044) calls a method named **redTint** , passing a reference to the picture that contains a cropped image of the skater. The method applies a red tint to the skater.

Listing 8: Apply a red tint to the skater.

```
redTint(skater);
```

5.343

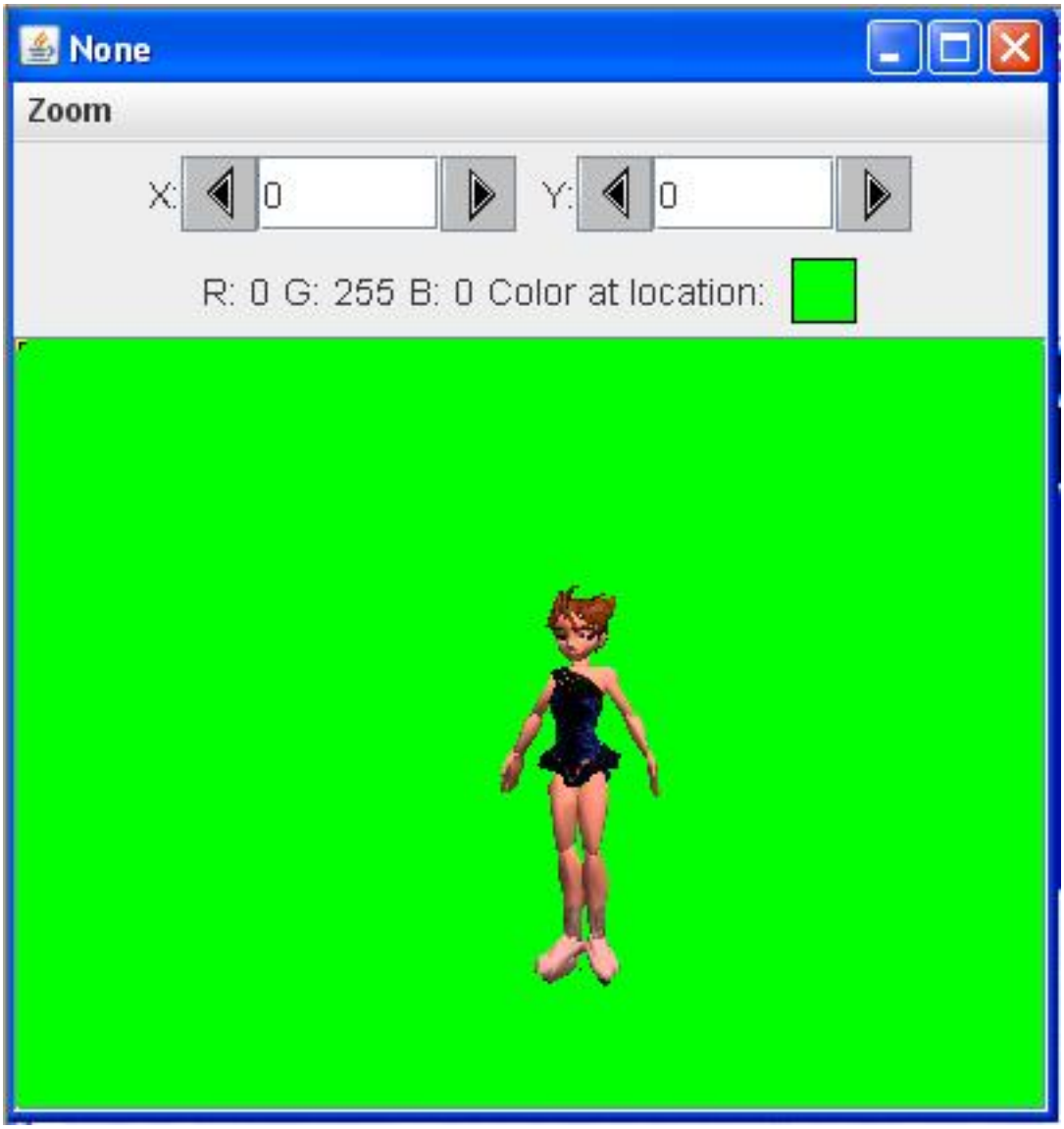
The **redTint** method assumes that the image being processed has a pure green background like that shown in Image 1 (p. 1034) . The method applies an algorithm very similar to that shown in Listing 7 (p. 1043) to apply a red tint to every pixel that is not pure green.

Because of the similarity of the code in the **redTint** method and the code in Listing 7 (p. 1043) , a detailed explanation of the **redTint** method should not be required. You can view the method in its entirety in Listing 10 (p. 1049) near the end of the module.

The skater with the red tint applied

If you were to display the picture referred to by **skater** immediately after the **redTint** method returns in Listing 8 (p. 1044) , you would see the image shown in Image 7.

Image 7: The skater with a red tint applied.



5.344

Compare Image 7 with Image 1

You can see the effect of applying the red tint process to the skater by comparing Image 7 (p. 1045) with Image 1 (p. 1034) . Note that the process does not change the color of the green pixels.

The remainder of the run method

Continuing with the **run** method, Listing 9 (p. 1046) calls a method named **greenScreenDraw** to draw the cropped, red-tinted skater on the snow scene as shown in Image 4 (p. 1037) .

Listing 9: The remainder of the run method.

```
    //Draw the skater on the snowScene.
  greenScreenDraw(skater,snowScene,0,0);

  //Display students name on the final output and
  // display it.
  snowScene.addMessage("Display your name here.",10,15);

  snowScene.explore();
  System.out.println(snowScene);

}//end run method
```

5.345

Behavior of the method

The **greenScreenDraw** method copies all non-green pixels from a source image to a destination image at a specified location. This method is very similar to methods that I have explained in earlier modules. Therefore, an explanation of the method in this module should not be needed.

You can view the **greenScreenDraw** method in its entirety in Listing 10 (p. 1049) near the end of the module.

Add text and display the final output image

When the **greenScreenDraw** method returns, Listing 9 (p. 1046) calls the **addMessage** method to display the student's name on the snow scene and then calls the **explore** method to produce the output image shown in Image 4 (p. 1037) . None of that should be new to you at this point.

Display text and return

Finally, Listing 9 (p. 1046) displays some information on the command line screen as shown in Image 5 (p. 1038) and returns control to the **main** method in Listing 1 (p. 1038) .

Terminate the program

Having nothing more to do, the **main** method terminates, causing the program to terminate and return control to the operating system.

5.3.28.6 Run the program

I encourage you to copy the code from Listing 10 (p. 1049) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click the following links to download the required input files:

- Prob04a.bmp [172]
- Prob04b.bmp [173]
- Prob04c.jpg [174]

[172] http://cnx.org/content/m44234/latest/Prob04a.bmp
[173] http://cnx.org/content/m44234/latest/Prob04b.bmp
[174] http://cnx.org/content/m44234/latest/Prob04c.jpg

5.3.28.7 Summary

In this module, you learned how to darken, brighten, and tint the colors in a **Picture** object.

5.3.28.8 What's next?

You will probably learn more than you already know about interfaces, arrays of type Object, etc., in the next module.

5.3.28.9 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 09 [175]
  - Part01 [176]
  - Part02 [177]
  - Part03 [178]

5.3.28.10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Darkening, Brightening, and Tinting the Colors in a Picture
- File: Java3018.htm
- Published: 08/01/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[175] http://www.youtube.com/playlist?list=PL11F9AC688AC89E56
[176] http://www.youtube.com/watch?v=T-pcmz5XmQY
[177] http://www.youtube.com/watch?v=T16JMwpBIUI
[178] http://www.youtube.com/watch?v=aXLKqYA_0Ng

5.3.28.11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 1049) below.

Listing 10: Complete program listing.

```
    /*File Prob04 Copyright 2008 R.G.Baldwin
*********************************************************/

import java.awt.Color;
public class Prob04{
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    obj.run();
  }//end main
}//end class Prob04
//=======================================================//

class Prob04Runner{

  public Prob04Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//

  public void run(){

    Picture skater = new Picture("Prob04a.bmp");
    skater.explore();
    skater = crop(skater,6,59,392,293);

    Picture hole = new Picture("Prob04b.bmp");
    hole.explore();
    hole = crop(hole,6,59,392,293);


    Picture snowScene = new Picture("Prob04c.jpg");
    snowScene.explore();
    snowScene = crop(snowScene,6,59,392,293);

    //Make all the pixels darker in the snow scene except
    // for those in the location of the hole. Make them
    // brighter.
    darkenBackground(hole,snowScene);

    //Apply a red tint to the skater
    redTint(skater);

    //Draw the skater on the snowScene.
    greenScreenDraw(skater,snowScene,0,0);

    //Display students name on the final output and
    // display it.
    snowScene.addMessage("Display your name here.",10,15);

    snowScene.explore();
    System.out.println(snowScene);

  }//end run method
  //-----------------------------------------------------//
```

-end-

5.3.29 Java3018r Review[179]

5.3.29.1 Table of Contents

5.3.29.2 Preface

This module contains review questions and answers keyed to the module titled Java3018: Darkening, Brightening, and Tinting the Colors in a Picture [180] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.29.3 Questions

5.3.29.3.1 Question 1 .

Given Image 1 (p. 1053) , Image 2 (p. 1054) , and Image 3 (p. 1055) , which of the following output images is produced by the code in Listing 1 (p. 1052) ?

A. Image 4 (p. 1056)
B. Image 5 (p. 1057)

[179] This content is available online at <http://cnx.org/content/m45782/1.1/>.
[180] http://cnx.org/content/m44234

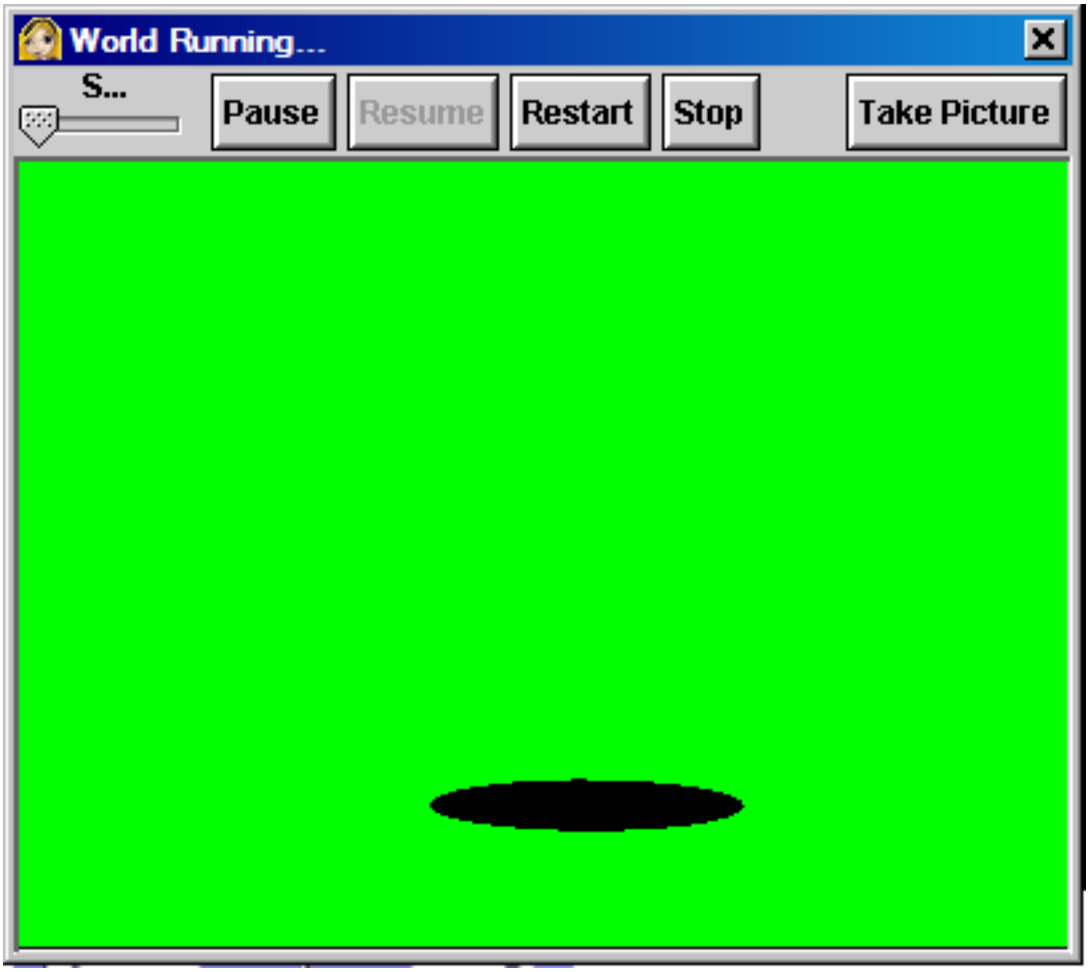
Listing 1. Question 1.

```
/*File Java3018ra Copyright 2013 R.G.Baldwin
*********************************************************/

import java.awt.Color;
public class Java3018ra{
  public static void main(String[] args){
    new Java3018raRunner().run();
  }//end main
}//end class Java3018ra
//=======================================================//

class Java3018raRunner{

  public Java3018raRunner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//

  public void run(){

    Picture skater = crop(new Picture("Prob04a.bmp"),6,59,392,293);
    Picture hole = crop(new Picture("Prob04b.bmp"),6,59,392,293);
    Picture snowScene = crop(new Picture("Prob04c.jpg"),6,59,392,293);
    processBackground(hole,snowScene);
    redTint(skater);
    greenScreenDraw(skater,snowScene,0,0);
    snowScene.explore();

  }//end run method
  //-----------------------------------------------------//

  private void redTint(Picture pic){
    Pixel[] pixels = pic.getPixels();
    Color color = null;
    int red = 0;
    int green = 0;
    int blue = 0;
    for(int cnt = 0;cnt < pixels.length;cnt++){
      color = pixels[cnt].getColor();
      //Apply a red tint to all non-green pixels
      if(!(color.equals(Color.GREEN))){
        //Increase the value of the red component
        red = color.getRed();
        if(red*1.25 < 255){
          red = (int)(red * 1.25);
        }else{
          red = 255;
        }//end else
        //Decrease the value of blue and green
        green = (int)(color.getGreen()*0.8);
        blue = (int)(color.getBlue()*0.8);

        //Apply the new color to the pixel.
        pixels[cnt].setColor(new Color(red,green,blue));
```

Image 1. Prob04a.bmp.

5.348

Image 2. Prob04b.bmp.



5.349

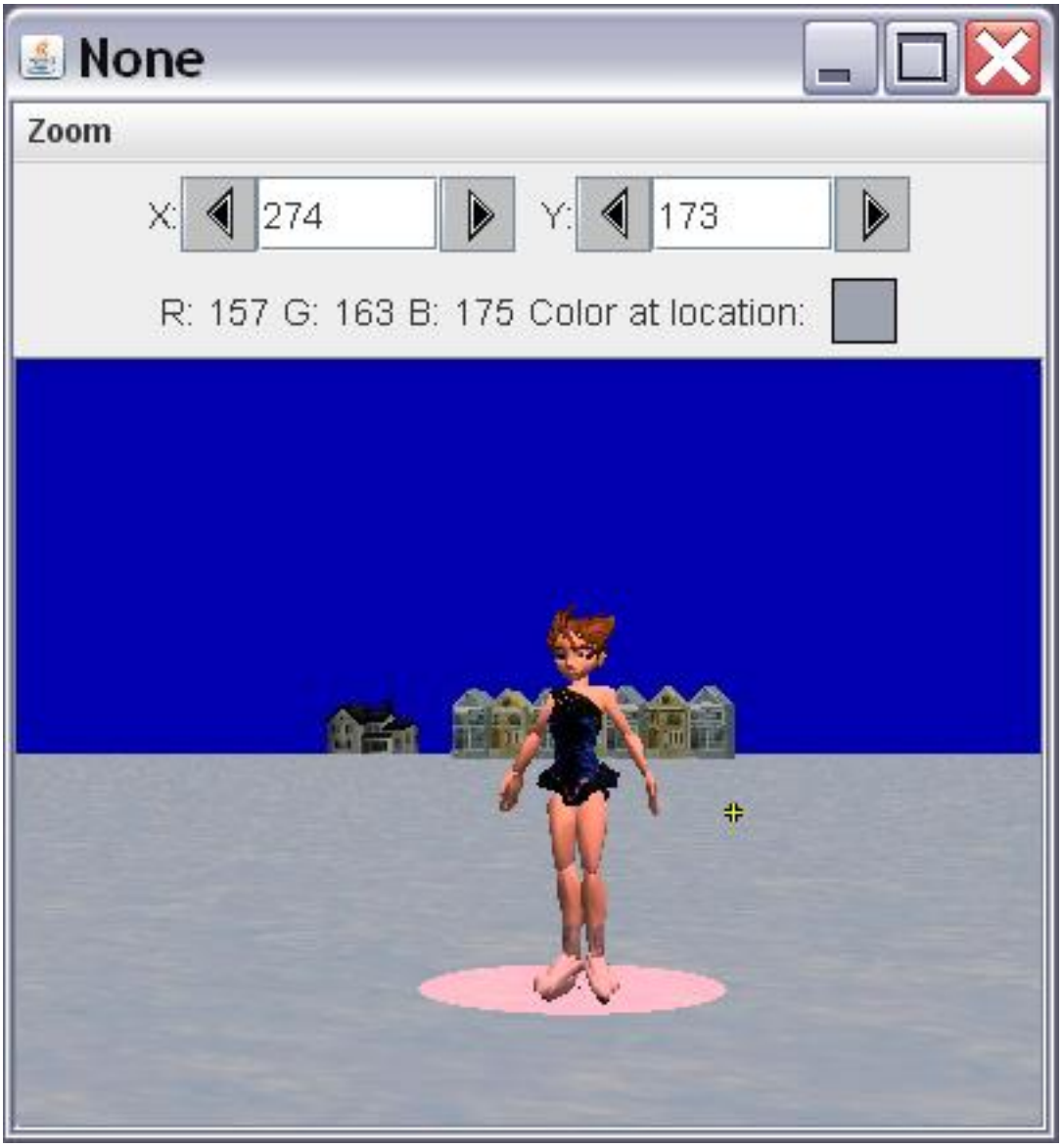
Image 3. Prob04c.jpg.

5.350

Image 4. Possible output image.

5.351

Image 5. Possible output image.



5.352

Answer 1 (p. 1059)

5.3.29.4 Images

- Image 1 (p. 1053) . Prob04a.bmp.
- Image 2 (p. 1054) . Prob04b.bmp.

- Image 3 (p. 1055) . Prob04c.jpg.
- Image 4 (p. 1056) . Possible output image.
- Image 5 (p. 1057) . Possible output image.

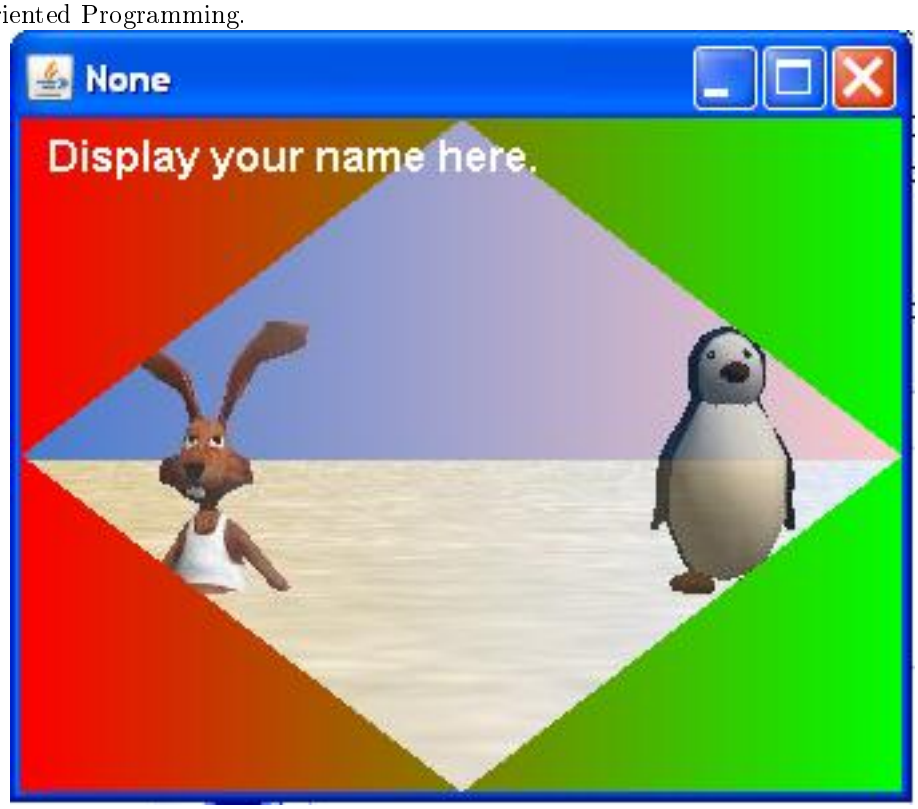
5.3.29.5 Listings

- Listing 1 (p. 1052) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.29.6 Answers

5.3.29.6.1 Answer 1

The code in Listing 1 (p. 1052) produces the output image shown in Image 4 (p. 1056) .
Back to Question 1 (p. 1051)

5.3.29.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3018r Review
- File: Java3018r.htm
- Published: 02/17/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.30 Java3018s Slides[181]

5.3.30.1 Table of Contents

5.3.30.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3018: Darkening, Brightening, and Tinting the Colors in a Picture [182] .

Click here [183] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.30.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3018s Slides
- File: Java3018s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[181] This content is available online at <http://cnx.org/content/m45636/1.2/>.
[182] http://cnx.org/content/m44234
[183] http://cnx.org/content/m45636/latest/a0-Index.htm

-end-

5.3.31 Java3020: Interfaces, Object Arrays, etc.[184]

5.3.31.1 Table of Contents

5.3.31.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

5.3.31.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.31.2.1.1 Images

5.3.31.2.1.2 Listings

[184] This content is available online at <http://cnx.org/content/m44214/1.5/>.

5.3.31.3 Preview

In this module, you will learn about :

- Interface definitions
- Implementing an interface in a class definition
- Defining interface methods in a class definition
- Storing references to new objects in elements of an array of type **Object**
- Casting elements to an interface type in order to call interface methods
- Parameterized constructors
- Overridden **toString** method

Program specifications

Write a program named **Prob05** that uses the class definition shown in Listing 1 (p. 1063) to produce the output shown in Image 1 (p. 1062) on the command line screen.

Image 1: Command line output for Prob05.

```
    Prob05
Put your first name here
Put your last name here
-18 -17 -16
-17 -17 -17
-12 -12 -12
```

5.353

No graphic output images required

There are no graphic output images required by this program. Therefore, it can be compiled and executed without a requirement to have Ericson's media library on the classpath.

Required text output

The output, which appears on the command line screen, consists of the six lines of text shown in Image 1 (p. 1062) .

Because the program generates random data for testing, the actual values will differ from one run to the next. However, in all cases:

- The values in the first row of numbers will be a sequence of consecutive integers in increasing algebraic order from left to right.
- All three values in the second row of numbers will match the value of the center number in the first row of numbers.
- All three values in the third row of numbers will be algebraically five greater than the values in the second row of numbers.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob05** shown in Listing 1 (p. 1063) .

5.3.31.4 General background information

Among other things, this program illustrates:

- Interface definitions
- Implementing an interface in a class definition
- Defining interface methods in a class definition
- Storing references to new objects in elements of an array of type **Object**
- Casting elements to an interface type in order to call interface methods
- Parameterized constructors
- Overridden **toString** method

5.3.31.5 Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 13 (p. 1074) near the end of the module.

Beginning of driver class for Prob05

The driver class for **Prob05** begins in Listing 1 (p. 1063) .

Listing 1: Beginning of driver class for Prob05.

```
    import java.util.*;

class Prob05{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomData = (byte)generator.nextInt();

    Object[] var1 = new Object[2];

    var1[0] = new Prob05MyClassA(randomData);
    var1[1] = new Prob05MyClassB(randomData);
```

Behavior of the code in Listing 1

Listing 1 (p. 1063) does the following:

- Gets and saves a random value of type **int** .
- Instantiates a new two-element array object of type **Object** . *(A reference to any object of any class or interface type can be stored in an array element of type* ***Object*** *.)*
- Populates the array object with references to objects of the classes:
  - Prob05MyClassA
  - Prob05MyClassB

The same random value is passed to the constructor for both objects when they are instantiated.

Put the driver class on temporary hold

At this point, I am going to put the driver class named **Prob05** on temporary hold and explain the class named **Prob05MyClassA** .

The interface named Prob05X

Having glanced ahead, I know that the class named **Prob05MyClassA** implements the interface named **Prob05X** so I will explain that interface first.

The interface named **Prob05X** is shown in its entirety in Listing 2 (p. 1064) .

Listing 2: The interface named Prob05X.

```
 interface Prob05X{
  public int getModifiedData();
  public int getData();
}//end interface
```

5.355

An interface definition

An interface definition can contain only two kinds of members:

- Constants
- Method declarations

By now, you should have studied interfaces in my online tutorials. Therefore, this explanation will be very brief.

Method declarations

Listing 2 (p. 1064) contains two method declarations.

A method declaration does not have a body. Its purpose is to establish the programming interface for that method in any class that implements the interface *(return type, name, arguments, etc.)* .

A method declaration provides no information about the behavior of the method.

A method declaration in an interface is implicitly abstract.

A concrete definition is required

Any class that implements an interface:

- Must provide a **concrete** version of every method that is declared in the interface, or
- The class must be declared **abstract** . *(In this case, abstract essentially means incomplete.)*

The class named Prob05MyClassA

The class named **Prob05MyClassA** , which implements the interface named **Prob05X** , must provide concrete versions of the methods named:

- public int getModifiedData()
- public int getData()

Beginning of the class named Prob05MyClassA

The class named **Prob05MyClassA** begins in Listing 3 (p. 1065) .

Listing 3: Beginning of the class named Prob05MyClassA.

```
  class Prob05MyClassA implements Prob05X{
private int data;//instance variable

Prob05MyClassA(int inData){//constructor
  System.out.println("Prob05");
  System.out.println("Put your first name here");
  data = inData;
}//end constructor
```

5.356

This class implements the interface named **Prob05X** .

A private instance variable

Listing 3 (p. 1065) begins by declaring a private instance variable of type **int** named **data** . As a private instance variable, it is accessible by any method or constructor defined within the class but is not accessible to methods from outside the class.

The constructor

The constructor for the class is shown in its entirety in Listing 3 (p. 1065) .

The constructor begins by displaying the problem number and the student's first name on the command line screen.

Then it assigns the value of the incoming parameter named **inData** to the variable named **data** . This makes that value available to the methods that are defined within the class.

The method named getModifiedData

We learned earlier [185] that the class named **Prob05MyClassA**

- must provide a concrete definition of the method named **getModifiedData** ,
- because that method is declared in the interface named **Prob05X** ,
- which is implemented by the class.

With the exception of some very subtle differences *(that are beyond the scope of this course)* , that concrete definition must match the signature of the declared method.

Code for the method named getModifiedData

The method named **getModifiedData** is shown in its entirety in Listing 4 (p. 1066) .

When this method is called, it

- subtracts a value of 1 from the value stored in the instance variable named **data** , and
- returns that modified value.

[185] http://cnx.org/content/m44214/latest/Java3020old.htm#concrete

Listing 4: The method named getModifiedData.

```
    public int getModifiedData(){
  return data - 1;
}//end getModifiedData()
```

5.357

The method named getData

We also learned earlier that the class named **Prob05MyClassA**

- must provide a concrete definition of the method named **getData,**
- which is also declared in the interface named **Prob05X** .

Code for the method named getData

The method named **getData** is shown in its entirety in Listing 5 (p. 1066) .

This method returns a copy of the value stored in the variable named **data** .

Listing 5: The method named getData.

```
    public int getData(){
  return data;
}//end getData()
```

5.358

A round trip

When the code in Listing 1 (p. 1063) instantiates an object of the **Prob05MyClassA** class, it passes a random value as a parameter to the constructor.

The constructor shown in Listing 3 (p. 1065) stores that random value in the instance variable named **data** .

When the method named **getModifiedData** is called, it returns a value that is the original random value less 1.

When the method named **getData** is called, it returns a copy of the original random value.

The toString method

The class named **Prob05MyClassA** extends the class named **Object** by default. It inherits a method named **toString** from the class named **Object** . The inherited method has very specific behavior.

Overridden toString method

The code in Listing 6 (p. 1067) overrides the inherited method to provide a different behavior when the method is executed in conjunction with an object of the **Prob05MyClassA** class.

The new behavior is to construct and return a string version of the value obtained by adding 5 to the value stored in **data** , which is the original random value.

Listing 6: Overridden toString method.

```
      public String toString(){
    return "" + (data + 5);
  }//end toString()

}//end class Prob05MyClassA
```

5.359

The end of the class named Prob05MyClassA

Listing 6 (p. 1067) also signals the end of the class definition for the class named **Prob05MyClassA** .

The class named Prob05MyClassB

Referring back to the code in the driver class in Listing 1 (p. 1063) , we see that the driver also instantiates an object of the class named **Prob05MyClassB** , passing the same random value to the constructor for the class.

The reference to the object is stored in the second element of the array object of type **Object** referred to by the reference variable named **var1** .

Beginning of the class named Prob05MyClassB

The beginning of the class named **Prob05MyClassB** is shown in Listing 7 (p. 1067) .

Listing 7: Beginning of the class named Prob05MyClassB.

```
    class Prob05MyClassB implements Prob05X{
  private int data;

  Prob05MyClassB(int inData){
    System.out.println("Put your last name here");
    data = inData;
  }//end constructor
```

5.360

Implements Prob05X

The first thing we notice is that this class also implements the interface named **Prob05X** . This requires that the class provide concrete definitions of the two methods declared in that interface.

Save the incoming parameter value

The constructor for the **Prob05MyClassB** class, which is shown in Listing 7 (p. 1067) , saves the incoming parameter value in a private instance variable named **data** .

Unrelated to the variable named data from before

It is important to note that this variable named **data** is completely unrelated to the private instance variable named **data** that is declared in Listing 3 (p. 1065) , even though they are the same type and they have the same name.

They belong to two different objects. Objects do not share instance variables.

The two objects are related

However, even though the two objects instantiated in Listing 1 (p. 1063) are instantiated from different classes, they are related in the sense that they have two ancestors in common. They both extend the class named **Object** by default and they both explicitly implement the interface named **Prob05X** . That means that they can both be treated as either type **Object** or type **Prob05X** .

Related through the interface by design

Because all classes are direct or indirect subclasses of the class named **Object** , all objects instantiated for any class are related at the **Object** level. However, the objects in this program are related through the **Prob05X** interface only because I designed the program that way.

The method named getModifiedData

The method named **getModifiedData** is shown in Listing 8 (p. 1068) .

Listing 8: The method named getModifiedData.

```
  public int getModifiedData(){
 return data + 1;
}//end getModifiedData()
```

5.361

Same behavior is not required

A comparison of Listing 8 (p. 1068) with Listing 4 (p. 1066) exposes a very important aspect of interface implementation.

If two different classes implement the same interface, they each must provide concrete definitions of all the method declared in the interface. When providing such concrete definitions, both classes must match the method signatures of the declared methods.

However, the behavior of a method as defined in one class is not required to be the same as the behavior of the method having the same signature in the other class.

The behavior is different

For example, the code in Listing 4 (p. 1066) *subtracts 1* from the value of **data** and returns that modified value.

The code in Listing 8 (p. 1068) *adds 1* to the value of **data** and returns that modified value.

Therefore, the behavior of the method named **getModifiedData** in an object instantiated from the class named **Prob05MyClassB** is completely different from the behavior of the method having the same signature in an object of the class named **Prob05MyClassA** .

The getData and toString methods

Listing 9 (p. 1069) shows the **getData** and **toString** methods as defined in the class named **Prob05MyClassB** .

Listing 9: The getData and toString methods.

```
      public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + (data + 5);
  }//end toString()

}//end class Prob05MyClassB
```

5.362

The behavior is the same

If you compare Listing 9 (p. 1069) with Listing 5 (p. 1066) and Listing 6 (p. 1067) , you will see that these two methods are defined the same in both classes. Therefore, these two methods have the same behavior regardless of which of the two objects instantiated in Listing 1 (p. 1063) they are called on.

Back to the driver class named Prob05

Returning now to the driver class named **Prob05** where we left off in Listing 1 (p. 1063) , Listing 10 (p. 1069) contains three statements that print information on the command line screen.

Listing 10: Print three items of information.

```
        System.out.print(
              ((Prob05X)var1[0]).getModifiedData() + " ");

    System.out.print(randomData + " ");

    System.out.println(
                    ((Prob05X)var1[1]).getModifiedData());
```

5.363

Three print statements

The first two statements in Listing 10 (p. 1069) call the **print** method and the last statement calls the **println** method.

When the **println** method is called, the onscreen cursor advances to the left side of the next line after the material has been printed.

However, when the **print** method is called, the cursor remains at the right end of the printed material.

Therefore, calling **print print println** in succession will cause three items of information to be printed on the same line.

A cast is required

Recall that the reference to each object instantiated in Listing 1 (p. 1063) is stored in an array element as type **Object** .

A reference to any object can be stored in a reference of type **Object** because the **Object** class is the superclass of all classes. *(References to array objects can also be stored as type* ***Object*** *but that fact is not germane to this program.)*

Only eleven methods can be called on type Object

However, once an object's reference is stored as type **Object** , the only methods that can be called on that object *(without casting)* are the eleven methods that are defined in the **Object** class. That group of eleven methods includes the method named **toString** but it does not include the methods named **getData** and **getModifiedData** .

Must change the type of the reference

Therefore, the first statement in Listing 10 (p. 1069) requires that a **cast** to be used to change the type of the reference back to a type on which the method can be called. There are a couple of choices in this regard.

Could cast to the class type

First, it is always possible to cast the reference back to the class from which the object was instantiated. Therefore, it would work to cast the reference from array element 0 in Listing 10 (p. 1069) to type **Prob05MyClassA** and to cast the reference from array element 1 to type **Prob05MyClassB** .

Cast to the interface type

In this program, there is another choice. Because both classes implement the interface named **Prob05X** , and the method named **getModifiedData** is declared in that interface, it also works to cast both references to the common interface type **Prob05X** .

That is what was done in Listing 10 (p. 1069) . Both references were cast to the interface type **Prob05X** .

The printed values

The first statement in Listing 10 (p. 1069) calls the method named **getModifiedData** as defined in Listing 4 (p. 1066) . This causes the original random value *less 1* to be printed.

The second statement in Listing 10 (p. 1069) simply prints the original random value that was saved in the variable named **randomData** in Listing 1 (p. 1063) .

The third statement in Listing 10 (p. 1069) calls the method named **getModifiedData** as defined in Listing 8 (p. 1068) . This causes the original random value *plus 1* to be printed.

Because this is a call to the **println** method, the onscreen cursor advances to the left side of the next line after the value is printed.

The three statements in Listing 10 (p. 1069) cause the first three values shown in Image 1 (p. 1062) to be printed on the command line screen.

Three more print statements

Continuing with the driver class named **Prob05** , Listing 11 (p. 1070) shows three more print statements.

Listing 11: Three more print statements.

```
    System.out.print(((Prob05X)var1[0]).getData() + " ");
System.out.print(randomData + " ");
System.out.println(((Prob05X)var1[1]).getData());
```

5.364

A cast is required

In this case, the **getData** method belonging to each of the objects is called in the first and third statements. *(Once again a cast is required.)*

Behavior of the getData methods is the same

Recall that the behavior of the **getData** method is the same in both objects. It simply returns a copy of the original random value that was passed to the constructor when each of the objects was instantiated.

The three statements in Listing 11 (p. 1070) produce the second set of three matching values shown in Image 1 (p. 1062) .

These three values match because all three print statements are printing essentially the same value. The original random value is printed in the middle statement in Listing 11 (p. 1070) . A copy of the original random value is printed in the first and third statements.

Print the references to the two objects

Things get a little bit more complicated in Listing 12 (p. 1071) .

Listing 12: Print the references to the two objects.

```
        System.out.print(((Prob05X)var1[0]) + " ");
    System.out.print(randomData + 5 + " ");
    System.out.println(((Prob05X)var1[1]));

  }//end main
}//end class Prob05
```

5.365

An automatic call to the toString method

Whenever an object's reference is passed to either the **print** method or the **println** method, the first thing that happens is that the **toString** method is called on the reference. The **toString** method always returns a reference to an object of the **String** class, and it is that string that is printed.

Inherited default behavior of the toString method

As I mentioned earlier, the **toString** method is defined with default behavior in the **Object** class. Since every class is a subclass of the **Object** class, every class inherits that method.

If the **toString** method is not overridden in a class or in any of the superclasses of a given class and the **toString** method is called on an object of the given class, the default behavior of the **toString** method will occur.

Can override to change the behavior

However, any class can override the **toString** method to produce different behavior and can pass that behavior down the inheritance hierarchy to subclasses of the class that overrides the method.

The toString method is overridden

In this program, the **toString** method is overridden in exactly the same way in both the **Prob05MyClassB** class and the **Prob05MyClassB** class. *(See Listing 6 (p. 1067) and Listing 9 (p. 1069) .)* Therefore, when the **toString** method is called on an object of either class, it will return a string representation of the value stored in the variable named **data** plus 5.

Pass object references to the print and println methods

The first statement in Listing 12 (p. 1071) passes the reference to the object stored in the first element of the array to the **print** method and the third statement passes the reference to the object stored in the second element of the array to the **println** method.

Execute overridden toString methods and print the returned values

The **print** and **println** methods cause the code in Listing 6 (p. 1067) and Listing 9 (p. 1069) to be executed. In both cases, this code returns a string that represents the original random value plus 5. This is the value that is displayed.

Print the random value plus 5

The second statement in Listing 12 (p. 1071) adds five to the original random number and prints the result. These three statements produce the third line of text in Image 1 (p. 1062) where all three values are the algebraic sum of the original random number plus 5.

Important - The cast is not required

Even though the references extracted from the array in the first and third statements in Listing 12 (p. 1071) are cast to the interface type **Prob05X** , that cast is unnecessary.

Because the original definition of the **toString** method appears in the class named **Object** , the **toString** method can be called on those objects even while they are being treated as though they are of type **Object** .

Runtime polymorphism

Furthermore, a very powerful capability of OOP known as runtime polymorphism would cause the overridden versions of the methods defined in Listing 6 (p. 1067) and Listing 9 (p. 1069) to be executed instead of the default version of the method defined in the **Object** class.

The end of the main method

Listing 12 (p. 1071) signals the end of the **main** method and the end of the class named **Prob05** . When the **main** method has nothing further to do, it terminates causing the program to terminate and return control to the operating system.

5.3.31.6 Run the program

I encourage you to copy the code from Listing 13 (p. 1074) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

5.3.31.7 Summary

In this module, you learned about :

- Interface definitions
- Implementing an interface in a class definition
- Defining interface methods in a class definition
- Storing references to new objects in elements of an array of type **Object**
- Casting elements to an interface type in order to call interface methods
- Parameterized constructors
- Overridden **toString** method

5.3.31.8 What's next?

You will learn how to scale images and how to rotate and translate images using the AffineTransform class in the next module.

5.3.31.9 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 10 [186]

[186] http://www.youtube.com/playlist?list=PL3DB0B7840C943C4C

- Part01 [187]
- Part02 [188]
- Part03 [189]
- Part04 [190]

5.3.31.10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Interfaces, Object Arrays, etc.
- File: Java3020.htm
- Published: 08/02/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.31.11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 13 (p. 1074) below.

[187] http://www.youtube.com/watch?v=10R_Xgo9QEo
[188] http://www.youtube.com/watch?v=vNPd6Sd7Wk8
[189] http://www.youtube.com/watch?v=_JFcPromgGk
[190] http://www.youtube.com/watch?v=A3bgpy5dCtQ

Listing 13: Complete program listing.

```
    /*File Prob05 Copyright 2008 R.G.Baldwin
**********************************************************/

import java.util.*;

class Prob05{
  public static void main(String[] args){

    Random generator = new Random(new Date().getTime());
    int randomData = (byte)generator.nextInt();

    Object[] var1 = new Object[2];

    var1[0] = new Prob05MyClassA(randomData);
    var1[1] = new Prob05MyClassB(randomData);

    System.out.print(
              ((Prob05X)var1[0]).getModifiedData() + " ");
    System.out.print(randomData + " ");
    System.out.println(
                   ((Prob05X)var1[1]).getModifiedData());

    System.out.print(((Prob05X)var1[0]).getData() + " ");
    System.out.print(randomData + " ");
    System.out.println(((Prob05X)var1[1]).getData());

    System.out.print(((Prob05X)var1[0]) + " ");
    System.out.print(randomData + 5 + " ");
    System.out.println(((Prob05X)var1[1]));

  }//end main
}//end class Prob05
//========================================================//

interface Prob05X{
  public int getModifiedData();
  public int getData();
}//end interface
//========================================================//

class Prob05MyClassA implements Prob05X{
  private int data;

  Prob05MyClassA(int inData){
    System.out.println("Prob05");
    System.out.println("Put your first name here");
    data = inData;
  }//end constructor
  //--------------------------------------------------//

  public int getModifiedData(){
    return data - 1;
  }//end getModifiedData()
  //--------------------------------------------------//
```

-end-

5.3.32 Java3020s Slides[191]

5.3.32.1 Table of Contents

5.3.32.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3020: Interfaces, Object Arrays, etc. [192]
.

Click here [193] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.32.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3020s Slides
- File: Java3020s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[191] This content is available online at <http://cnx.org/content/m45632/1.2/>.
[192] http://cnx.org/content/m44214
[193] http://cnx.org/content/m45632/latest/a0-Index.htm

5.3.33 Java3022: Scaling, Rotating, and Translating Images using Affine Transforms[194]

5.3.33.1 Table of Contents

5.3.33.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [195] .

5.3.33.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.33.2.1.1 Images

5.3.33.2.1.2 Listings

[194] This content is available online at <http://cnx.org/content/m44223/1.6/>.
[195] http://cnx.org/content/m44148/latest/

- Listing 9 (p. 1087) . Perform the concatenated transform.
- Listing 10 (p. 1090) . Complete program listing.

5.3.33.3 Preview

In this module, you will learn how to scale images and how to rotate and translate images using the **AffineTransform** class.

Program specifications

Write a program named **Prob01** that uses the class definition shown in Listing 1 (p. 1081) and Ericson's media library along with the image file named **Prob01.jpg** *(see Image 1 (p. 1077))* to produce the output images shown in Image 2 (p. 1078) and Image 3 (p. 1079) .

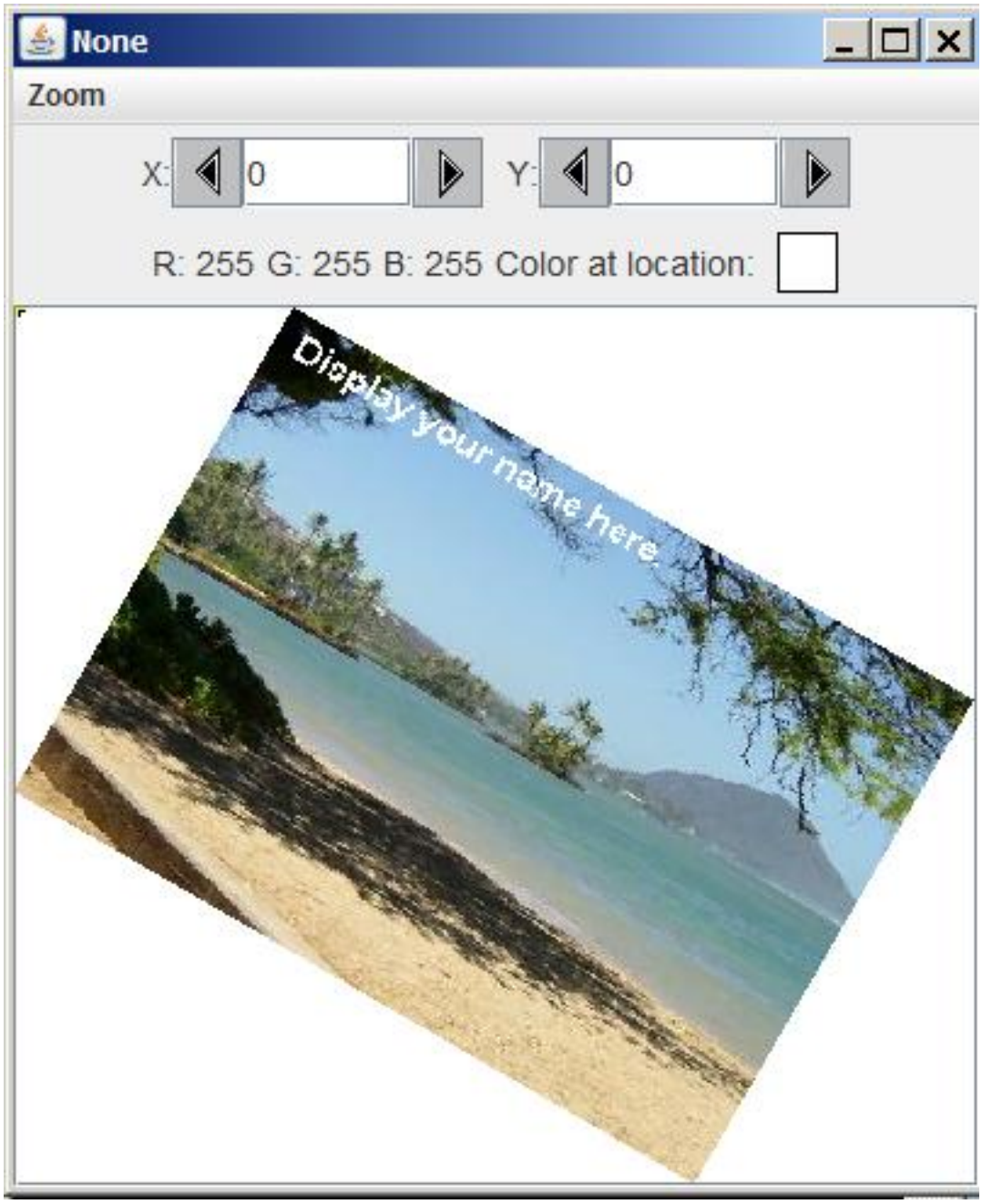
Image 1: Input file named Prob01.jpg.

5.367

Image 2: First output image.

5.368

Image 3: Second output image.



5.369

Scale and rotate

The image from the file named **Prob01.jpg** must be scaled and then rotated 30 degrees clockwise. A scale factor of 0.95 must be applied to the horizontal and a scale factor of 0.9 must be applied to the vertical.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob01** shown in Listing 1 (p. 1081) .

Required output text

In addition to the two output images mentioned above, your program must display your name and the other line of text shown in Image 4 (p. 1080) on the command-line screen.

Image 4: Required output text.

```
 Display your name here.
Picture, filename None height 360 width 394
```

5.370

5.3.33.4 General background information

Writing the code from scratch to rotate an image can be a daunting task. However, the task is made much easier through the use of the standard **AffineTransform** class, which is included in the standard Java library.

The **AffineTransform** class can also be used to scale and translate images.

5.3.33.5 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 10 (p. 1090) near the end of the module.

The driver class named Prob01

The driver class containing the **main** method is shown in Listing 1 (p. 1081) .

Listing 1: The driver class named Prob01.

```
    import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob01Runner().run();
  }//end main method
}//end class Prob01
```

5.371

Instantiate a new object and call its run method

As has been the case in several earlier modules, the code in the **main** method instantiates a new object of the class named **Prob01Runner** and calls the **run** method on that object.

When the **run** method returns, the **main** method terminates causing the program to terminate.

Beginning of the class named Prob01Runner

The class named **Prob01Runner** begins in Listing 2 (p. 1081) .

Listing 2: Beginning of the class named Prob01Runner.

```
    class Prob01Runner{
  public Prob01Runner(){
    System.out.println("Display your name here.");
  }//end constructor
```

5.372

Listing 2 (p. 1081) shows the constructor for the class, which simply displays the student's name on the command line screen as shown in Image 4 (p. 1080) .

The run method

The **run** method, which is called in Listing 1 (p. 1081) , is shown in its entirety in Listing 3 (p. 1082) .

Listing 3: The run method.

```
  public void run(){
Picture pic = new Picture("Prob01.jpg");
//Add your name and display the picture.
pic.addMessage("Display your name here.",10,20);
pic.explore();

pic = pic.scale(0.95,0.9);
pic = rotatePicture(pic,30);

pic.explore();
System.out.println(pic);
}//end run
```

5.373

Mostly familiar code

You are already familiar with all of the code in Listing 3 (p. 1082) except for the call to the **scale** method and the call to the **rotatePicture** method.

The scale method

The **scale** method is part of Ericson's library. However, it is not included in the library on the CD in the back of my copy of her textbook. It is included in the zip files containing later versions, which can be downloaded from Ericson's website. *(See Java OOP: The Guzdial-Ericson Multimedia Class Library* [196] *.)*

The scale method is straightforward

When the **scale** method is called on a **Picture** object, it creates and returns a reference to a new **Picture** object that is a scaled version of the original.

Parameters

The **scale** method requires two parameters of type **double** . The first parameter is the scale factor that is applied to the horizontal dimension of the picture. The second parameter is the scale factor that is applied to the vertical dimension of the picture.

Replace original picture with scaled picture

The reference to the new **Picture** object returned by the **scale** method in Listing 3 (p. 1082) is stored in the variable named **pic** overwriting the reference to the original **Picture** object. From this point forward, all operations are performed on the scaled version of the original picture.

Beginning of the method named rotatePicture

The method named **rotatePicture** begins in Listing 4 (p. 1083) .

[196] http://cnx.org/content/m44148/latest/

Listing 4: Beginning of the method named rotatePicture.

```
 private Picture rotatePicture(Picture pic,double angle){

//Prepare the rotation transform
AffineTransform rotateTransform =
                                new AffineTransform();
rotateTransform.rotate(Math.toRadians(angle),
                       pic.getWidth()/2,
                       pic.getHeight()/2);
```

5.374

Rotate and translate

The **rotatePicture** method accepts a reference to a **Picture** object along with a rotation angle in degrees.

It creates and returns a new **Picture** object that is of the correct size, containing the rotated version of the image as shown in Image 3 (p. 1079) .

The incoming image is rotated around its center by the specified rotation angle. Then it is translated to and drawn in the center of the new **Picture** object.

Affine transforms

The **rotatePicture** method uses *affine transforms* to rotate and translate the image. Affine transforms can also be used to scale images, but it is easier to scale images using Ericson's **scale** method.

However, the lack of complexity of the **scale** method is easily made up for by the complexity of affine transforms.

Google me

I have published several tutorials discussing and explaining the use of the **AffineTransform** class in Java. You can locate those modules by going to Google and searching for the following keywords:

richard baldwin affine transform

The AffineTransform class

The **AffineTransform** class is part of the standard Java library. Here is part of what the documentation [197] has to say about the class:

"The AffineTransform class represents a 2D affine transform that performs a linear mapping from 2D coordinates to other 2D coordinates that preserves the "straightness" and "parallelness" of lines. Affine transformations can be constructed using sequences of translations, scales, flips, rotations, and shears."

The ideas behind affine transforms

One of the ideas behind affine transforms is that you can create an affine transform object and apply it to an unlimited number of other objects. This might be useful in a game program, for example, where a large number of enemy ships need to be rotated, translated, and scaled in unison.

Concatenated affine transform objects

Another idea is that you can create two or more affine transform objects, concatenate them, and apply the concatenated transform object to an unlimited number of other objects.

Application of concatenated transform objects

Applying a concatenated transform to an object is equivalent to applying one of the transform objects to the original object and then applying the other transform objects to the transformed objects in sequential

[197] http://java.sun.com/javase/6/docs/api/java/awt/geom/AffineTransform.html

fashion. Concatenation of transform objects can result in considerable computational savings in certain situations.

A larger Picture object is required

Looking back at Image 2 (p. 1078) and Image 3 (p. 1079) , you can see that the **Picture** object required to contain the rotated image must be larger than the **Picture** object required to contain the original image. You will learn how to compute the dimensions of the larger **Picture** object later in this module.

Behavior of the rotatePicture method

The **rotatePicture** method performs the following operations:

- Prepare an **AffineTransform** object that can be used to rotate the incoming image around its center by the specified angle.
- Get the dimensions of a rectangle of sufficient size to contain the rotated image.
- Prepare an **AffineTransform** object that will translate the rotated image to the center of a new, larger **Picture** object having the dimensions computed above.
- Concatenate the rotation transform object with the translation transform object.
- Create a new **Picture** object with the dimensions computed above.
- Apply the concatenated transform to the incoming image and draw the transformed image in the new **Picture** object.
- Return a reference to the new **Picture** object containing the rotated and translated image.

Prepare the rotation transform

Listing 4 (p. 1083) begins by instantiating a new object of the **AffineTransform** class and saving the object's reference in the local reference variable named **rotateTransform** .

Call an overloaded rotate method

Then Listing 4 (p. 1083) calls one of four overloaded **rotate** methods on the rotation transform object.

Three parameters are required

This version of the **rotate** method requires three parameters:

- theta - the angle of rotation measured in radians
- anchorx - the X coordinate of the rotation anchor point
- anchory - the Y coordinate of the rotation anchor point

To make a long story short...

The **rotate** method prepares the transform object to rotate an image around the point specified by the last two parameters.

The angle of rotation must be specified in radians.

Convert from degrees to radians

Listing 4 calls the static **toRadians** method of the **Math** class to convert the rotation angle from degrees to radians. The angle in radians is passed as the first parameter *(theta)* to the **rotate** method.

Compute the anchor point

Then the code in Listing 4 computes the coordinates of the center of the image and passes those coordinates to the **rotate** method as *anchorx* and *anchory* .

The dimensions of the new Picture object

How would you compute the dimensions of the new **Picture** object required to barely contain the rotated image shown in Image 3 (p. 1079) ?

The computation of those dimensions is not rocket science, but would certainly require you to know quite a lot about dealing with angles and triangles.

Fortunately, we don't have to perform that computation

Ericson provides a method named **getTransformEnclosingRect** that will perform that computation for us, returning the required dimensions in the form of a reference to a standard Java **Rectangle2D** object.

Compute the dimensions of the new Picture object

The code in Listing 5 (p. 1085) calls the **getTransformEnclosingRect** method on the previously scaled **Picture** object passing a reference to the rotation transform object to get the required dimensions for a **Picture** object that will contain the rotated image.

Listing 5: Compute the dimensions of the new Picture object.

```
    Rectangle2D rectangle2D =
       pic.getTransformEnclosingRect(rotateTransform);

int resultWidth = (int)(rectangle2D.getWidth());
int resultHeight = (int)(rectangle2D.getHeight());
```

5.375

The dimensions of the rectangle

After getting a reference to the rectangle, Listing 5 (p. 1085) gets and saves the **width** and **height** of the rectangle. These values will be used later to instantiate a new **Picture** object of the same size as the rectangle.

Prepare the translation transform

Listing 6 (p. 1085) prepares a translation transform that can be used to translate the rotated image to the center of the new **Picture** object.

Listing 6: Prepare the translation transform.

```
    AffineTransform translateTransform =
                                 new AffineTransform();
translateTransform.translate(
                 (resultWidth - pic.getWidth())/2,
                 (resultHeight - pic.getHeight())/2);
```

5.376

A new AffineTransform object

Listing 6 (p. 1085) begins by instantiating a new object of the **AffineTransform** class and saving the object's reference in the local reference variable named **translateTransform** .

Call the translate method on the transform object

Then Listing 6 (p. 1085) calls the **translate** method on the **AffineTransform** object.

According to the documentation [198] , the required parameters of the **translate** method are:

- tx - the distance by which coordinates are translated in the X axis direction
- ty - the distance by which coordinates are translated in the Y axis direction

[198] http://java.sun.com/javase/6/docs/api/java/awt/geom/AffineTransform.html#translate%28double,%20double%29

Compute the translation distance components

Listing 6 (p. 1085) computes the distance from the center of the image to the center of the new **Picture** object and passes the X and Y components of this distance to the **translate** method.

Two AffineTransform objects

At this point, we have two different **AffineTransform** objects. One is capable of rotating the image by a specified angle. The other is capable of translating the image by a specified amount.

We could apply the two transforms sequentially to the image being careful to rotate before we translate. *(The order of rotation and translation makes a huge difference.)*

A more computationally economical approach

The preferred approach is to concatenate the two transform objects and apply only the concatenated transform object to the image. This is particularly important if the transforms are going to be applied to a large number of images such as in a game program for example.

Concatenate the transforms

Listing 7 (p. 1086) calls the **concatenate** method on the translation transform passing a reference to the rotation transform as a parameter. This modifies the translation transform in such a way that it can be used to rotate the image around its center point and then translate it to the center of the new **Picture** object.

Listing 7: Concatenate the transforms.

```
translateTransform.concatenate(rotateTransform);
```

5.377

Instantiate the new Picture object

Listing 8 (p. 1086) instantiates a new **Picture** object with the dimensions computed in Listing 5 (p. 1085) . This **Picture** object will be used to contain and return the rotated image.

Listing 8: Instantiate the new Picture object .

```
Picture result = new Picture(
                        resultWidth,resultHeight);
```

5.378

Perform the concatenated transform

Listing 9 performs the rotation and translation, draws the modified image in the new **Picture** object, and returns a reference to the new **Picture** object.

Listing 9: Perform the concatenated transform .

```
        Graphics2D g2 = (Graphics2D)result.getGraphics();

    g2.drawImage(pic.getImage(),translateTransform,null);

    return result;
  }//end rotatePicture
}//end class Prob01Runner
```

5.379

Call the getGraphics method

Listing 9 (p. 1087) begins by calling Ericson's **getGraphics** method on the new **Picture** object and casting the returned value to the standard type **Graphics2D** .

A cast is required

Ericson's **getGraphics** method returns a reference to the graphics context of the **Picture** object as type **Graphics** . That reference must be cast to type **Graphics2D** before the **drawImage** method can be called on the reference.

Call the drawImage method

Then Listing 9 (p. 1087) calls the standard **drawImage** method on the reference to the graphics context passing three parameters to the method. This is one of two overloaded versions of the **drawImage** method defined in the standard **Graphics2D** class.

The first parameter

The first required parameter for this version of the **drawImage** method is a reference to an object of type **Image** containing the image that is to be drawn. In this case, Ericson's **getImage** method is called on the **Picture** object to get the image and pass it as the first parameter.

The second parameter

The second required parameter is a reference to an **AffineTransform** object that is to be applied to the image before it is drawn. Our concatenated transform object is passed as the second parameter.

The third parameter

The third required parameter is a reference to an object of the standard type **ImageObserver** or **null** . If you would like to know more about the use of an **ImageObserver** object, go to Google and search for the following keywords:

richard baldwin java imageobserver

We don't need an image observer in this case so Listing 9 (p. 1087) passes **null** for the third parameter.

When the drawImage method returns

When the **drawImage** method returns, the image will have been rotated, translated, and drawn in the center of the new **Picture** object as shown in Image 3 (p. 1079) .

Return a Picture and terminate the method

Listing 9 (p. 1087) returns a reference to the **Picture** object, *(which now contains the rotated image)* and terminates, returning control to the **run** method in Listing 3 (p. 1082) .

Return to the run method

Returning to the **run** method in Listing 3 (p. 1082) , we see that the remaining code in the **run** method:

- Calls Ericson's **explore** method on the returned **Picture** object producing the screen output shown in Image 3 (p. 1079) .

- Passes the returned **Picture** object to the **println** method producing the last line of text output shown in Image 4 (p. 1080) .
- Returns control the **main** method in Listing 1 (p. 1081) , causing the program to terminate as soon as the user dismisses both images from the screen.

5.3.33.6 Run the program

I encourage you to copy the code from Listing 10 (p. 1090) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. For example, try reversing the order of translation and rotation beginning with Listing 7 (p. 1086) . Make certain that you can explain why your changes behave as they do.

Click Prob01.jpg [199] to download the required input image file.

5.3.33.7 Summary

You learned how to scale images and how to rotate and translate images using the **AffineTransform** class.

5.3.33.8 What's next?

In the next module, you will learn how to mirror images both horizontally and vertically.

5.3.33.9 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 11 [200]

5.3.33.10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Scaling, Rotating, and Translating Images using Affine Transforms
- File: Java3022.htm
- Published: 08/02/12
- Revised: 02/17/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

[199] http://cnx.org/content/m44223/latest/Prob01.jpg
[200] http://vimeo.com/channels/itse2321/21211960

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.33.11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 1090) below.

Listing 10: Complete program listing.

```
    /*File Prob01 Copyright 2008 R.G.Baldwin
Revised 12/17/08
*******************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob01Runner().run();
  }//end main method
}//end class Prob01
//=======================================================//

class Prob01Runner{
  public Prob01Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-------------------------------------------------------//
  public void run(){
    Picture pic = new Picture("Prob01.jpg");
    //Add your name and display the picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    pic = pic.scale(0.95,0.9);
    pic = rotatePicture(pic,30);

    pic.explore();
    System.out.println(pic);
  }//end run
  //-------------------------------------------------------//

  //This method accepts a reference to a Picture object
  // along with a rotation angle in degrees. It creates
  // and returns a new Picture object that is of the
  // correct size to contain and display the incoming
  // picture after it has been rotated around its center
  // by the specified rotation angle and translated to the
  // center of the new Picture object.
  private Picture rotatePicture(Picture pic,double angle){

    //Set up the rotation transform
    AffineTransform rotateTransform =
                                  new AffineTransform();
    rotateTransform.rotate(Math.toRadians(angle),
                           pic.getWidth()/2,
                           pic.getHeight()/2);

    //Get the required dimensions of a rectangle that will
    // contain the rotated image.
    Rectangle2D rectangle2D =
           pic.getTransformEnclosingRect(rotateTransform);
    int resultWidth = (int)(rectangle2D.getWidth());
```

-end-

5.3.34 Java3022r Review[201]

5.3.34.1 Table of Contents

5.3.34.2 Preface

This module contains review questions and answers keyed to the module titled Java3022: Scaling, Rotating, and Translating Images using Affine Transforms [202] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.34.3 Questions

5.3.34.3.1 Question 1 .

Given the input image in Image 1 (p. 1094) , which of the following output images is produced by the code in Listing 1 (p. 1093) ?

A. Image 2 (p. 1095)
B. Image 3 (p. 1096)

[201] This content is available online at <http://cnx.org/content/m45783/1.1/>.
[202] http://cnx.org/content/m44223

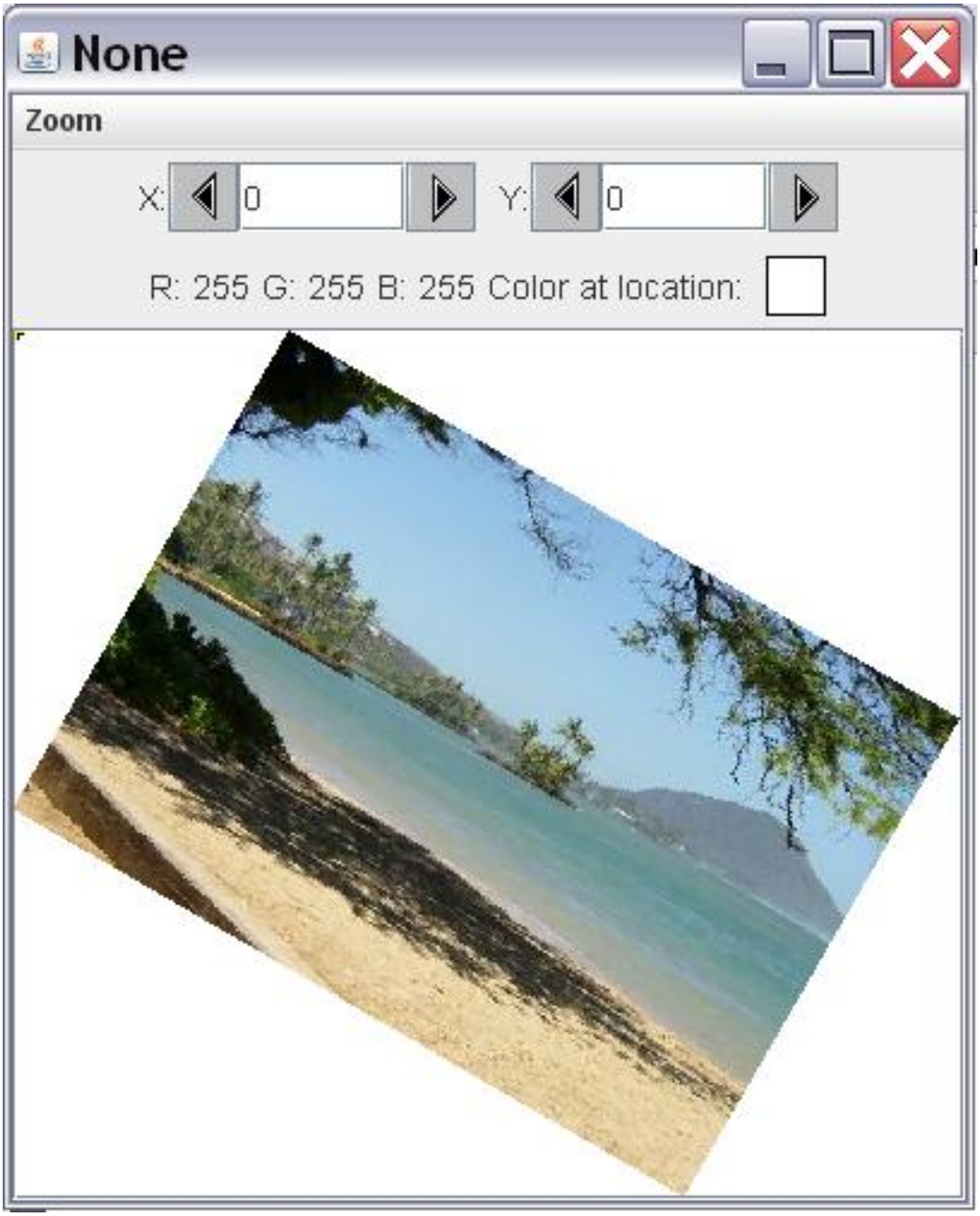
Listing 1. Question 1.

```
/*File Java3022ra Copyright 2013 R.G.Baldwin
Revised 02/17/13
*******************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Java3022ra{
  public static void main(String[] args){
    new Java3022raRunner().run();
  }//end main method
}//end class Java3022ra
//=======================================================//

class Java3022raRunner{
  public void run(){
    procPix(new Picture("Prob01.jpg").scale(0.7,0.7),-30);
  }//end run
  //-----------------------------------------------------//

  private void procPix(Picture pic,double angle){

    AffineTransform xformA = new AffineTransform();
    xformA.rotate(Math.toRadians(angle),pic.getWidth()/2,
                                      pic.getHeight()/2);

    Rectangle2D rectangle2D =
                    pic.getTransformEnclosingRect(xformA);
    int resultWidth = (int)(rectangle2D.getWidth());
    int resultHeight = (int)(rectangle2D.getHeight());

    AffineTransform xformB = new AffineTransform();
    xformB.translate((resultWidth - pic.getWidth())/2,
                     (resultHeight - pic.getHeight())/2);

    xformB.concatenate(xformA);
    Picture result = new Picture(
                                resultWidth,resultHeight);

    Graphics2D g2 = (Graphics2D)result.getGraphics();
    g2.drawImage(pic.getImage(),xformB,null);
    result.explore();
  }//end
  //-----------------------------------------------------//

}//end class Java3022raRunner
```

Image 1. Prob01.jpg.

5.382

Image 2. Possible output image.



5.383

Image 3. Possible output image.

5.384

Answer 1 (p. 1098)

5.3.34.4 Images

- Image 1 (p. 1094) . Prob01.jpg.
- Image 2 (p. 1095) . Possible output image.
- Image 3 (p. 1096) . Possible output image.

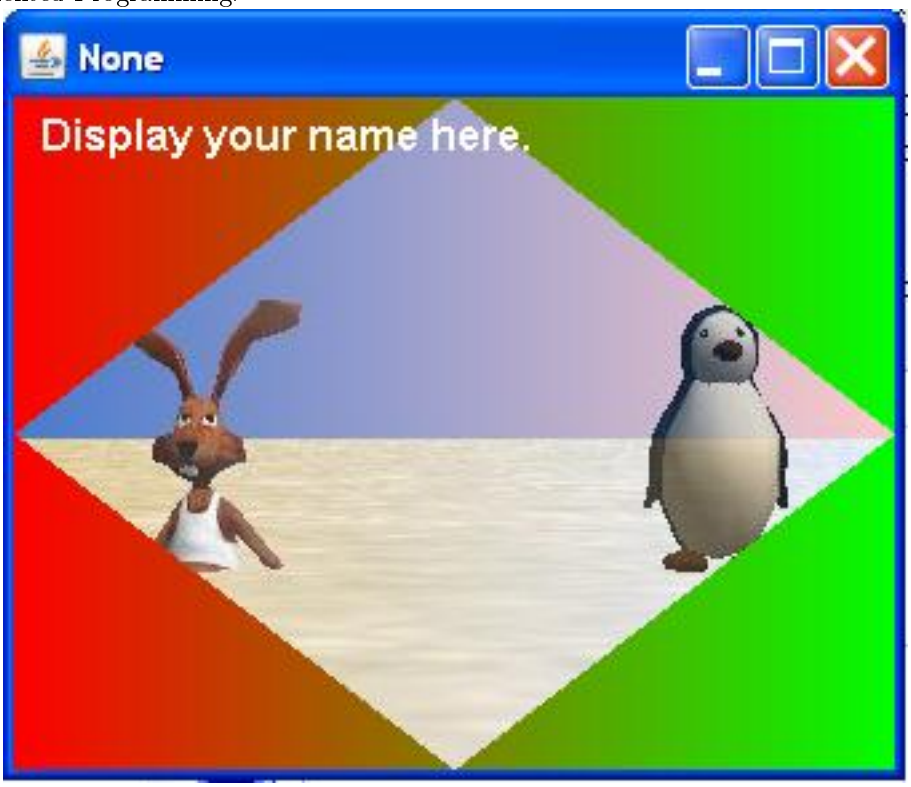
5.3.34.5 Listings

- Listing 1 (p. 1093) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.34.6 Answers

5.3.34.6.1 Answer 1

The code in Listing 1 (p. 1093) produces the output image shown in Image 3 (p. 1096) .
Back to Question 1 (p. 1092)

5.3.34.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3022r Review
- File: Java3022r.htm
- Published: 02/17/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.35 Java3022s Slides[203]

5.3.35.1 Table of Contents

5.3.35.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3022: Scaling, Rotating, and Translating Images using Affine Transforms [204] .

Click here [205] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.35.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3022s Slides
- File: Java3022s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[203] This content is available online at <http://cnx.org/content/m45639/1.2/>.
[204] http://cnx.org/content/m44223
[205] http://cnx.org/content/m45639/latest/a0-Index.htm

-end-

5.3.36 Java3024: Mirroring Images[206]

5.3.36.1 Table of Contents

5.3.36.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [207] .

5.3.36.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.36.2.1.1 Images

5.3.36.2.1.2 Listings

[206] This content is available online at <http://cnx.org/content/m44228/1.6/>.
[207] http://cnx.org/content/m44148/latest/

5.3.36.3 Preview

In this module, you will learn how to mirror images, both horizontally and vertically.

Program specifications

Write a program named **Prob02** that uses the class definition shown in Listing 1 (p. 1105) and Ericson's media library along with the image file named **Prob02a.jpg** *(shown in Image 1* (p. 1101)) to produce the three graphic output images shown in Image 2 (p. 1102) , Image 3 (p. 1103) , and Image 4 (p. 1104) .

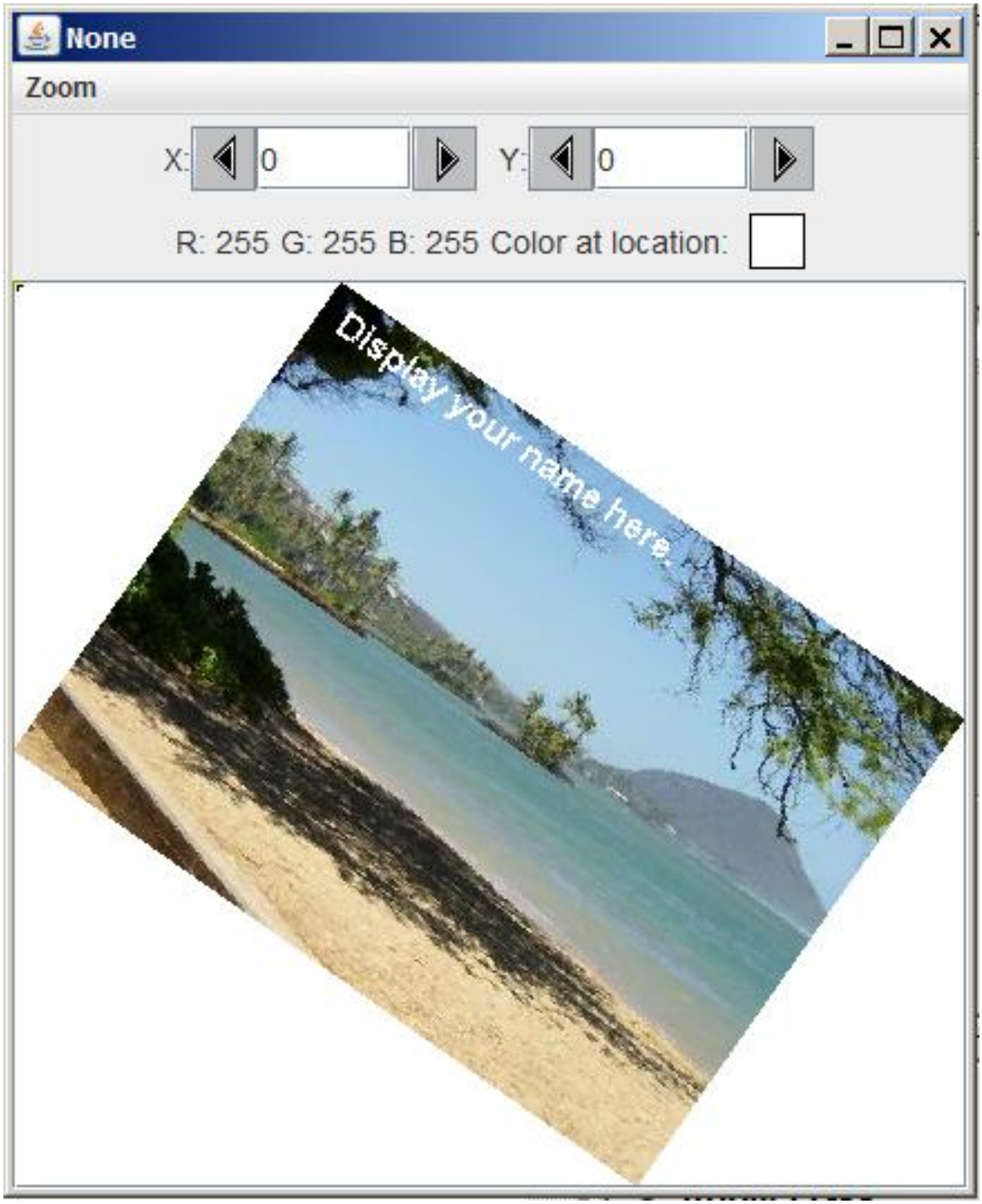
Image 1: Input file named Prob02a.jpg.

5.385

Image 2: First output image.

5.386

Image 3: Second output image.



5.387

Image 4: Third output image.

5.388

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob02** shown in Listing 1 (p. 1105) .

Rotate and mirror

The image from the file named **Prob02a.jpg** is rotated by 35 degrees. It is not scaled. Then the top-left quadrant of the picture containing the rotated image is mirrored into the top-right quadrant. Following this, the top half of the picture is mirrored into the bottom half.

Required output text

In addition to the three output images mentioned above, your program must display your name and the other line of text shown in Image 5 (p. 1105) on the command-line screen

---

Image 5: Required output text.

```
    Display your name here.
Picture, filename None height 404 width 425
```

5.389

5.3.36.4 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 6 (p. 1113) near the end of the module.

The driver class named Prob02

The driver class containing the **main** method is shown in Listing 1 (p. 1105) .

Listing 1: The driver class named Prob02.

```
    public class Prob02{
  public static void main(String[] args){
    new Prob02Runner().run();
  }//end main method
}//end class Prob02
```

5.390

Instantiate a new object and call its run method

The code in the **main** method instantiates a new object of the class named **Prob02Runner** and calls the **run** method on that object.

When the **run** method returns, the **main** method terminates causing the program to terminate.

Beginning of the class named Prob02Runner

The beginning of the class named **Prob02Runner** , including the constructor, is shown in Listing 2 (p. 1106) .

Listing 2: Beginning of the class named Prob02Runner.

```
  class Prob02Runner{
public Prob02Runner(){
  System.out.println("Display your name here.");
}//end constructor
```

5.391

The constructor displays the student's name, producing the first line of output text shown in Image 5 (p. 1105) .

The run method

The **run** method that is called in Listing 1 (p. 1105) is shown in its entirety in Listing 3 (p. 1106) .

Listing 3: The run method.

```
  public void run(){
 Picture pix = new Picture("Prob02a.jpg");
 //Add your name and display the output picture.
 pix.addMessage("Display your name here.",10,20);
 //Display the input picture.
 pix.explore();

 pix = rotatePicture(pix,35);
 pix.explore();

 pix = mirrorUpperQuads(pix);
 pix = mirrorHoriz(pix);
 pix.explore();
 System.out.println(pix);
}//end run
```

5.392

Very familiar code

Except for the calls to the methods named **mirrorUpperQuads** and **mirrorHoriz** in Listing 3 (p. 1106) , you should already be familiar with all of the code in Listing 3.

The rotatePicture method

For example, the call to the method named **rotatePicture** is essentially the same as code that I explained in an earlier module. Therefore, I won't explain that method again in this module. You will find the code for the method named **rotatePicture** in Listing 6 (p. 1113) near the end of the module.

Operate on the picture with the rotated image

The original picture is replaced by a picture containing the rotated image shown in Image 3 (p. 1103) . From this point forward, all operations are performed on the **Picture** object containing the rotated image.

The method named mirrorUpperQuads

The method named **mirrorUpperQuads** that is called in the **run** method in Listing 3 (p. 1106) is shown in Listing 4 (p. 1107) .

Behavior of the method named mirrorUpperQuads

This method mirrors the upper-left quadrant of a picture into the upper-right quadrant as shown in Image 6 (p. 1109) .

Listing 4: The method named mirrorUpperQuads.

```
  private Picture mirrorUpperQuads(Picture pix){

 Pixel leftPixel = null;
 Pixel rightPixel = null;
 int midpoint = pix.getWidth()/2;
 int width = pix.getWidth();

 for(int row = 0;row < pix.getHeight()/2;row++){
   for(int col = 0;col < midpoint;col++){
     leftPixel = pix.getPixel(col,row);
     rightPixel =
                pix.getPixel(width-1-col,row);
     rightPixel.setColor(leftPixel.getColor());
   }//end inner loop
 }//end outer loop

 return pix;
}//end mirrorUpperQuads
```

5.393

Declare four working variables

Listing 4 (p. 1107) begins by declaring and initializing four working variables. The purpose of these variables should be obvious on the basis of their names and their initialization expressions.

Copy the pixel colors

Then Listing 4 (p. 1107) uses a double nested **for** loop to copy the colors from the pixels in the upper-left quadrant into the pixels in the upper-right quadrant. This is done in such a way as to form a mirror image about the center point as shown in Image 6 (p. 1109) .

The outer loop

The outer loop iterates on the rows of pixels in the top half of the image. Only the top half of the image is processed in this method because the top half will be mirrored into the bottom half later on.

The inner loop

The inner loop iterates on the columns in the left half of the image, copying pixel colors from the left half into the pixels in the right half.

Destruction of pixel colors

The colors in the pixels in the upper-right quadrant are overwritten by this method.

In effect, this method and the one following it destroys all of the pixel colors originally in the right half of the picture of the rotated image and all of the pixel colors originally in the bottom half of the picture.

The final picture shown in Listing 4 (p. 1107) contains only pixel from the upper-left quadrant of the picture with the rotated image.

Return a modified Picture object

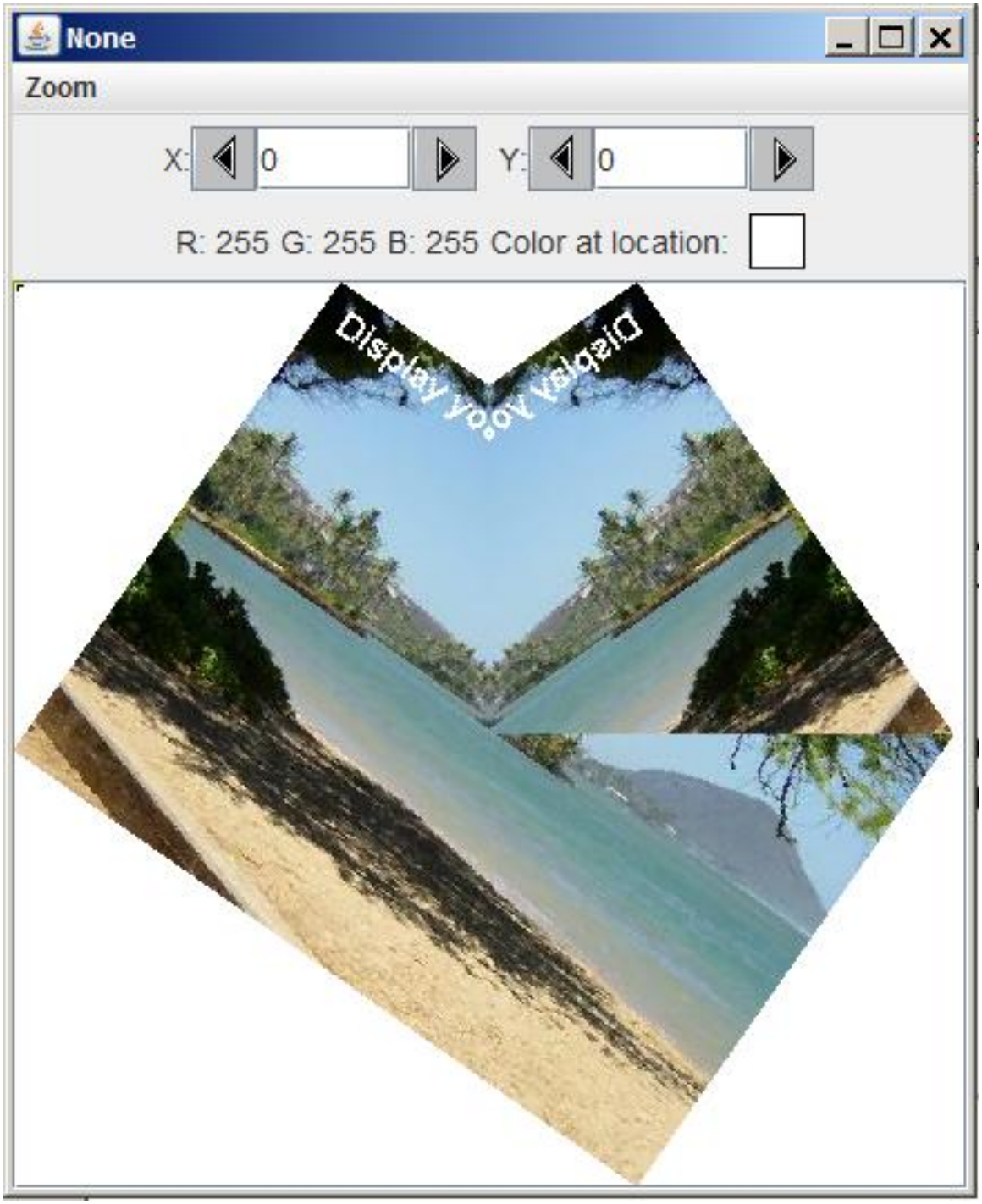
Finally, the code in Listing 4 (p. 1107) returns the modified **Picture** object to the **run** method in Listing 3 (p. 1106) .

At this point, the picture with the rotated image is replaced by the version of the picture returned by the **mirrorUpperQuads** method.

Picture output from the mirrorUpperQuads method

If you were to display the picture at that point, you would see the image shown in Image 6 (p. 1109) .

Image 6: Picture output from the mirrorUpperQuads method.



5.394

The upper-left quadrant has been mirrored

As you can see from Image 6 (p. 1109) , at this point in the process, the upper-left quadrant has been mirrored into the upper-right quadrant, but the bottom half of the picture is undisturbed. It's time to do something about that.

Call the mirrorHoriz method

The next statement in the **run** method in Listing 3 (p. 1106) is a call to the **mirrorHoriz** method passing the picture shown in Image 6 (p. 1109) as a parameter.

The method named mirrorHoriz

The method named **mirrorHoriz** is shown in Listing 5 (p. 1110) . This method mirrors the top half of a picture into the bottom half of the same picture. It will be used to mirror the top half of the picture in Image 6 (p. 1109) into the bottom half.

Listing 5: The method named mirrorHoriz.

```
  private Picture mirrorHoriz(Picture pix){

  Pixel topPixel = null;
  Pixel bottomPixel = null;
  int midpoint = pix.getHeight()/2;
  int height = pix.getHeight();

  for(int col = 0;col < pix.getWidth();col++){
    for(int row = 0;row < midpoint;row++){
      topPixel = pix.getPixel(col,row);
      bottomPixel =
                 pix.getPixel(col,height-1-row);
      bottomPixel.setColor(topPixel.getColor());
    }//end inner loop
  }//end outer loop

  return pix;
 }//end mirrorHoriz method

}//end class Prob02Runner
```

5.395

Very similar to the mirrorUpperQuads method

This method is very similar to the previous method named **mirrorUpperQuads** .

Four working variables and a nested for loop

As before, Listing 5 (p. 1110) declares and initializes four working variables. These variables are used in a nested **for** loop to copy pixel colors from the top half of the picture into the pixels in the bottom half.

The outer and inner loops

In this case, the outer loop iterates on all of the columns going from left to right.

The inner loop iterates on rows, from the top row to the vertical midpoint, copying the colors from the pixels from the top half into the pixels in the bottom half.

The end of the class

Listing 5 (p. 1110) also signals the end of the class named **Prob02Runner** and the end of the program.

5.3.36.5 Run the program

I encourage you to copy the code from Listing 6 (p. 1113) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click Prob02a.jpg [208] to download the required input image file for this program.

5.3.36.6 Summary

You learned how to mirror images both horizontally and vertically.

5.3.36.7 What's next?

In the next module, you will learn to use a variety of Java2D classes including GradientPaint.

5.3.36.8 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 12 [209]

5.3.36.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Mirroring Images
- File: Java3024.htm
- Published: 08/04/12
- Revised: 02/18/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[208] http://cnx.org/content/m44228/latest/Prob02a.jpg
[209] http://vimeo.com/channels/itse2321/21217708

5.3.36.10 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 6 (p. 1113) below.

Listing 6: Complete program listing.

```
    /*File Prob02 Copyright 2008 R.G.Baldwin
**********************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob02Runner().run();
  }//end main method
}//end class Prob02
//========================================================//

class Prob02Runner{
  public Prob02Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-------------------------------------------------------//
  public void run(){
    Picture pix = new Picture("Prob02a.jpg");
    //Add your name and display the output picture.
    pix.addMessage("Display your name here.",10,20);
    //Display the input picture.
    pix.explore();

    pix = rotatePicture(pix,35);
    pix.explore();

    pix = mirrorUpperQuads(pix);
    pix = mirrorHoriz(pix);
    pix.explore();
    System.out.println(pix);
  }//end run
  //-------------------------------------------------------//

  private Picture rotatePicture(Picture pix,
                                double angle){

    //Set up the rotation transform
    AffineTransform rotateTransform =
                                    new AffineTransform();
    rotateTransform.rotate(Math.toRadians(angle),
                           pix.getWidth()/2,
                           pix.getHeight()/2);

    //Get the required dimensions of a rectangle that will
    // contain the rotated image.
    Rectangle2D rectangle2D =
           pix.getTransformEnclosingRect(rotateTransform);
    int resultWidth = (int)(rectangle2D.getWidth());
    int resultHeight = (int)(rectangle2D.getHeight());

    //Set up the translation transform that will translate
```

-end-

5.3.37 Java3024r Review[210]

5.3.37.1 Table of Contents

5.3.37.2 Preface

This module contains review questions and answers keyed to the module titled Java3024: Mirroring Images [211].

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.37.3 Questions

5.3.37.3.1 Question 1 .

Given the input image shown in Image 1 (p. 1117) , which of the following output images is produced by the code in Listing 1 (p. 1116) ?

A. Image 2 (p. 1118)
B. Image 3 (p. 1119)

[210] This content is available online at <http://cnx.org/content/m45784/1.1/>.
[211] http://cnx.org/content/m44228

Listing 1. Question 1.

```
/*File Java3024ra Copyright 2013 R.G.Baldwin
*********************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Java3024ra{
  public static void main(String[] args){
    new Java3024raRunner().run();
  }//end main method
}//end class Java3024ra
//=======================================================//

class Java3024raRunner{
  public void run(){
    Picture pix = new Picture("Prob02a.jpg");
    pix = rotatePicture(pix,35);
    pix = mirrorUpperQuads(pix);
    pix = mirrorHoriz(pix);
    pix.explore();
    System.out.println(pix);
  }//end run
  //-----------------------------------------------------//

  private Picture rotatePicture(Picture pix,
                                double angle){

    AffineTransform rotateTransform =
                                    new AffineTransform();
    rotateTransform.rotate(Math.toRadians(angle),
                           pix.getWidth()/2,
                           pix.getHeight()/2);

    Rectangle2D rectangle2D =
           pix.getTransformEnclosingRect(rotateTransform);
    int resultWidth = (int)(rectangle2D.getWidth());
    int resultHeight = (int)(rectangle2D.getHeight());

    AffineTransform translateTransform =
                                    new AffineTransform();
    translateTransform.translate(
                      (resultWidth - pix.getWidth())/2,
                      (resultHeight - pix.getHeight())/2);

    translateTransform.concatenate(rotateTransform);
    Picture result = new Picture(
                             resultWidth,resultHeight);

    Graphics2D g2 = (Graphics2D)result.getGraphics();
    g2.drawImage(pix.getImage(),translateTransform,null);

    return result;
  }//end rotatePicture
```

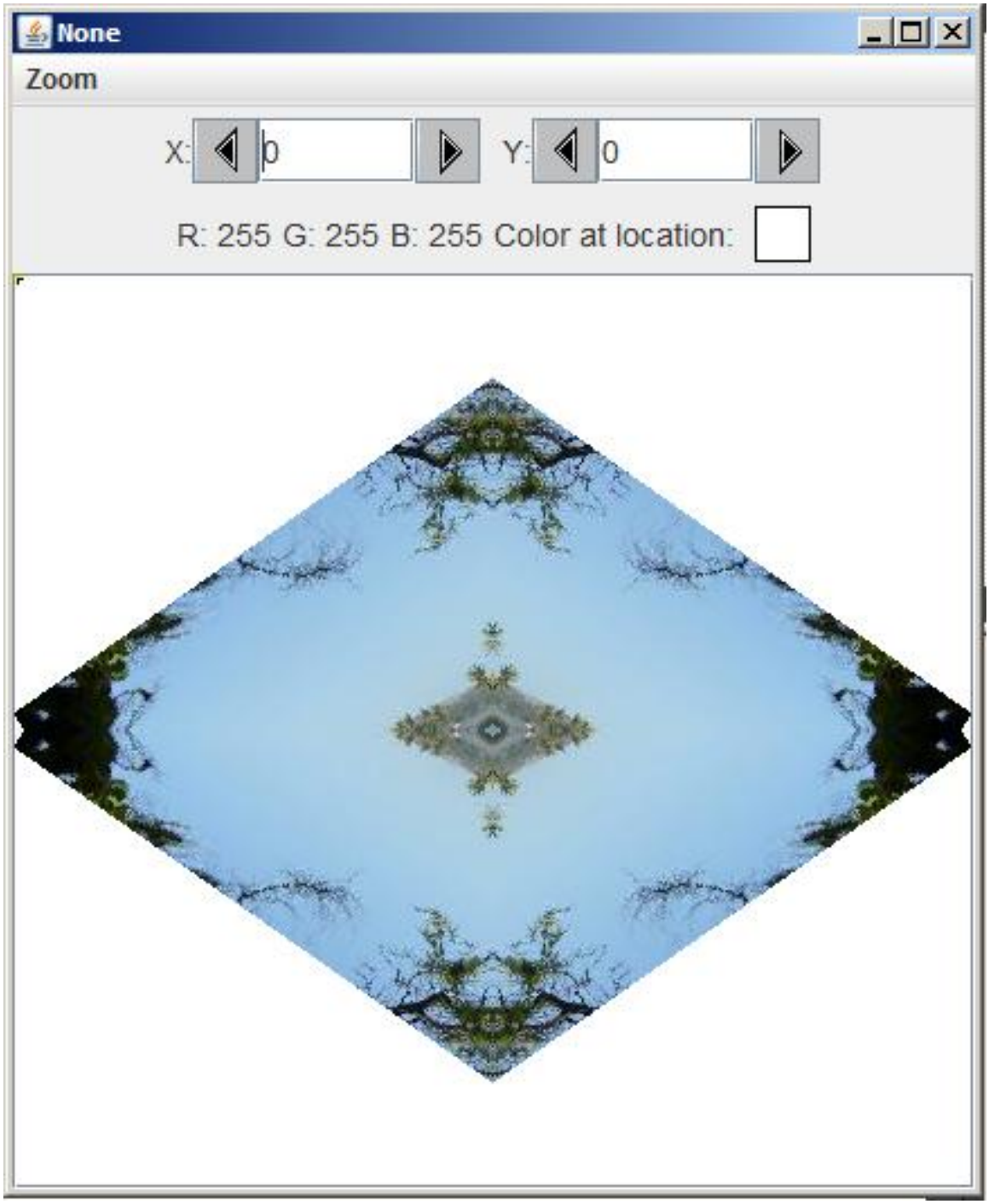
Image 1. Prob02a.jpg.

5.398

Image 2. Possible output image.

5.399

Image 3. Possible output image.



5.400

Answer 1 (p. 1121)

5.3.37.4 Images

- Image 1 (p. 1117) . Prob02a.jpg.
- Image 2 (p. 1118) . Possible output image.
- Image 3 (p. 1119) . Possible output image.

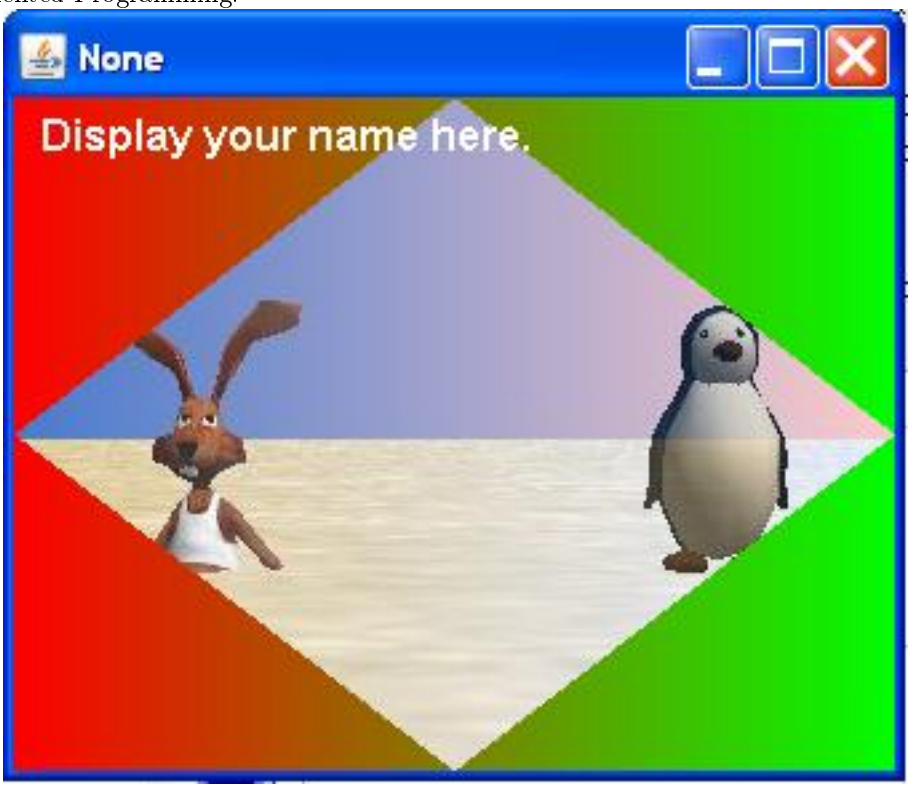
5.3.37.5 Listings

- Listing 1 (p. 1116) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.37.6 Answers

5.3.37.6.1 Answer 1

The code in Listing 1 (p. 1116) produces the output image shown in Image 2 (p. 1118) .
Back to Question 1 (p. 1115)

5.3.37.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3024r Review
- File: Java3024r.htm
- Published: 02/18/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.38 Java3024s Slides[212]

5.3.38.1 Table of Contents

5.3.38.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3024: Mirroring Images [213] .

Click here [214] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.38.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3024s Slides
- File: Java3024s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[212] This content is available online at <http://cnx.org/content/m45640/1.2/>.
[213] http://cnx.org/content/m44228
[214] http://cnx.org/content/m45640/latest/a0-Index.htm

5.3.39 Java3026: GradientPaint and other Java2D Classes[215]

5.3.39.1 Table of Contents

5.3.39.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [216] .

5.3.39.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.39.2.1.1 Images

5.3.39.2.1.2 Listings

[215] This content is available online at <http://cnx.org/content/m44242/1.6/>.
[216] http://cnx.org/content/m44148/latest/

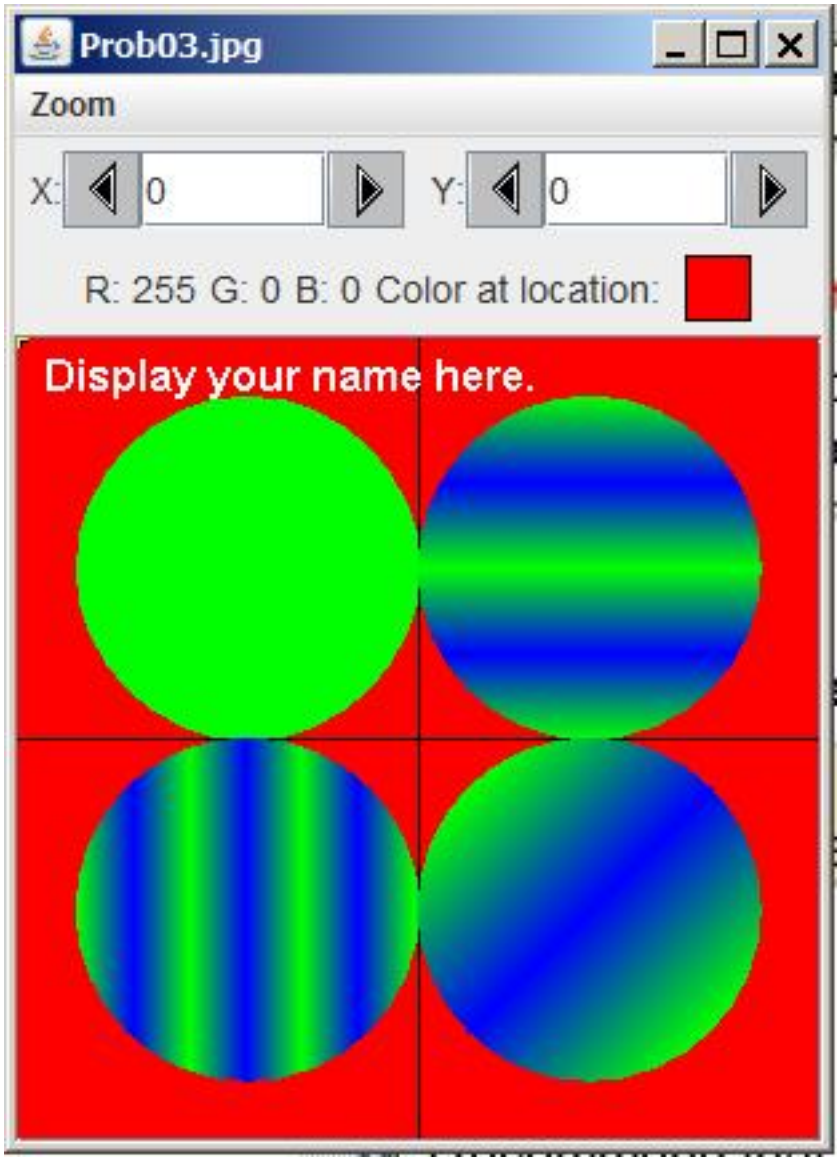
5.3.39.3 Preview

In this module you will learn to use the **GradientPaint** class along with a variety of other Java2D classes.

Program specifications

Write a program named **Prob03** that uses the class definition shown in Listing 1 (p. 1126) and Ericson's media library along with the image file named **Prob03.jpg** to produce the graphic output image shown in Image 1 (p. 1125) . *(Note that the image in the file named Prob03.jpg is a blank white image. You could also create this blank image using one of the constructors for the* ***Picture*** *class.)*

Image 1: Required graphic output.



5.401

Circles with gradient paint

The program draws four circles in the quadrants of a Cartesian coordinate system. One is filled with solid green. The other three are filled with cyclic gradient paint from green to blue.

The number of cycles varies in each circle, as do the axes along which the gradient occurs.

The background is set to Color.RED.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob03** shown in Listing 1 (p. 1126) .

Required text output

In addition to the output image mentioned above, your program must display your name and the other line of text shown in Image 2 (p. 1126) .

Image 2: Required text output.

```
    Display your name here.
Picture, filename Prob03.jpg height 300 width 300
```

5.402

5.3.39.4 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 9 (p. 1138) near the end of the module.

The driver class named Prob03

The driver class containing the main method is shown in Listing 1 (p. 1126) .

Listing 1: The driver class named Prob03.

```
    public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
```

5.403

If you have been studying the earlier modules in this series, no explanation of Listing 1 (p. 1126) should be required.

Beginning of the class named Prob03Runner

The class named **Prob03Runner** begins in Listing 2 (p. 1127) .

Listing 2: Beginning of the class named Prob03Runner.

```
    class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-------------------------------------------------//

  public void run(){
    Picture pic = new Picture("Prob03.jpg");
//  Picture pic = new Picture(300,300);

    pic.setAllPixelsToAColor(Color.RED);

    process(pic);

    //Add your name and display the output picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    System.out.println(pic);
  }//end run
```

5.404

Avoiding use of a blank input image file

This program was originally written using an early version of Ericson's class library that didn't support the second statement in the **run** method. *(That statement was disabled by turning it into a comment in Listing 2* (p. 1127) *)* As a result, with that library, it was necessary to read an image file containing a blank white image to create a **Picture** object with a blank white image.

No longer a problem

That problem was rectified with an update to her library and the disabled statement can be substituted for the statement immediately above it. If you do that, you won't need the input image file.

Except for that, and the call to the method named **process** , you should already understand all of the code in Listing 2 (p. 1127) .

Beginning of the method named process

The method named **process** begins in Listing 3 (p. 1128) .

Listing 3: Beginning of the method named process.

```
 private void process(Picture pic){

Graphics2D g2 = (Graphics2D)(pic.getGraphics());

int width = pic.getWidth();
int height = pic.getHeight();
```

5.405

Java2D graphics

This is basically a module on the use of classes from the Java2D section of the standard class library. *(See my lessons 300 through 324 on Java2D graphics here* [217] *.)* Classes from Ericson's library are used mainly to support the display aspects of the program.

What are Java2D graphics?

Although the capabilities provided by Java2D graphics are wide and varied, in one way or another, they generally have to do with the creation of images by drawing.

A graphics context is required

In order to use the methods that will be using, it is necessary to gain access to the graphics context of an object as type **Graphics2D** . The first statement in Listing 3 (p. 1128) calls Ericson's **getGraphics** method to gain access to the graphics context of a **Picture** object.

A cast to type Graphics2D is required

However, the **getGraphics** method returns a reference to the graphics context as type **Graphics** . In order for us to use it to do what we want to do, we must cast it to type **Graphics2D** . This gives us access to many more methods that would be the case without the cast.

Save as type Graphics2D in a variable named g2

The graphics context for the **Picture** object is saved in Listing 3 (p. 1128) as type **Graphics2D** . The reference is saved in the reference variable named **g2** .

Save the width and height of the Picture object

The last two statements in Listing 3 (p. 1128) get and save the **width** and the **height** of the image encapsulated in the **Picture** object.

Translate the origin to the center of the image

By default, the origin *(with coordinates of 0,0)* is in the upper-left corner of the image. However, we would like to be able to work with a coordinate system in which the origin is at the center.

Listing 4 (p. 1128) calls the **translate** method to move the origin to the center of the image.

Listing 4: Translate the origin to the center of the image.

```
g2.translate(width/2,height/2);
```

5.406

[217] http://www.dickbaldwin.com/tocadv.htm

(Fortunately, you already understand affine transforms. Otherwise, you might not be able to understand the documentation for the ***translate*** *method.)*

From this point forward...

From this point forward, we can think of the coordinates of the pixel at the very center of the object as having values of 0,0. Locations to the left of center have negative X coordinates and locations above the center have negative Y coordinates.

Draw the black horizontal and vertical axes

The next thing we want to do is to draw the black horizontal and vertical axes that you see in the center of the image in Image 1 (p. 1125) . This is accomplished by the code in Listing 5 (p. 1129) .

Listing 5: Draw the black horizontal and vertical axes.

```
    //Set the drawing color to black
g2.setColor(Color.BLACK);

//Draw x-axis
g2.draw(new Line2D.Double(-width/2, 0.0,
                          width/2, 0.0));

//Draw y-axis
g2.draw(new Line2D.Double(0.0, -width/2,
                         0.0,  height/2));
```

5.407

Set the drawing color to black

Listing 5 (p. 1129) begins by calling the **setColor** method to set the drawing color to Color.BLACK.

(BLACK is a static constant in the ***Color*** *class that represents the color black.)*

A new Line2D.Double object

The fourth line of code in Listing 5 (p. 1129) instantiates a new object of the **Line2D.Double** class. This object represents a line extending between two points specified by coordinate values passed as parameters to the constructor.

A black horizontal line

The first pair of coordinate values specifies the left end of the black horizontal line in Image 1 (p. 1125) . The second pair of coordinate values specifies the right end of the black horizontal line in Image 1 (p. 1125) .

(See my Lesson Number 300 [218] *for an explanation of the somewhat unusual name of a class consisting of two words separated by a period:* ***Line2D.Double*** *.)*

Pass the line object to the Draw method

The new object's reference is passed to the **draw** method, which is responsible for causing the line to be drawn on the graphics context.

A black vertical axis

The last statement in Listing 5 (p. 1129) draws the black vertical line shown in Image 1 (p. 1125) .

Coordinates relative to the origin at the center

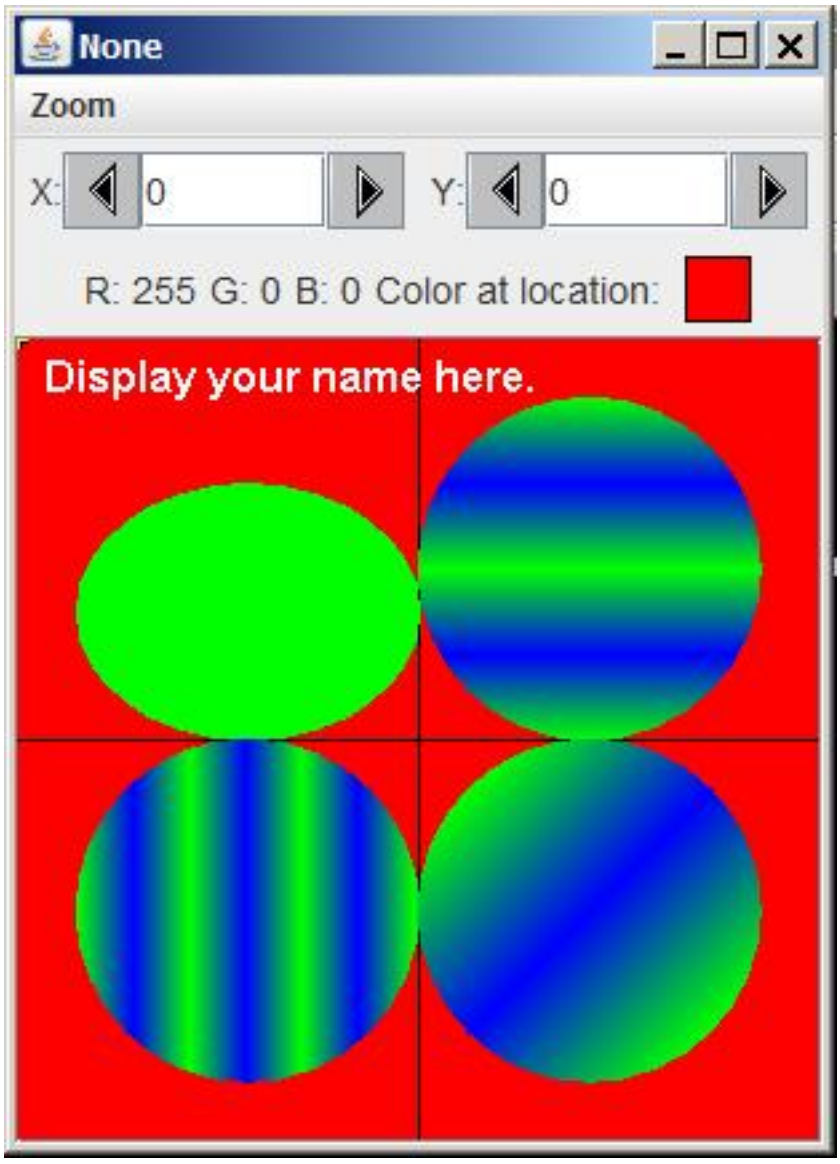
Note that in both cases, the end points of the line are specified using coordinate values that are relative to the origin, which is positioned at the center of the drawing context.

[218] http://www.dickbaldwin.com/java/Java300.htm

Draw the solid green filled ellipse in the upper-left quadrant

In case you are unfamiliar with the term, an *ellipse* is the 2D shape shown in the upper-left quadrant of Image 3 (p. 1130) . A circle is an ellipse with the major and minor axes having the same lengths.

Image 3: A drawing of an ellipse.



5.408

There is no circle class

Java does not provide a **Circle2D.Double** class but it does provide an **Ellipse2D.Double** class,

which you can use to draw circles.

(The ***Graphics*** *class also provides a* ***drawOval*** *method that can be used to draw circles but those circles won't suffice for what we will be doing in this module.)*

Four constructor parameters

The constructor for the **Ellipse2D.Double** class requires four parameters of type **double** . The first two parameters are the X and Y coordinates of the upper-left corner of an imaginary rectangle.

The next two parameters are the **width** and the **height** of the imaginary rectangle.

The sides of the imaginary rectangle are parallel to the X and Y axes, but can be rotated using affine transforms.

An ellipse inside an imaginary rectangle

The ellipse is constructed inside the imaginary rectangle such that it is symmetrical about its horizontal and vertical axes and it touches all four sides of the rectangle. *(The documentation refers to the rectangle as a framing rectangle.)*

How do you construct a circle?

If the rectangle is actually a square, then the ellipse becomes a circle.

Construct an ellipse inside an imaginary square

The first statement in Listing 6 (p. 1131) constructs an object of type **Ellipse2D.Double** inside a 128x128 square that just fits in the upper left quadrant of our Cartesian coordinate system shown in Image 1 (p. 1125) . The circle object's reference is saved in the reference variable named **circle1** .

Listing 6: Draw the solid green filled ellipse in the upper-left quadrant.

```
    //Upper left quadrant
Ellipse2D.Double circle1 =
         new Ellipse2D.Double(-128,-128,128,128);

//Solid GREEN fill
g2.setPaint(Color.GREEN);

g2.fill(circle1);

g2.draw(circle1);
```

5.409

Set the painting color

The statement near the middle of Listing 6 (p. 1131) sets the painting color to **Color.GREEN** .

(Note that Listing 6 (p. 1131) *calls* ***setPaint*** *whereas Listing 5 (p. 1129) calls* ***setColor*** *. I will leave it as an exercise for the student to study the documentation in order to understand the difference between* ***setColor*** *and* ***setPaint*** *.)*

Fill the circle referred to by circle1

The second statement from the bottom in Listing 6 (p. 1131) calls the **fill** method of the **Graphics2D** class passing the circle object's reference as a parameter. Although this can get quite complex, in this simple case, it causes the circle object to be filled with the paint color *(green)* .

Draw the filled circle object

Finally, the last statement in Listing 6 (p. 1131) calls the **draw** method of the **Graphics2D** class to cause the filled circle to be drawn inside the framing rectangle that was specified when the circle was

constructed. This results in the filled green circle in the upper-left quadrant in Image 1 (p. 1125) .

Draw a circle with a gradient fill in the upper-right quadrant

This is where things tend to get a little complicated.

If you compare the circles in the upper-left and upper-right quadrants in Image 1 (p. 1125) , you will see that they look considerably different.

Difference caused by parameter to the setPaint method

If you compare the code in Listing 6 (p. 1131) with the code in Listing 7 (p. 1132) , you will see that except for the coordinate values that position the circle, the only difference is the parameter that is passed to the **setPaint** method.

Listing 7: Draw a circle with a gradient fill in the upper-right quadrant.

```
    //Upper right quadrant
//Gradient GREEN to BLUE, cyclic along horizontal
// axis.
Ellipse2D.Double circle2 =
   new Ellipse2D.Double(0.0,-128,128,128);
g2.setPaint(new GradientPaint(
                64,0,Color.GREEN,
                64,-32,Color.BLUE,true));
g2.fill(circle2);
g2.draw(circle2);
```

5.410

A simple color paint versus a gradient paint

Listing 6 (p. 1131) passes an object of the simple **Color** class representing the color green to the **setPaint** method.

Listing 7 (p. 1132) passes an object of the **GradientPaint** class to the **setPaint** method.

Therefore, we need to understand the behavior of an object of the **GradientPaint** class.

Constructor parameters

There are four overloaded constructors for the **GradientPaint** class. The constructor used in Listing 7 is one of the most complicated. It requires the following parameters:

- x1 - x coordinate of the first specified Point in user space
- y1 - y coordinate of the first specified Point in user space
- color1 - Color at the first specified Point
- x2 - x coordinate of the second specified Point in user space
- y2 - y coordinate of the second specified Point in user space
- color2 - Color at the second specified Point
- cyclic - true if the gradient pattern should cycle repeatedly between the two colors; false otherwise

Two points and two colors

Basically, the class allows you to specify two points and two colors *(plus one additional boolean parameter)* when you construct an object of the class.

One of the colors is associated with each point.

An imaginary line segment

Think of the two points as being at the ends of an imaginary line segment, which can be at any angle relative to the horizontal.

Colors at the ends of the line segment

When a shape is drawn using a gradient fill, one of the colors will appear at one end of the line segment and the other color will appear at the other end of the line segment.

The color will change

The color will change gradually from one color to the other along the imaginary line segment connecting the two points.

Perpendicular color bands on the sides

The colors extend out to the sides of the imaginary line segment in bands that are perpendicular to the line segment.

The cyclic parameter

If the last *(cyclic)* parameter is set to false, the color will only change along the imaginary line segment. Areas beyond each end of the line segment will be the colors that are specified for the points at the ends of the line segment.

(This parameter was not set to false for any of the circles in Image 1.)

If the cyclic parameter is true...

The pattern of color change that occurs along the line segment will extend in a cyclic fashion beyond the ends of the line segment all the way to infinity.

Compare constructor parameters with upper-right quadrant

Now consider the parameter values used in Listing 7 (p. 1132) and compare them with the image in the upper-right quadrant in

A simple color paint versus a gradient paint

Listing 6 passes an object of the simple **Color** class representing the color green to the **setPaint** method.

Listing 7 passes an object of the **GradientPaint** class to the **setPaint** method.

Therefore, we need to understand the behavior of an object of the **GradientPaint** class.

Constructor parameters

There are four overloaded constructors for the **GradientPaint** class. The constructor highlighted in yellow in Listing 7 is one of the most complicated. It requires the following parameters:

- x1 - x coordinate of the first specified Point in user space
- y1 - y coordinate of the first specified Point in user space
- color1 - Color at the first specified Point
- x2 - x coordinate of the second specified Point in user space
- y2 - y coordinate of the second specified Point in user space
- color2 - Color at the second specified Point
- cyclic - true if the gradient pattern should cycle repeatedly between the two colors; false otherwise

Two points and two colors

Basically, the class allows you to specify two points and two colors *(plus one additional boolean parameter)* when you construct an object of the class.

One of the colors is associated with each point.

An imaginary line segment

Think of the two points as being at the ends of an imaginary line segment, which can be at any angle relative to the horizontal.

Colors at the ends of the line segment

When a shape is drawn using a gradient fill, one of the colors will appear at one end of the line segment and the other color will appear at the other end of the line segment.

The color will change

The color will change from one color to the other along the imaginary line segment connecting the two points.

Perpendicular color bands on the sides

The colors extend out to the sides of the imaginary line segment in bands that are perpendicular to the line segment.

The cyclic parameter

If the last *(cyclic)* parameter is set to false, the color will only change along the imaginary line segment. Areas beyond each end of the line segment will be the colors that are specified for the points at the ends of the line segment.

(This parameter was not set to false for any of the circles in Image 1.)

If the cyclic parameter is true...

The pattern of color change that occurs along the line segment will extend in a cyclic fashion beyond the ends of the line segment all the way to infinity.

Compare constructor parameters with upper-right quadrant

Now consider the parameter values used in Listing 7 (p. 1132) and compare them with the image in the upper-right quadrant in Image 1 (p. 1125) .

The location of the first point

The diameter of each circle in Image 1 (p. 1125) is 128 pixels. The first point for the upper-right quadrant in Image 1 (p. 1125) is on the X axis at the point where the circle touches the X axis. This point is specified to have a color of green.

The location of the second point

The second point is 32 pixels directly above the first point. Therefore, the imaginary line segment is perpendicular to the X axis. It extends from the X axis one-fourth of the way to the top of the circle. The second point is specified to have a color of blue.

The color change

As a result, the color changes from green to blue along the 32-pixel line segment. Color bands extend to the right and left, perpendicular to the line segment.

The cyclic parameter

The cyclic parameter is set to true, so the pattern repeats along the remaining vertical dimension of the circle. The color changes from blue back to green along the next 32-pixel vertical distance causing the circle to be green at the center.

The pattern repeats again resulting in a blue band three-fourths of the way from the bottom to the top of the circle and a green band at the top of the circle.

That's all there is to it

Now that you know the scheme, you should be able to examine the code in Listing 8 (p. 1135) for the remaining two quadrants and understand how the constructor parameters resulted in the color patterns that you see in Image 1 (p. 1125) .

Listing 8: Code for the remaining two quadrants.

```
        //Lower left quadrant
    //Gradient GREEN to BLUE, cyclic along vertical axis.
    //horizontal axis
    Ellipse2D.Double circle3 =
       new Ellipse2D.Double(-128,0.0,128,128);
    g2.setPaint(
            new GradientPaint(
                    -128,-64,Color.GREEN,
                    -107,-64,Color.BLUE,true));
    g2.fill(circle3);
    g2.draw(circle3);

    //Lower right quadrant
    //Gradient GREEN to BLUE, cyclic along
    // 45 degree angle
    Ellipse2D.Double circle4 =
        new Ellipse2D.Double(0,0,128,128);
    g2.setPaint(
            new GradientPaint(
                    19,19,Color.GREEN,
                    64,64,Color.BLUE,true));

    g2.fill(circle4);
    g2.draw(circle4);

  }//end process

}//end class Prob03Runner
```

5.411

An interesting anomaly

I will point out one interesting anomaly, however. Even though we shifted the origin to the center, we did not change the direction that represents positive values on the Y axis.

Coordinates above the center are negative and coordinates below the center are positive.

Where is the line segment?

If you examine the constructor parameters for the lower-left quadrant in Listing 8 (p. 1135) , you will see that the imaginary line segment isn't in the lower-left quadrant.

The line segment is 21 pixels in length, parallel to the horizontal axis. However, it is 64 pixels above the horizontal axis. That is not in the lower-left quadrant.

The effect is...

The effect is as though the color gradient fills the universe, and is based on a line segment placed anywhere in the universe.

However, we only see the color gradient through the shape that we specify as a parameter to the **fill** method. In other words, that shape is a window through which we can see a background consisting of

gradient color changes.

Our window is in the lower-left quadrant

In this case, that shape is the circle in the lower-left quadrant, so we see the gradient color effect in the lower-left quadrant.

How about the bottom-right quadrant

I stated earlier that the imaginary line segment can be at any angle relative to the horizontal axis.

For the bottom right quadrant, the line segment is 64 pixels in length. It lies along a line that goes through the origin and is at 45 degrees clockwise relative to the horizontal.

Where is the line segment positioned?

One end of the line segment is at the center of the circle. The other end of the line segment is at the point where the 45-degree line intersects the edge of the circle closest to the origin.

You can take it from there

You should be able to take it from there and explain the color gradient in the circle in the lower-right quadrant. Recall that the diameter of the circle is 128 pixels. The length of the line segment is 64 pixels or one-half the diameter.

The end of the class

Listing 8 (p. 1135) signals the end of the class named **Prob03Runner** and the end of the program.

5.3.39.5 Run the program

I encourage you to copy the code from Listing 9 (p. 1138) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. For example, try setting the cyclic constructor parameter to false and observe the effect. Make certain that you can explain why your changes behave as they do.

Click Prob03.jpg [219] to download the input image file if you elect to use it.

5.3.39.6 Summary

In this module, you learned to use the **GradientPaint** class along with a variety of other Java2D classes.

5.3.39.7 What's next?

In the next module, you will Learn how to use shapes to clip images during the drawing process.

5.3.39.8 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 13 [220]

5.3.39.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: GradientPaint and other Java2D Classes
- File: Java3026.htm
- Published: 08/04/12
- Revised: 02/18/13

[219] http://cnx.org/content/m44242/latest/Prob03.jpg
[220] http://vimeo.com/channels/itse2321/21220418

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.39.10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 9 (p. 1138) below.

Listing 9: Complete program listing.

```
    /*File Prob03 Copyright 2008 R.G.Baldwin
**********************************************************/
import java.awt.geom.Line2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GradientPaint;

public class Prob03{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
//========================================================//

class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //--------------------------------------------------//
  public void run(){
    Picture pic = new Picture("Prob03.jpg");
    pic.setAllPixelsToAColor(Color.RED);

    process(pic);
    //Add your name and display the output picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    System.out.println(pic);
  }//end run
  //--------------------------------------------------//

  private void process(Picture pic){

    Graphics2D g2 = (Graphics2D)(pic.getGraphics());

    int width = pic.getWidth();
    int height = pic.getHeight();

    //Translate origin to center of Frame
    g2.translate(width/2,height/2);
    g2.setColor(Color.BLACK);
    //Draw x-axis
    g2.draw(new Line2D.Double(-width/2,0.0,width/2,0.0));
    //Draw y-axis
    g2.draw(new Line2D.Double(0.0,-width/2,0.0,height/2));

    //Upper left quadrant, Solid GREEN fill
    Ellipse2D.Double circle1 =
      new Ellipse2D.Double(-128,-128,128,128);
    g2.setPaint(Color.GREEN);
    g2.fill(circle1);
    g2.draw(circle1);
```

-end-

5.3.40 Java3026r Review[221]

5.3.40.1 Table of Contents

5.3.40.2 Preface

This module contains review questions and answers keyed to the module titled Java3026: GradientPaint and other Java2D Classes [222] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.40.3 Questions

5.3.40.3.1 Question 1 .

Which of the following output images is produced by the program shown in Listing 1 (p. 1141) ?

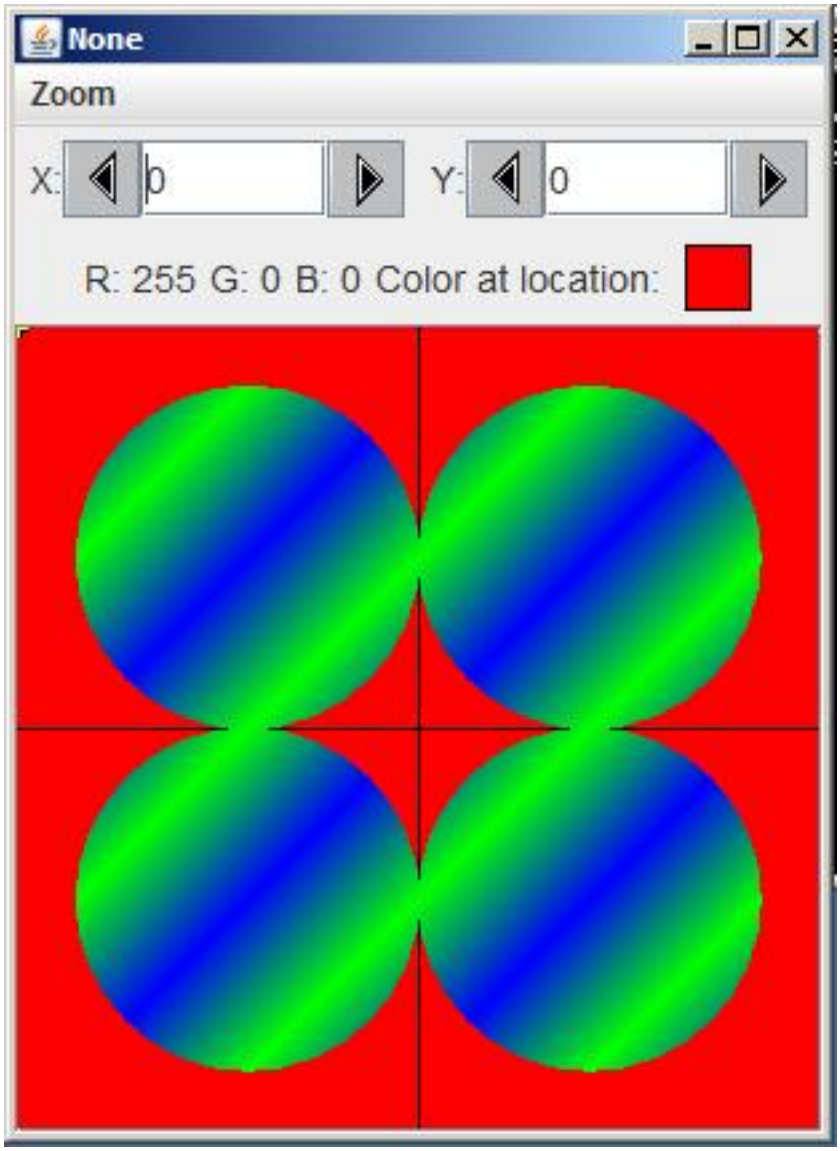
A. Image 1 (p. 1142)
B. Image 2 (p. 1143)

[221] This content is available online at <http://cnx.org/content/m45785/1.1/>.
[222] http://cnx.org/content/m44242

Listing 1. Question 1.

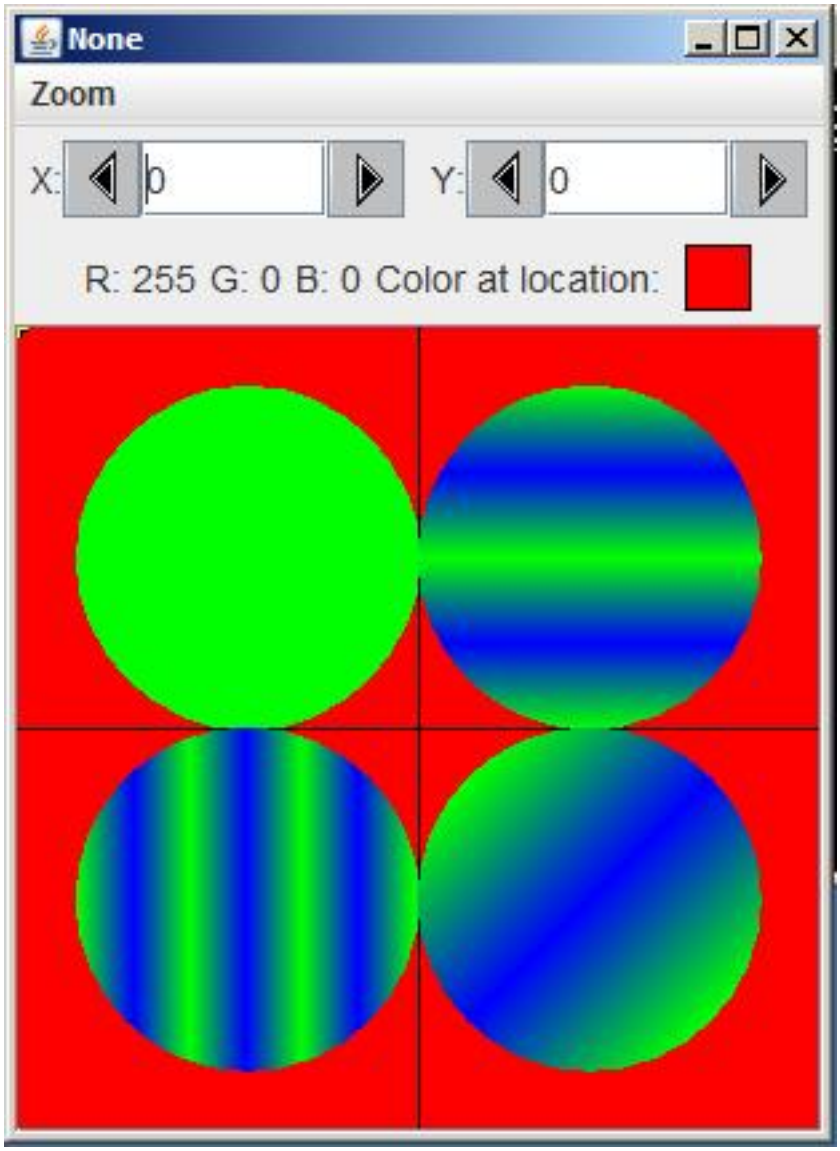
```
/*File Java3026ra Copyright 2013 R.G.Baldwin
*********************************************************/
import java.awt.geom.Line2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GradientPaint;

public class Java3026ra{
  public static void main(String[] args){
    new Java3026raRunner().run();
  }//end main method
}//end class Java3026ra
//=======================================================//

class Java3026raRunner{
  public void run(){
    Picture pic = new Picture(300,300);
    pic.setAllPixelsToAColor(Color.RED);
    process(pic);
    pic.explore();
  }//end run
  //-----------------------------------------------------//

  private void process(Picture pic){

    Graphics2D g2 = (Graphics2D)(pic.getGraphics());

    int width = pic.getWidth();
    int height = pic.getHeight();

    g2.translate(width/2,height/2);
    g2.setColor(Color.BLACK);
    g2.draw(new Line2D.Double(-width/2,0.0,width/2,0.0));
    g2.draw(new Line2D.Double(0.0,-width/2,0.0,height/2));
    Ellipse2D.Double circle1 =
                  new Ellipse2D.Double(-128,-128,128,128);

    g2.setPaint(new GradientPaint(-64,-64,Color.BLUE,
                                -32,-32,Color.GREEN,true));
    g2.fill(circle1);
    g2.draw(circle1);

    Ellipse2D.Double circle2 =
                   new Ellipse2D.Double(0.0,-128,128,128);
    g2.fill(circle2);
    g2.draw(circle2);

    Ellipse2D.Double circle3 =
                   new Ellipse2D.Double(-128,0.0,128,128);
    g2.fill(circle3);
    g2.draw(circle3);
```

Image 1. Possible output image.



5.414

Image 2. Possible output image.



5.415

Answer 1 (p. 1145)

5.3.40.4 Images

- Image 1 (p. 1142) . Possible output image.
- Image 2 (p. 1143) . Possible output image.

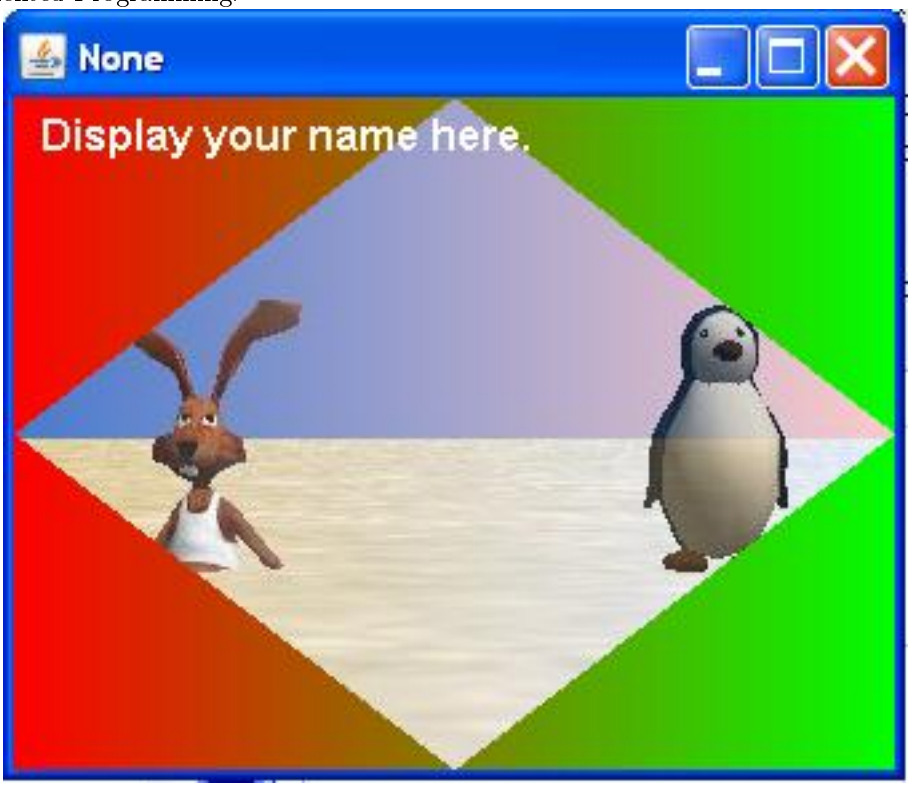
5.3.40.5 Listings

- Listing 1 (p. 1141) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.40.6 Answers

5.3.40.6.1 Answer 1

The code in Listing 1 (p. 1141) produces the output image shown in Image 1 (p. 1142) .
Back to Question 1 (p. 1140)

5.3.40.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3026r Review
- File: Java3026r.htm
- Published: 02/18/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.41 Java3026s Slides[223]

5.3.41.1 Table of Contents

5.3.41.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3026: GradientPaint and other Java2D Classes [224] .

Click here [225] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.41.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3026s Slides
- File: Java3026s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[223] This content is available online at <http://cnx.org/content/m45643/1.2/>.
[224] http://cnx.org/content/m44242
[225] http://cnx.org/content/m45643/latest/a0-Index.htm

-end-

5.3.42 Java3028: Clipping Images[226]

5.3.42.1 Table of Contents

5.3.42.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [227] .

5.3.42.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.42.2.1.1 Images

5.3.42.2.1.2 Listings

[226] This content is available online at <http://cnx.org/content/m44246/1.6/>.
[227] http://cnx.org/content/m44148/latest/

5.3.42.3 Preview

In this module, you will learn how to use shapes to clip images during the drawing process.

Program specifications

Write a program named **Prob04** that uses the class definition shown in Listing 1 (p. 1151) and Ericson's media library along with the image file named **Prob04a.jpg** *(see Image 1 (p. 1148))* to produce the graphic output images shown in Image 2 (p. 1149) and Image 3 (p. 1150) . Don't forget to display your name in the output image as shown.

Image 1: Input file named Prob04a.jpg.

Figure 5.416: Image 1: Input file named Prob04a.jpg.

Image 2: First output image.

Figure 5.417: Image 2: First output image.

Image 3: Second output image.

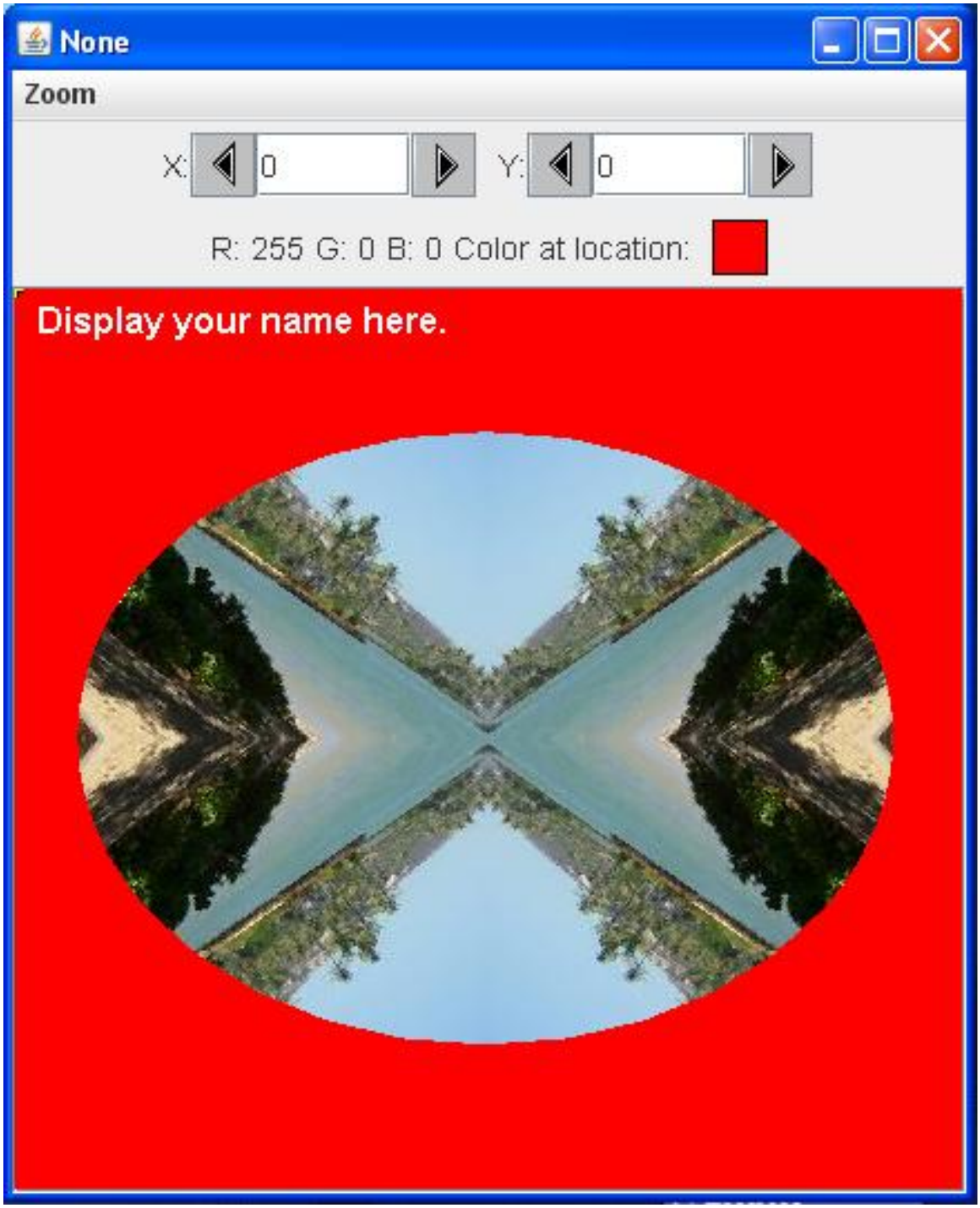


Figure 5.418: Image 3: Second output image.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob04** shown in Listing 1 (p. 1151) .

Rotate, mirror, and clip

The program rotates a **Picture** object by 35 degrees with no scaling. Then it does a four-way mirror on the rotated picture. Finally, it clips the image to an elliptical format as shown in Image 3 (p. 1150) .

Required output text

In addition to the two output images shown above, your program must display your name and the other line of text shown in Image 4 (p. 1151) .

Image 4: Required text output.

```
    Display your name here.
Picture, filename None height 404 width 425
```

Figure 5.419: Image 4: Required text output.

5.3.42.4 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 5 (p. 1157) near the end of the module.

The driver class named Prob04

The driver class containing the **main** method is shown in Listing 1 (p. 1151) .

Listing 1: The driver class named Prob04.

```
    public class Prob04{
  public static void main(String[] args){
    new Prob04Runner().run();
  }//end main method
}//end class Prob04
```

Figure 5.420: Listing 1: The driver class named Prob04.

If you have been studying the earlier modules in this collection, no explanation of Listing 1 (p. 1151) should be required.

Beginning of the class named Prob04Runner

The class named **Prob04Runner** begins in Listing 2 (p. 1152) .

Listing 2: Beginning of the class named Prob04Runner.

```
  class Prob04Runner{

public Prob04Runner(){
  System.out.println("Display your name here.");
}//end constructor
//-------------------------------------------------//

public void run(){
  Picture pix = new Picture("Prob04a.jpg");

  //Rotate and mirror the picture.
  pix = rotatePicture(pix,35);
  pix = mirrorUpperQuads(pix);
  pix = mirrorHoriz(pix);

  pix.explore();
```

Figure 5.421: Listing 2: Beginning of the class named Prob04Runner.

Nothing new here

There is nothing new in Listing 2 (p. 1152) .

After instantiating a new **Picture** object from the given image file, Listing 2 (p. 1152) calls three methods to rotate, mirror, and display the picture, producing the graphic output shown in Image 2 (p. 1149) .

All of the code to accomplish this is essentially the same as code that I have explained in earlier modules.

Clip the picture and display your name

Then Listing 3 (p. 1153) calls the **clipToEllipse** method to clip the picture to an ellipse on a red background as shown in Image 3 (p. 1150) . The **clipToEllipse** method is new to this module, so I will explain it shortly.

Listing 3: Clip the picture and display your name.

```
    pix = clipToEllipse(pix);

  //Add your name and display the output picture.
  pix.addMessage("Display your name here.",10,20);
  pix.explore();

  System.out.println(pix);
}//end run
```

Figure 5.422: Listing 3: Clip the picture and display your name.

The remaining code in Listing 3 (p. 1153) is a repeat of code that I have explained in earlier modules, so I won't have anything further to say about it.

The method named clipToEllipse

The method named **clipToEllipse** is shown in its entirety in Listing 4 (p. 1154) .

Listing 4: The method named clipToEllipse.

```
  private Picture clipToEllipse(Picture pix){
Picture result =
          new Picture(pix.getWidth(),pix.getHeight());
result.setAllPixelsToAColor(Color.RED);

//Get the graphics2D object
Graphics2D g2 = (Graphics2D)(result.getGraphics());

//Create an ellipse for clipping
Ellipse2D.Double ellipse =
  new Ellipse2D.Double(28,64,366,275);

//Use the ellipse for clipping
g2.setClip(ellipse);

//Draw the image
g2.drawImage(pix.getImage(),0,0,pix.getWidth(),
                            pix.getHeight(),
                            null);

return result;
}//end clipToEllipse
```

Figure 5.423: Listing 4: The method named clipToEllipse.

Behavior of the clipToEllipse method

The **clipToEllipse** method receives an incoming parameter that is a reference to an object of the **Picture** class. Basically, here is what the method does:

- Instantiate a **Picture** object with an all white background that is the same size as the incoming **Picture** object.
- Call Ericson's **setAllPixelsToAColor** method to convert the white background into a red background.
- Call Ericson's **getGraphics** method to get the **Graphics** object encapsulated in the red **Picture** object.
- Cast the **Graphics** object's reference to type **Graphics2D** .
- Construct a new **Ellipse2D.Double** object with the position, width, and height specified by the constructor parameters.
- Call Sun's **setClip** method to set the clipping area on the red **Picture** object to match the position and shape of the ellipse.
- Call Ericson's **getImage** method to get the **Image** object encapsulated in the incoming **Picture** object.
- Call Sun's **drawImage** method to draw that portion of the incoming picture that fits inside the ellipse on the red **Picture** object.

The new code

The only code in Listing 4 (p. 1154) that is new to this module is the call to the **setClip** method.

The **setClip** method is defined in the **Graphics** class and inherited into the **Graphics2D** class.

(Among other things, that means that it wasn't necessary for me to cast the ***Graphics*** *object to type* ***Graphics2D*** *in Listing 4 (p. 1154) .)*

The setClip method

There are a couple of overloaded versions of the **setClip** method. The one used in Listing 4 (p. 1154) requires an incoming parameter of the interface type **Shape** .

The Shape interface

Briefly, Sun tells us that the **Shape** interface *"provides definitions for objects that represent some form of geometric shape."*

There are several dozen classes that implement the **Shape** interface, one of which is the class named **Ellipse2D.Double** . Therefore, the object of that type that is instantiated in Listing 4 (p. 1154) satisfies the type requirement for being passed to the **setClip** method.

Behavior of the setClip method

With regard to the behavior of the **setClip** method, Sun tells us that the method

"Sets the current clipping area to an arbitrary clip shape."

What is the significance of the clipping area?

The closest answer that I can find for that question is the following statement in Sun's description of the **Graphics** class:

"All rendering operations modify only pixels which lie within the area bounded by the current clip, which is specified by a ***Shape*** *in user space and is controlled by the program using the* ***Graphics*** *object."*

In other words...

The *clipping area* is analogous to the *current clip* . In this case, the position and shape of the current clip is the position and shape of the ellipse.

When the image is later drawn on the red **Picture** object, only those pixels within the ellipse are modified to show the image. The remaining pixels retain their original color, which was set to red early in Listing 4 (p. 1154) .

End of discussion

That concludes my explanation of this program. You will find the methods that I didn't discuss in Listing 5 (p. 1157) near the end of the module.

5.3.42.5 Run the program

I encourage you to copy the code from Listing 5 (p. 1157) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click Prob04a.jpg [228] to download the required input image file.

5.3.42.6 Summary

In this module, you learned how to use shapes to clip images during the drawing process.

5.3.42.7 What's next?

In the next module, you will learn how to merge pictures.

5.3.42.8 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 14 [229]

[228] http://cnx.org/content/m44246/latest/Prob04a.jpg
[229] http://vimeo.com/channels/itse2321/21221510

5.3.42.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Clipping Images
- File: Java3028.htm
- Published: 08/06/12
- Revised: 02/18/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.3.42.10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 5 (p. 1157) below.

Listing 5: Complete program listing.

```
    /*File Prob04 Copyright 2008 R.G.Baldwin
**********************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;
import java.awt.geom.Ellipse2D;
import java.awt.Color;

public class Prob04{
  public static void main(String[] args){
    new Prob04Runner().run();
  }//end main method
}//end class Prob04
//========================================================//

class Prob04Runner{
  public Prob04Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-------------------------------------------------------//
  public void run(){
    Picture pix = new Picture("Prob04a.jpg");

    //Rotate and mirror the picture.
    pix = rotatePicture(pix,35);
    pix = mirrorUpperQuads(pix);
    pix = mirrorHoriz(pix);

    pix.explore();

    //Clip the picture to an ellipse on a red background.
    pix = clipToEllipse(pix);

    //Add your name and display the output picture.
    pix.addMessage("Display your name here.",10,20);
    pix.explore();

    System.out.println(pix);
  }//end run
  //-------------------------------------------------------//

  private Picture clipToEllipse(Picture pix){
    Picture result =
             new Picture(pix.getWidth(),pix.getHeight());
    result.setAllPixelsToAColor(Color.RED);

    //Get the graphics2D object
    Graphics2D g2 = (Graphics2D)(result.getGraphics());

    //Create an ellipse for clipping
    Ellipse2D.Double ellipse =
      new Ellipse2D.Double(28,64,366,275);

    //Use the ellipse for clipping
```

-end-

5.3.43 Java3028r Review[230]

5.3.43.1 Table of Contents

5.3.43.2 Preface

This module contains review questions and answers keyed to the module titled Java3028: Clipping Images [231] .

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.43.3 Questions

5.3.43.3.1 Question 1 .

Given the input image shown in Image 1 (p. 1161) , which of the following output images is produced by the code in Listing 1 (p. 1160) ?

A. Image 2 (p. 1162)
B. Image 3 (p. 1163)

[230] This content is available online at <http://cnx.org/content/m45786/1.1/>.
[231] http://cnx.org/content/m44246

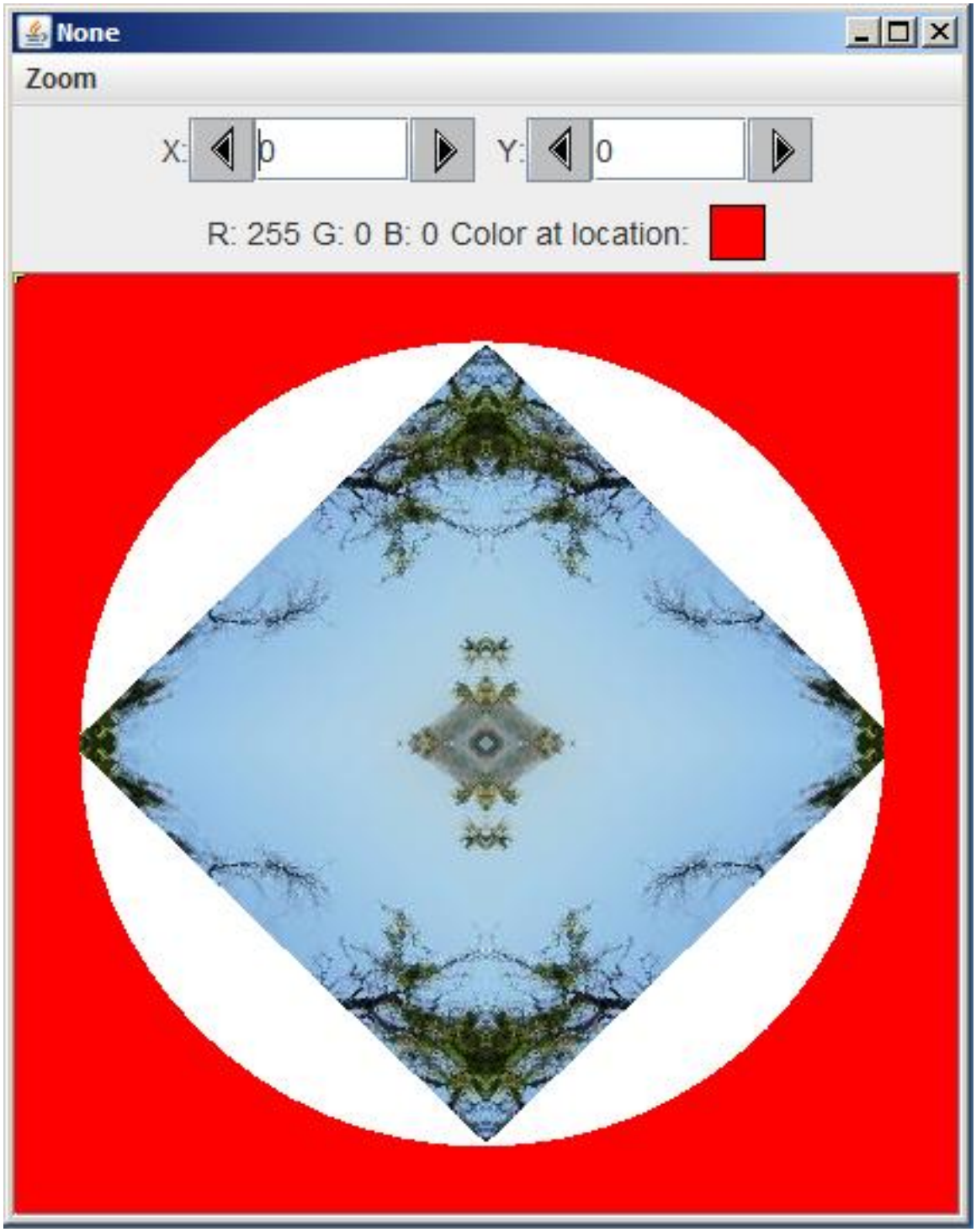
Listing 1. Question 1.

```
/*File Java3028ra Copyright 2013 R.G.Baldwin
*********************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;
import java.awt.geom.Ellipse2D;
import java.awt.Color;

public class Java3028ra{
  public static void main(String[] args){
    new Java3028raRunner().run();
  }//end main method
}//end class Java3028ra
//=======================================================//

class Java3028raRunner{

  public void run(){
    Picture pix = new Picture("Prob04a.jpg");
    pix = rotatePicture(pix,35);
    pix = mirrorUpperQuads(pix);
    pix = mirrorHoriz(pix);
    pix = clipToEllipse(pix);
    pix.explore();
  }//end run
  //-----------------------------------------------------//

  private Picture clipToEllipse(Picture pix){
    Picture result =
             new Picture(pix.getWidth(),pix.getHeight());
    result.setAllPixelsToAColor(Color.RED);
    Graphics2D g2 = (Graphics2D)(result.getGraphics());
    Ellipse2D.Double ellipse =
                      new Ellipse2D.Double(28,64,366,275);
    g2.setClip(ellipse);
    g2.drawImage(pix.getImage(),0,0,pix.getWidth(),
                                          pix.getHeight(),
                                                    null);
    return result;
  }//end clipToEllipse
  //-----------------------------------------------------//

  private Picture rotatePicture(Picture pix,
                                double angle){

    AffineTransform rotateTransform =
                                     new AffineTransform();
    rotateTransform.rotate(Math.toRadians(angle),
                           pix.getWidth()/2,
                           pix.getHeight()/2);

    Rectangle2D rectangle2D =
        pix.getTransformEnclosingRect(rotateTransform);
```

Image 1. Prob04a.jpg

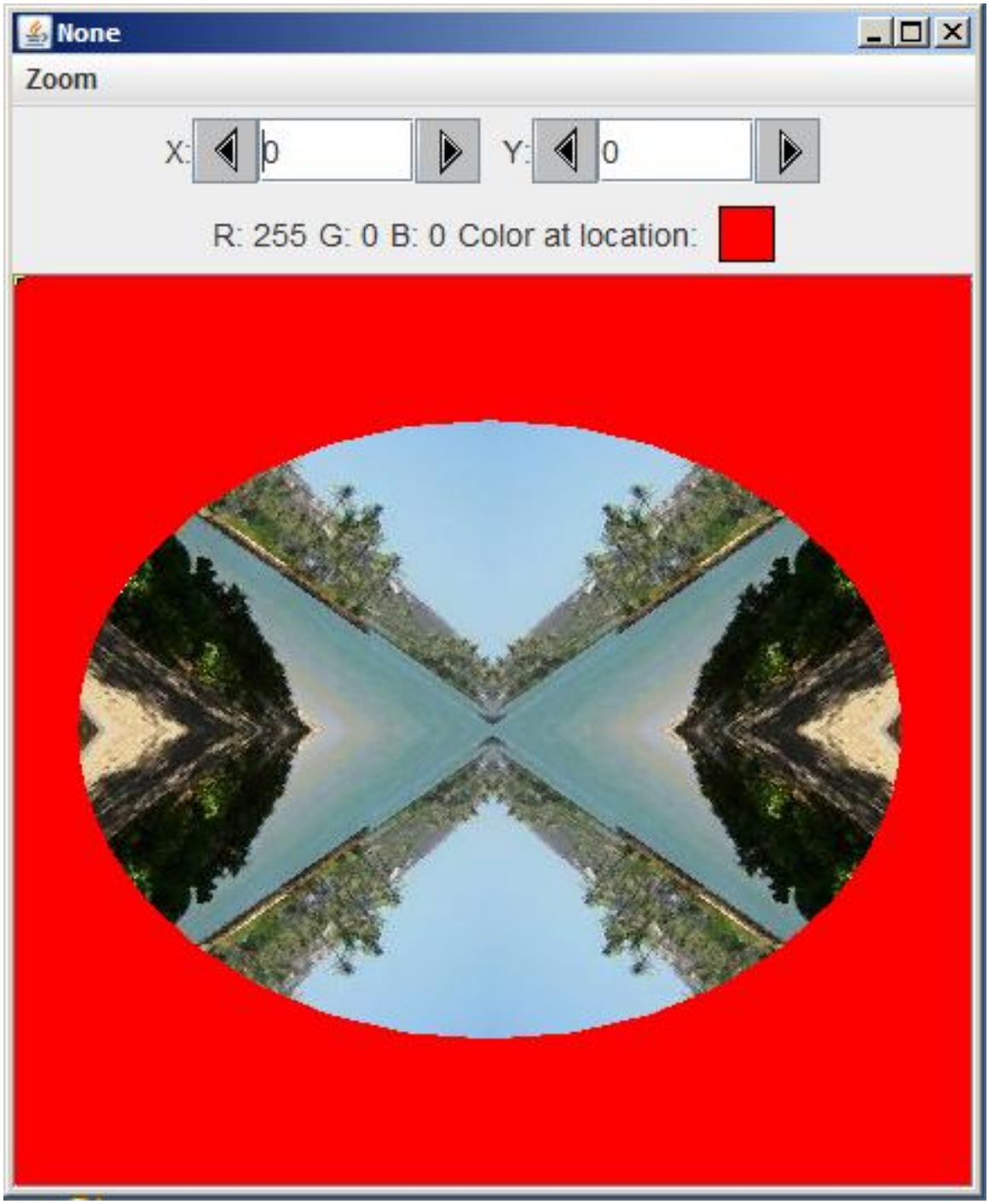
5.426

Image 2. Possible output image.



5.427

Image 3. Possible output image.



5.428

Answer 1 (p. 1165)

5.3.43.4 Images

- Image 1 (p. 1161) . Prob04a.jpg
- Image 2 (p. 1162) . Possible output image.
- Image 3 (p. 1163) . Possible output image.

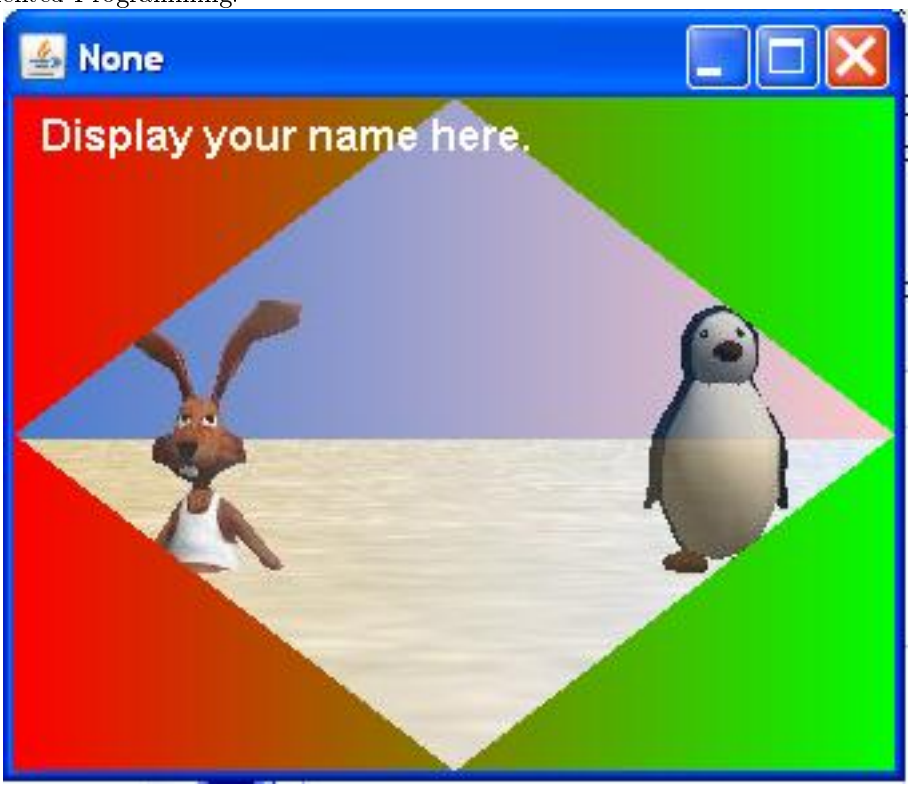
5.3.43.5 Listings

- Listing 1 (p. 1160) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.43.6 Answers

5.3.43.6.1 Answer 1

The code in Listing 1 (p. 1160) produces the output image shown in Image 3 (p. 1163) .
Back to Question 1 (p. 1159)

5.3.43.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3028r Review
- File: Java3028r.htm
- Published: 02/18/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.44 Java3028s Slides[232]

5.3.44.1 Table of Contents

5.3.44.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3028: Clipping Images [233] .

Click here [234] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.44.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3028s Slides
- File: Java3028s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[232] This content is available online at <http://cnx.org/content/m45646/1.2/>.
[233] http://cnx.org/content/m44246
[234] http://cnx.org/content/m45646/latest/a0-Index.htm

5.3.45 Java3030: Merging Pictures[235]

5.3.45.1 Table of Contents

5.3.45.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [236] .

5.3.45.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

5.3.45.2.1.1 Images

5.3.45.2.1.2 Listings

[235] This content is available online at <http://cnx.org/content/m44247/1.6/>.
[236] http://cnx.org/content/m44148/latest/

5.3.45.3 Preview

In this module, you will learn how to do a linear merge on two pictures based on the distance of each pixel from the left side of the picture.

Program specifications

Write a program named **Prob05** that uses the class definition shown in Listing 1 (p. 1171) and Ericson's media library along with the image files named **Prob05a.jpg** and **Prob05b.jpg** *(see Image 1 (p. 1168) and Image 2 (p. 1169))* to produce the graphic output image shown in Image 3 (p. 1170) .

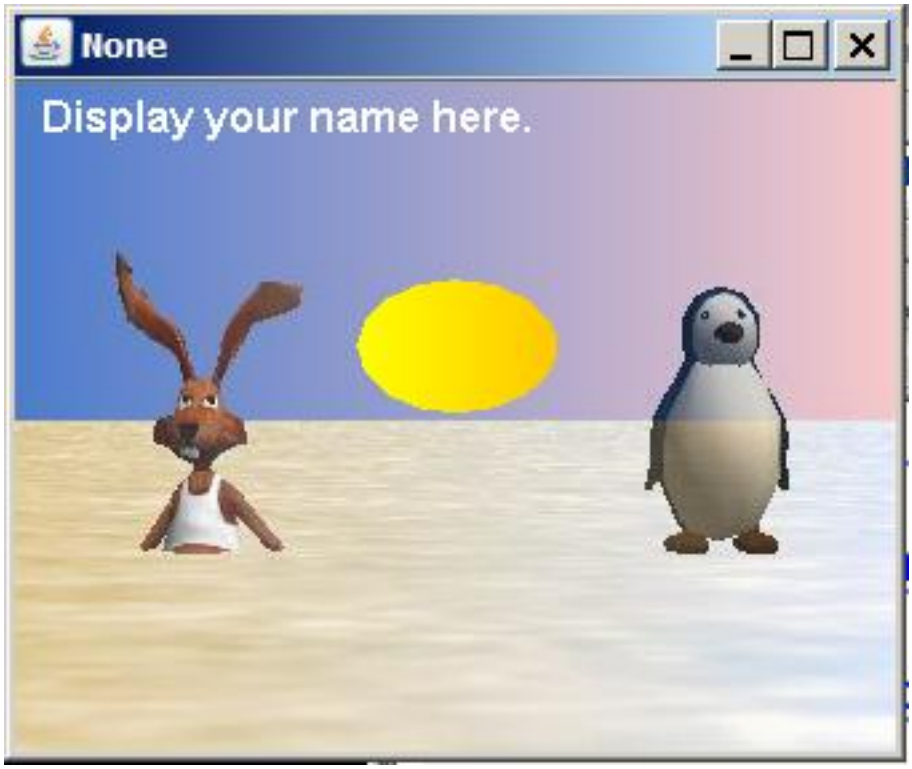
Image 1: Input file named Prob05a.jpg.

5.429

Image 2: Input file named Prob05b.jpg.

5.430

Image 3: Required graphic output image.



5.431

Required output text

In addition to the output image mentioned above, your program must display your name and the other line of text shown in Image 4 on the command-line screen.

Image 4: Required output text.

```
    Display your name here.
Picture, filename None height 252 width 330
```

5.432

5.3.45.4 Discussion and sample code

This program does a linear merge on two pictures based on the distance of each pixel from the left side of the picture. The program also adds a sun with a gradient and the student's name to the picture.

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 6 (p. 1177) near the end of the module.

The driver class named Prob05

The driver class containing the **main** method is shown in Listing 1 (p. 1171) .

Listing 1: The driver class named Prob05.

```
  public class Prob05{
public static void main(String[] args){
  new Prob05Runner().run();
}//end main method
}//end class Prob05
```

5.433

The code in Listing 1 (p. 1171) shouldn't require an explanation at this stage in the course.

Beginning of the class named Prob05Runner

The class named **Prob05Runner** begins in Listing 2 (p. 1171) .

Listing 2: Beginning of the class named Prob05Runner.

```
  class Prob05Runner{
public Prob05Runner(){
  System.out.println("Display your name here.");
}//end constructor
```

5.434

As with Listing 1 (p. 1171) , the code in Listing 2 (p. 1171) shouldn't require an explanation.

The run method of the Prob05Runner class

Listing 1 (p. 1171) calls the **run** method on an object of the **Prob05Runner** class. The **run** method is shown in its entirety in Listing 3 (p. 1172) .

Listing 3: The run method of the Prob05Runner class.

```
  public void run(){
 Picture penguin = new Picture("Prob05a.jpg");
 Picture hare = new Picture("Prob05b.jpg");
 merge(hare,penguin);
 hare = crop(hare,6,58,330,252);
 hare.addMessage("Display your name here.",10,20);
 drawSun(hare);
 hare.show();
 System.out.println(hare);
}//end run
```

5.435

The only thing in Listing 3 (p. 1172) that I haven't explained in earlier modules is the call to the **merge** method, so I will limit my discussion to that method.

The merge method

The **merge** method is used to merge the image in Image 1 (p. 1168) with the image in Image 2 (p. 1169) to produce the image shown in Image 3 (p. 1170) .

(Note, however, that the merged image was cropped to eliminate the buttons at the top of F igure 1 (p. 1168) and Image 2 (p. 1169) before displaying it in Image 3 (p. 1170) .)

A linear merge

The **merge** method does a linear merge on two pictures based on the distance of each pixel from the left side of the picture.

The method assumes that both pictures have the same dimensions.

Beginning of the merge method

The **merge** method begins in Listing 4 (p. 1173) .

Listing 4: Beginning of the merge method.

```
  private void merge(Picture left,Picture right){
int width = left.getWidth();
int height = left.getHeight();

double scaleL = 0;
double scaleR = 0;
int redL = 0;
int greenL = 0;
int blueL = 0;
int redR = 0;
int greenR = 0;
int blueR = 0;
Pixel pixelL = null;
Pixel pixelR = null;
```

5.436

The code in Listing 4 (p. 1173) simply declares and initializes a large number of working variables.

Do the merge

The merge is accomplished in the nested **for** loop in Listing 5 (p. 1174) .

Listing 5: Do the merge.

```
      for(int row = 0;row < height;row++){
    for(int col = 0;col < width;col++){
      scaleR = (double)col/width;
      scaleL = 1.0 - scaleR;
      pixelL = left.getPixel(col,row);
      pixelR = right.getPixel(col,row);

      redL = pixelL.getColor().getRed();
      greenL = pixelL.getColor().getGreen();
      blueL = pixelL.getColor().getBlue();

      redR = pixelR.getColor().getRed();
      greenR = pixelR.getColor().getGreen();
      blueR = pixelR.getColor().getBlue();

      redL = (int)(redL*scaleL + redR*scaleR);
      greenL = (int)(greenL*scaleL + greenR*scaleR);
      blueL = (int)(blueL*scaleL + blueR*scaleR);

      pixelL.setColor(new Color(redL,greenL,blueL));
    }//end inner loop
  }//end outer loop
}//end merge
```

5.437

Difficult to explain

Although the code in Listing 5 (p. 1174) is long, tedious, and ugly, it isn't complicated However, it is somewhat difficult to explain in words.

Two scale factors

The body of the **for** loop begins by computing a pair of scale factors named **scaleR** and **scaleL** . The factor named **scaleR** has a maximum value of 1.0 and is directly proportional to the distance of the current pixel from the left edge of the picture.

The factor named **scaleL** also has a maximum value of 1.0 and is inversely proportional to the distance of the pixel from the left edge.

Get and save color components

The red, green, and blue values for the same pixel location in each of the pictures are obtained and saved.

Compute a new set of color values

A new set of red, green, and blue color values are computed as the sum of scaled versions of the pixel colors from the rabbit image pixel and the penguin image pixel.

The nature of the scaling

The scaling is such that the pixel colors from the rabbit image contribute most heavily to output pixels to the left of center and pixel colors from the penguin image contribute most heavily to output pixels to the right of center.

You can see the effect of this scaling algorithm in Image 3 (p. 1170) .

The pixels in the horizontal center

The pixels along a vertical line at the center of the output image contain equal contributions of colors from both images.

End of discussion

That concludes the explanation of the code in this program.

You can view the **drawSun** and **crop** methods in Listing 6 (p. 1177) near the end of the module.

5.3.45.5 Run the program

I encourage you to copy the code from Listing 6 (p. 1177) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click Prob05a.jpg [237] and Prob05b.jpg [238] to download the required input image files.

5.3.45.6 Summary

In this module, you learned how to merge two pictures.

5.3.45.7 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 15 [239]

5.3.45.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Merging Pictures
- File: Java3030.htm
- Published: 08/07/12
- Revised: 01/01/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

[237] http://cnx.org/content/m44247/latest/Prob05a.jpg
[238] http://cnx.org/content/m44247/latest/Prob05b.jpg
[239] http://vimeo.com/channels/itse2321/21222484

5.3.45.9 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 6 (p. 1177) below.

Listing 6: Complete program listing.

```
    /*File Prob05 Copyright 2008 R.G.Baldwin
************************************************************/
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Graphics;
import java.awt.GradientPaint;
import java.awt.geom.Ellipse2D;

public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob05Runner().run();
  }//end main method
}//end class Prob05
//============================================================//

class Prob05Runner{
  public Prob05Runner(){
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------------//
  public void run(){
    Picture penguin = new Picture("Prob05a.jpg");
    Picture hare = new Picture("Prob05b.jpg");
    merge(hare,penguin);
    hare = crop(hare,6,58,330,252);
    hare.addMessage("Display your name here.",10,20);
    drawSun(hare);
    hare.show();
    System.out.println(hare);
  }//end run
  //-----------------------------------------------------------//

  private void drawSun(Picture pic){

    Graphics2D g2d = (Graphics2D)(pic.getGraphics());
    int width = 75;
    int height = 50;
    int center = pic.getWidth()/2;
    int xCoor = center - width/2;
    int yCoor = 75;


    //Create the gradient for painting from yellow to red
    // with yellow at the left of the sun and red at the
    // right.
    GradientPaint gPaint = new GradientPaint(
                               xCoor, yCoor+height/2,
                               Color.YELLOW,
                               xCoor+width,yCoor+height/2,
                               Color.ORANGE);

    //Set the gradient and draw the ellipse
    g2d.setPaint(gPaint);
    g2d.fill(new Ellipse2D.Double(
```

-end-

5.3.46 Java3030r Review[240]

5.3.46.1 Table of Contents

5.3.46.2 Preface

This module contains review questions and answers keyed to the module titled Java3030: Merging Pictures [241].

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

5.3.46.3 Questions

5.3.46.3.1 Question 1 .

Given the two input images shown in Image 1 (p. 1181) and Image 2 (p. 1182) , which of the following two output images is produced by the code in Listing 1 (p. 1180) ? Note that the differences in the two possible output images are subtle. Also note the RGB color values shown at the same cursor location in all four images.

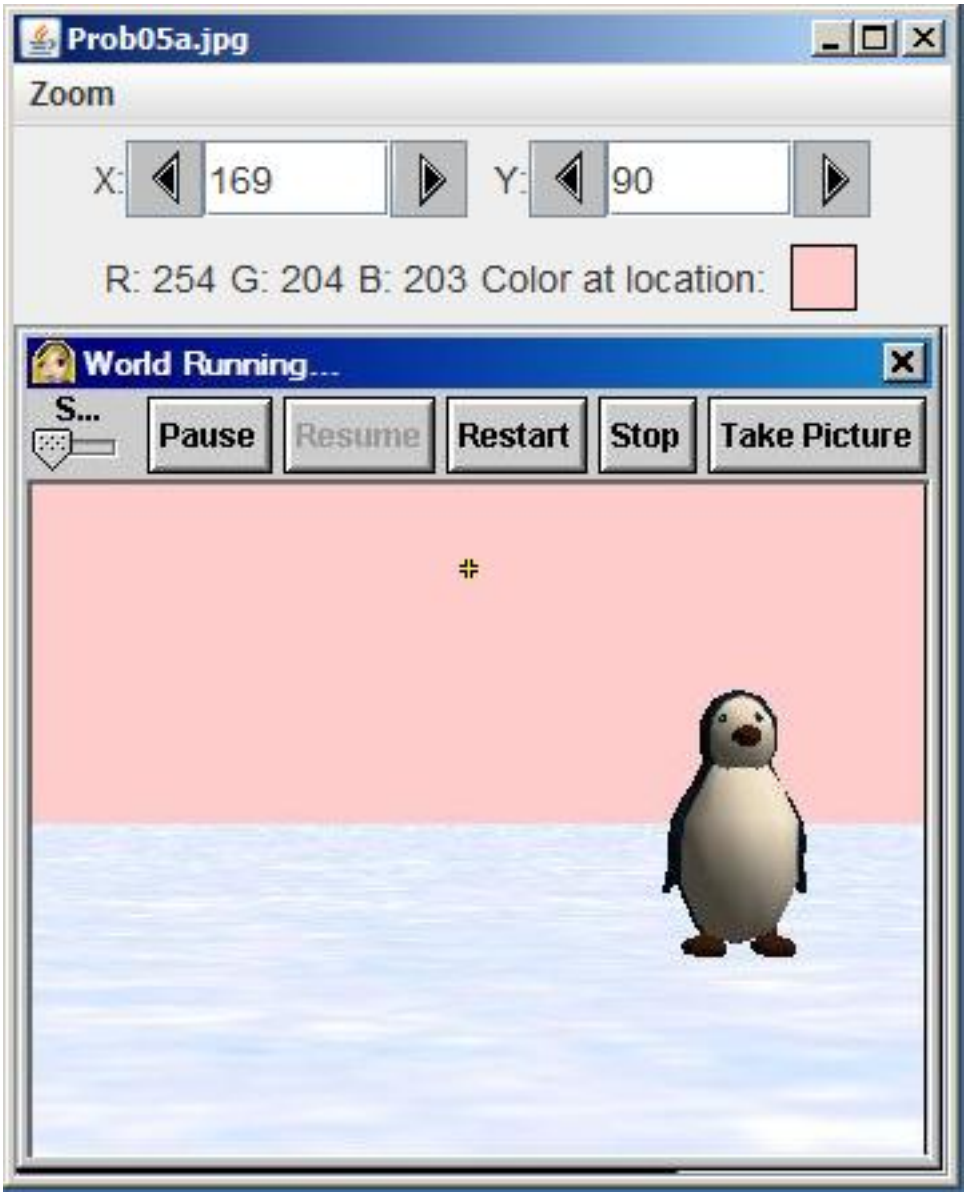
A. Image 3 (p. 1183)
B. Image 4 (p. 1184)

[240] This content is available online at <http://cnx.org/content/m45787/1.1/>.
[241] http://cnx.org/content/m44247

Listing 1. Question 1.

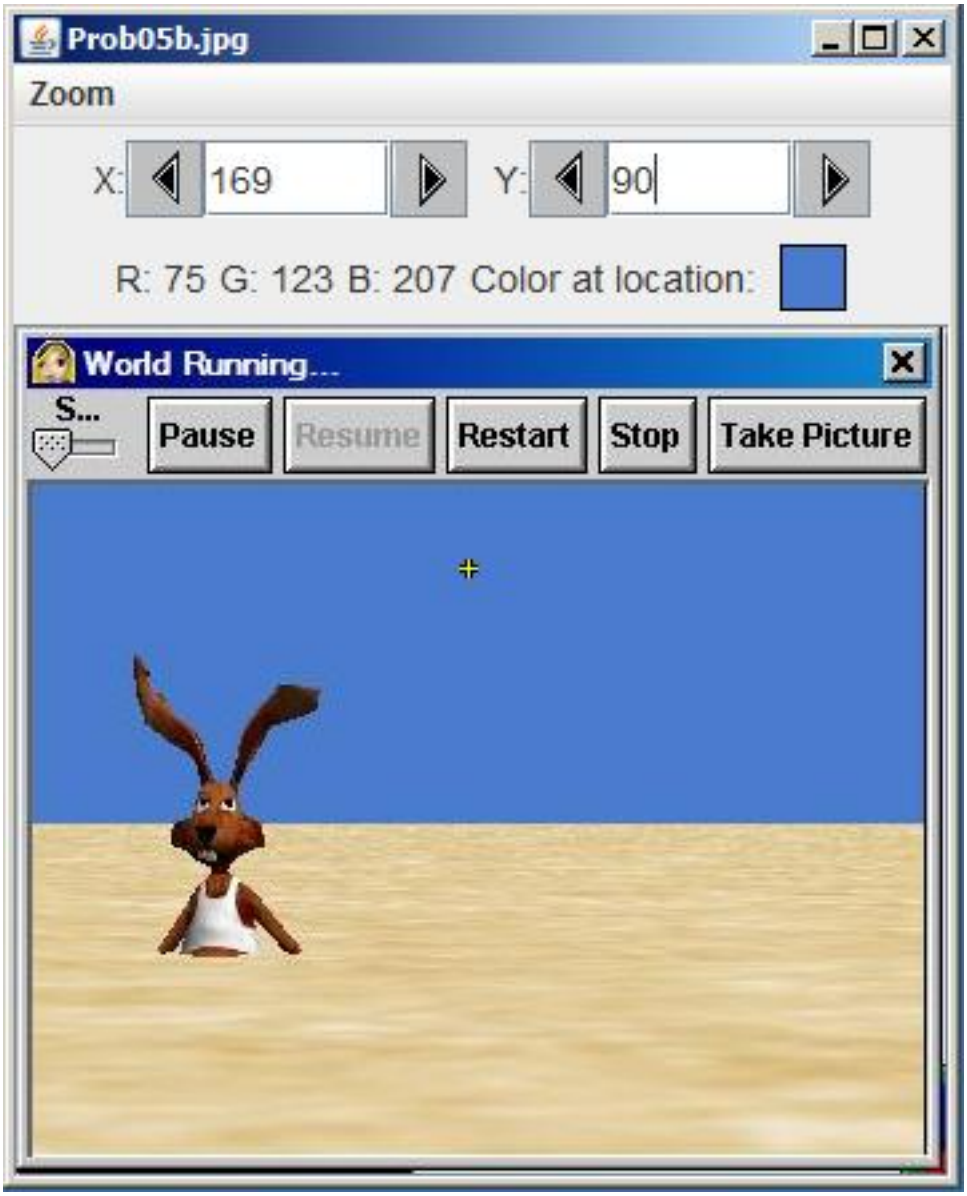
```
/*File Java3030ra Copyright 2013 R.G.Baldwin
************************************************************/
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Graphics;
import java.awt.GradientPaint;
import java.awt.geom.Ellipse2D;

public class Java3030ra{
  public static void main(String[] args){
    new Java3030raRunner().run();
  }//end main method
}//end class Java3030ra
//============================================================//

class Java3030raRunner{
  public void run(){
    Picture penguin = new Picture("Prob05a.jpg");
    penguin.explore();
    Picture hare = new Picture("Prob05b.jpg");
    hare.explore();
    merge(hare,penguin);
    hare.explore();
  }//end run
  //----------------------------------------------------//

  private void merge(Picture left,Picture right){
    int width = left.getWidth();
    int height = left.getHeight();

    double scaleL = 0;
    double scaleR = 0;
    int redL = 0;
    int greenL = 0;
    int blueL = 0;
    int redR = 0;
    int greenR = 0;
    int blueR = 0;
    Pixel pixelL = null;
    Pixel pixelR = null;

    for(int row = 0;row < height;row++){
      for(int col = 0;col < width;col++){
        scaleR = (double)col/width;
        scaleR *= scaleR;
        scaleL = 1.0 - scaleR;
        pixelL = left.getPixel(col,row);
        pixelR = right.getPixel(col,row);

        redL = pixelL.getColor().getRed();
        greenL = pixelL.getColor().getGreen();
        blueL = pixelL.getColor().getBlue();

        redR = pixelR.getColor().getRed();
```

Image 1. One of two input images.



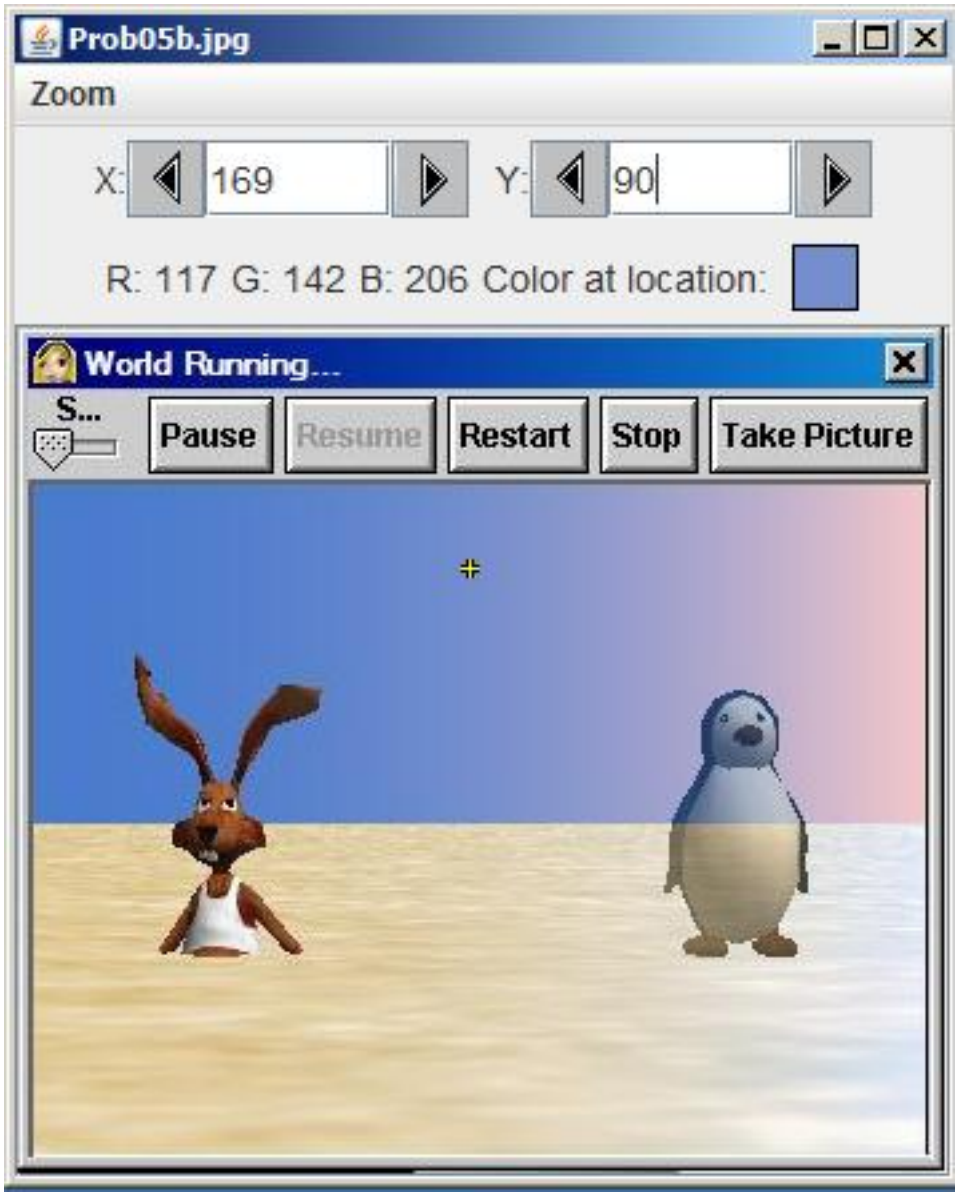
5.440

Image 2. The second of two input images.



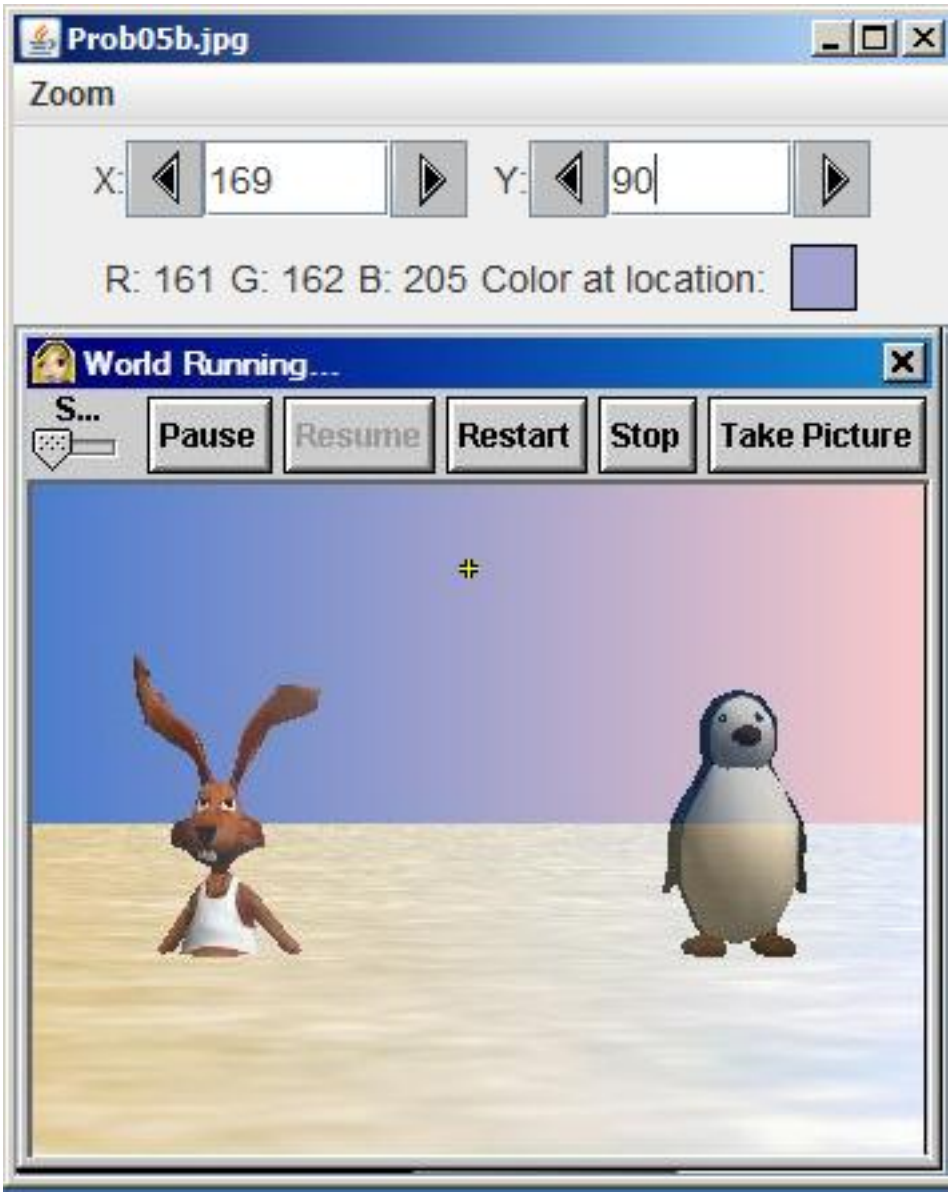
5.441

Image 3. Possible output image.



5.442

Image 4. Possible output image.



5.443

Answer 1 (p. 1186)

5.3.46.4 Images

- Image 1 (p. 1181) . One of two input images.
- Image 2 (p. 1182) . The second of two input images.

- Image 3 (p. 1183) . Possible output image.
- Image 4 (p. 1184) . Possible output image.

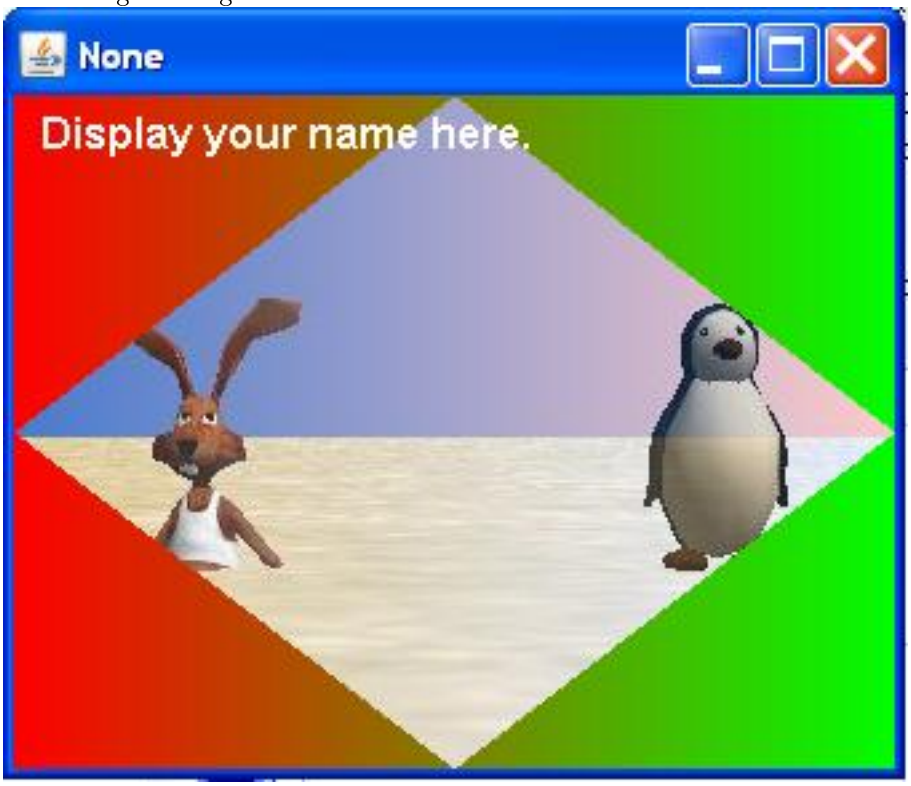
5.3.46.5 Listings

- Listing 1 (p. 1180) . Question 1.

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.

The image is also an example of the kinds of things that we do in my course titled ITSE 2321, Object-Oriented Programming.



This image was also inserted for the purpose of inserting space between the questions and the answers.

5.3.46.6 Answers

5.3.46.6.1 Answer 1

The code in Listing 1 (p. 1180) produces the output image shown in Image 3 (p. 1183) .
Back to Question 1 (p. 1179)

5.3.46.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3030r Review
- File: Java3030r.htm
- Published: 02/18/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.3.47 Java3030s Slides[242]

5.3.47.1 Table of Contents

5.3.47.2 Instructions for viewing slides

This module contains lecture slides keyed to the module titled Java3030: Merging Pictures [243] .

Click here [244] to open an index to the slides.

Then use the links beginning with the label "aa" to open the first slide that you want to view in a new window in your browser.

Then use the [Next] and [Prev] links on the individual slides to navigate back and forth through the slides.

You can also use the links on the index page to jump to a particular slide to avoid having to cycle through the slides in sequence.

5.3.47.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java3030s Slides
- File: Java3030s.htm
- Published: 01/06/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

[242] This content is available online at <http://cnx.org/content/m45648/1.2/>.
[243] http://cnx.org/content/m44247
[244] http://cnx.org/content/m45648/latest/a0-Index.htm

5.4 The Java Collections Framework

5.4.1 Java4010: Getting Started with Java Collections[245]

5.4.1.1 Table of Contents

[245] This content is available online at <http://cnx.org/content/m46135/1.3/>.

- Complete program listing (p. 1197)

5.4.1.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming in general and the Java Collections framework in particular.

The purpose of this module is to introduce you to the Java Collections Framework. Once you learn how to use the framework, it is unlikely that you will need to reinvent common data structures, search algorithms, or sorting algorithms again, because those capabilities are neatly packaged within the framework.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [246] in Oracle's Java Tutorials [247] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.1.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.1.2.1.1 Listings

- Listing 1 (p. 1191) . A new object of the TreeSet class.
- Listing 2 (p. 1192) . Populate the collection.
- Listing 3 (p. 1192) . The Populator class.
- Listing 4 (p. 1194) . Get an Iterator object.
- Listing 5 (p. 1194) . Traverse the collection.
- Listing 6 (p. 1198) . Complete program listing

5.4.1.3 Preview

This module provides a brief introduction to the use of the *Java Collections Framework* . The framework is designed to encourage you to reuse rather than to reinvent collections and maps.

A collection represents a group of objects, known as its elements. Some collections allow duplicate elements while others do not. Some collections are ordered and others are not. *(Maps will be discussed in future modules.)*

The *Collections Framework* is defined by a set of interfaces and associated contracts, and provides concrete implementations of the interfaces for the most common data structures. In addition, the framework also provides several abstract implementations, which are designed to make it easier for you to create new and different implementations while still maintaining the structural polymorphic integrity of the framework.

5.4.1.4 Generics

The code in this series of modules is written with no thought given to Generics [248] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

[246] http://docs.oracle.com/javase/tutorial/collections/index.html
[247] http://docs.oracle.com/javase/tutorial/index.html
[248] http://docs.oracle.com/javase/tutorial/java/generics/index.html

5.4.1.5 Introduction

5.4.1.5.1 A quiz

Let's begin with a little quiz to establish your baseline knowledge of the Collections Framework. Take a look at the program in Listing 6 (p. 1198) near the end of this module. Which of the following is the output produced by that program?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 12344
- E. 1234
- F. None of the above.

If your answer was **1234** *(and it wasn't a guess)* then you may already know quite a lot about the use of the Collections Framework. If not, keep reading to begin learning about the framework.

5.4.1.5.2 Elements of the Framework are easy to use

This simple introductory program is not intended to do anything useful. Instead, it was designed to illustrate several important features of the framework, including the ease with which elements of the framework can be reused in your programs.

5.4.1.5.3 Don't reinvent the wheel

As many of you already know, I am a college professor. I specialize in teaching OOP using Java. In the past, many college courses in Data Structures *(often referred to as CS2 courses)* have emphasized the concept of *reinventing the wheel* . Students were required to learn how to reinvent a variety of complex data structures in order to successfully complete the course.

Hopefully, with the conversion of these CS2 courses to Java OOP, the emphasis will change to *reuse* instead of *reinvent* .

5.4.1.5.4 Collections Framework encourages reuse

The Java Collections Framework is designed to encourage programmers to reuse existing interfaces and classes instead of inventing new ones. In the event that it is necessary to invent a new class or interface, the programmer is encouraged to integrate it into the framework in a polymorphic manner.

5.4.1.6 Sample program

I am going to provide a brief discussion of the sample program *(shown in Listing 6 (p. 1198))* in this module. Later, I will provide more detailed discussions of many of the features used in that program.

5.4.1.7 Interesting code fragments

I will break this program down and discuss it in fragments.

5.4.1.7.1 An object of the TreeSet class

The code fragment in Listing 1 (p. 1191) instantiates an object of the **TreeSet** class and stores the object's reference in a reference variable of type **Collection** named **ref** .

Listing 1. A new object of the TreeSet class.

```
  class Worker{
public void doIt(){
  Collection ref = new TreeSet();
```

Figure 5.444: Listing 1. A new object of the TreeSet class.

5.4.1.7.2 Collection is an interface

The **TreeSet** class implements the **SortedSet** interface, which extends the **Set** interface, which in turn extends the **Collection** interface. Thus, a **TreeSet** object *is a* **Collection** . Therefore, a reference to a **TreeSet** object can be stored in a reference variable of type **Collection** , and can be treated as the generic type **Collection** .

5.4.1.7.3 What is a TreeSet object?

Among other things, in CS2 courses, we worry about the time and memory cost of a collection. According to Sun, the **TreeSet** class guarantees that the sorted set will be in ascending element order, and provides guaranteed log(n) time cost for the basic operations (*add* , *remove* and *contains*).

5.4.1.7.3.1 What does ascending element order mean?

Again, according to Sun, the elements will be sorted according to the *natural order* of the elements *(see the Comparable interface)* or by a comparator *(see the Comparator interface)* provided at the time the set is created. This depends on which overloaded constructor is used. I will have more to say about these alternatives in a subsequent module.

5.4.1.7.3.2 What does log(n) time cost mean?

I'm not going to try to explain the details of log(n) time cost here. Suffice it to say that the *add* , *remove* , and *contains* methods execute very fast. *(I will have more to say about this is a subsequent module.)*

5.4.1.7.3.3 A TreeSet object is a Set

An object of the **TreeSet** class also *is a* **Set** . One of the characteristics of a Java **Set** *(an object that implements the Set interface)* is that it can contain no duplicate elements. Therefore, a **TreeSet** object can contain no duplicate elements. If the **add** method of a **TreeSet** object is called in an attempt to add a duplicate element, the element will not be added.

5.4.1.7.3.4 A TreeSet object is a SortedSet

The **TreeSet** class also implements the **SortedSet** interface. This guarantees that the contents of a **TreeSet** object will be in ascending element order, regardless of the order in which the elements are added. *(In a subsequent module, I will discuss how comparisons are made to enforce the ordering of the elements.)*

5.4.1.7.3.5 A TreeSet object is a Collection

Because an object of the **TreeSet** class *is a* **Collection** , a reference to such an object can be passed to any method that requires an incoming parameter of type **Collection** . The receiving method can call any method on that reference that is declared in the **Collection** interface **.** *(I will discuss such methods in detail in subsequent modules.)*

5.4.1.7.4 Populate the Collection

The statement in Listing 2 (p. 1192) passes the **TreeSet** object's reference to a method named **fillIt** , which is a static method of the **Populator** class. *(The Populator class is a class of my own design whose only purpose is to illustrate the polymorphic behavior achieved using the Collections Framework.)* The behavior of this method is to add elements to the incoming **Collection** object without regard for the actual type of the object *(the class from which the object was instantiated).*

Listing 2. Populate the collection.

```
Populator.fillIt(ref);
```

Figure 5.445: Listing 2. Populate the collection.

At this point, I am going to discuss the **fillIt** method of the **Populator** class called in Listing 2 (p. 1192) . The entire class definition of the **Populator** class, including the **fillit** method, is shown in Listing 3 (p. 1192) .

Listing 3. The Populator class.

```
   class Populator{
  public static void fillIt(Collection ref){
    ref.add(new Integer(4));
    ref.add(new Integer(4));
    ref.add(new Integer(3));
    ref.add(new Integer(2));
    ref.add(new Integer(1));
  }//end fillIt()
}//end class populator
```

Figure 5.446: Listing 3. The Populator class.

5.4.1.7.4.1 Don't know, don't care

As you can see in the above fragment, the **fillIt** method receives the reference to the **TreeSet** object as type **Collection** . This method doesn't know, and doesn't care, what the actual type of the object is. All

it cares about is that the object *is a* **Collection** object. *(Otherwise, the object's reference couldn't be passed in as a parameter. A type mismatch would occur.)*

Because the incoming parameter is a reference to a **Collection** object, the **fillIt** method can call the **add** method on the object with confidence that the behavior of the **add** method will be appropriate for the specific type of object involved. *(For example, the behavior of the add method for an object of the TreeSet class will probably be different from the behavior of the add method for an object of some other class that implements the Collection interface.)*

5.4.1.7.4.2 Polymorphism in action

The great thing about polymorphic behavior is that the author of the **fillIt** method doesn't need to be concerned about the implementation details of the **add** method.

5.4.1.7.4.3 Add five elements with some duplicates

The code in the **fillIt** method adds five elements to the object. Each element is a reference to a new object of type **Integer** . Two of the objects encapsulate the **int** value 4, and thus are duplicates.

The **int** values encapsulated in the **Integer** objects are not in ascending order. Rather, they are added to the object in descending order. *(They could be added in any order and the end result would be the same.)*

5.4.1.7.4.4 Filter out the duplicates

The **add** method for the **TreeSet** object filters out the duplicate element in order to satisfy the contract of the **Collection** interface.

5.4.1.7.4.5 Notification of duplicates

In this case, the author didn't care what happens in the case of duplicate elements. If the author of the **fillIt** method does care what happens in the case of duplicates, she can find out when an object is a duplicate.

According to the contract of the **Collection** interface, the **add** method must return *true* if the call to the method modifies the contents of the object and must return *false* if the collection does not permit duplicates and the collection already contains the specified element.

5.4.1.7.4.6 Sort the elements

Even though the elements are passed to the **add** method in descending order *(or could be passed in any other order),* they are stored and maintained in the **TreeSet** object in such a way that they can later be accessed in ascending order.

5.4.1.7.4.7 The TreeSet object is now populated

When the **fillIt** method returns, the **TreeSet** object contains four *(not five)* elements with no duplicates. Each element is a reference to an object of type **Integer** . Those references are maintained in such a way as to make them accessible in ascending order, based on the **int** values encapsulated in each of the **Integer** objects.

5.4.1.7.5 Get an Iterator object

Returning now to the **doIt** method in the **Worker** class that was called in Listing 1 (p. 1191) , the statement in Listing 4 (p. 1194) calls the **iterator** method on the **TreeSet** object's reference that is stored in the reference variable of type **Collection** .

Listing 4. Get an Iterator object.

```
Iterator iter = ref.iterator();
```

Figure 5.447: Listing 4. Get an Iterator object.

The call to the **iterator** method on any **Collection** object returns an instance of a class that implements the **Iterator** interface. The **Iterator** object can be used to traverse the collection, gaining access to each element in order. *(The concept of in order means different things for different kinds of collections. For a collection instantiated from the TreeSet class, in order means in ascending order.)*

5.4.1.7.5.1 Again, don't know, don't care

Again, the author of the method that uses the **Collection** object doesn't need to know or care about the internal implementation of the collection, or the implementation of the methods of the **Iterator** object. They simply do what they do, and can be used for their intended purpose.

5.4.1.7.5.2 An Iterator object acts as a doorkeeper

The **Iterator** interface declares three methods:

- hasNext()
- next()
- remove()

You might say that an **Iterator** object acts as a doorkeeper for the collection object that it represents, providing access to the contents of the collection in a very specific manner.

5.4.1.7.5.3 Traverse the collection

The code fragment in Listing 5 (p. 1194) below shows how the first two of the above methods can be used to

- Traverse the collection, accessing each of the object's elements in succession.
- Display the value encapsulated in the object referred to by each element.

As mentioned earlier, when the collection is an object instantiated from the **TreeSet** class, access to the elements is provided in ascending order.

Listing 5. Traverse the collection.

```
   while(iter.hasNext()){
  System.out.print(iter.next());
}//end while loop
```

Figure 5.448: Listing 5. Traverse the collection.

5.4.1.7.5.4 Four elements with no duplicates

At this point, the **TreeSet** object contains four elements, with no duplicates. Each of the elements is a reference to an object of type **Integer** . The code in the loop in Listing 5 (p. 1194) causes each of those elements to be accessed and displayed in ascending order. This causes the following text to appear on the screen:

1234

5.4.1.7.6 An editorial opinion

In my opinion, this is the kind of knowledge that a computer science student in a modern data structures course should be learning. This is a far departure from courses of the past where CS2 students were required to memorize the intricate details of how to implement various data structures.

5.4.1.7.6.1 What kind of knowledge is needed?

Does an architect need to understand the detailed inner workings of an air conditioning compressor in order to design a cooling system into a building? Of course not!

However, the architect does need to know the tradeoffs among the available cooling systems in terms of initial cost, operating cost, size, efficiency, etc.

Does an audio technician need to understand the detailed inner workings of an electronic audio equalizer in order to construct an integrated audio system? Absolutely not! If that were a requirement, there would likely be very few audio systems in existence.

However, the audio technician does need to understand the tradeoffs among the various available audio equalizers.

5.4.1.7.6.2 The same concept applies to software design

Does an OOP software designer need to know the detailed inner workings of the various kinds of collection objects in order to use them effectively? No!

However, the software designer does need to know the tradeoffs among the various types of collection objects in terms of their operational behavior.

Modern CS2 students should be learning about the performance and operational differences among the different types of collections, and how to use available frameworks to create and use those collections. They should not be wasting their time learning how to reinvent them. They have more important ways to spend their time, and they have more important things to learn.

5.4.1.7.6.3 An analogy

Frankly, I don't care how the programmers at Sun implemented the **TreeSet** class, so long as the behavior of objects instantiated from that class meets the published specifications.

As an analogy, I also don't care how they implemented the **Random** class, so long as objects instantiated from the **Random** class provide the pseudo random values that I need in my programs.

I see no conceptual differences between the **TreeSet** class and the **Random** class from a software reuse viewpoint.

- I can instantiate an object of the **Random** class to produce pseudo random values, without caring how those values are actually generated. However, if I am working in cryptography, I might need to know how many such values can be generated before the sequence repeats.
- I can use any of the thirty or so methods of the **Math** class to produce a variety of complex mathematical values without caring about how those values are actually produced. However, since many of those values are approximations, I might need to know something about the quality of the approximation.

- I can instantiate an object of the **TreeSet** class to create a collection object, which guarantees that the sorted set will be in ascending element order, and provide log(n) time cost for the basic operations of *add* , *remove* , and *contains* . As long as I know that, I have very little need to know exactly how the collection object is implemented.

5.4.1.7.6.4 Its time to reinvent the CS2 curriculum

I'm confident that the future employers of most students share my opinion on this. I don't know of any employer who wants their programmers to spend time and dollars reinventing the classical data structures. What those employers are looking for is a staff of programmers who understand the tradeoffs among the data structures, and when it is appropriate to use each of the different structures.

It is time to reinvent the curriculum in CS2 courses by

- Encouraging the understanding of techniques for software reuse.
- Teaching when, why, and how each of the different structures should be used.
- Discouraging the reinvention of those structures.

5.4.1.8 Run the program

I encourage you to copy the code from Listing 6 (p. 1198) and paste it into your text editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.1.9 Summary

In this module, I have provided a brief introduction to the use of the *Java Collections Framework* . The framework is designed to encourage you to reuse rather than to reinvent collections and maps *(I will have more to say about maps in a future module).*

A collection represents a group of objects, known as its elements.

While some collections allow duplicate elements, others do not. Some collections are ordered and others are not ordered.

The Collections Framework is defined by a set of interfaces and associated contracts. The framework provides concrete implementations of the interfaces *(classes)* for the most common data structures. In addition, the framework also provides several abstract implementations, which are designed to make it easier for you to create new and different concrete implementations.

The **TreeSet** class is a concrete implementation of the **SortedSet** interface. The **SortedSet** interface extends **Set** , which extends **Collection** . Thus, a **TreeSet** object *is a* **SortedSet** . Also it *is a* **Set** , and it *is a* **Collection** .

The **TreeSet** class guarantees that the sorted set will be in ascending element order, and provides guaranteed log(n) time cost for the basic operations (*add* , *remove* and *contains*).

TreeSet objects can be treated as the generic type **Collection** . Methods declared in the **Collection** interface can be called on a **TreeSet** object without regard for the actual class from which the object was instantiated. *(This is polymorphic behavior.)*

When such methods are called, the author of the program can have confidence that the behavior of the method will be appropriate for an object of the class from which the object was instantiated. In my opinion, this is the true essence of object-oriented behavior.

5.4.1.10 What's next ?

This is the first module in a miniseries on the Collection Framework. Subsequent modules will teach you how to use the framework for creating and using various types of collections and maps.

Once you learn how to use the framework, it is unlikely that you will need to reinvent classical data structures, search algorithms, or sorting algorithms, because those capabilities are neatly packaged within the framework.

5.4.1.11 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java0410: Getting Started with Java Collections
- File: Java0410.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.4.1.12 Complete program listing

A complete listing of the program is provided in Listing 6 (p. 1198) below.

Listing 6. Complete program listing.

```
    import java.util.TreeSet;
import java.util.Collection;
import java.util.Iterator;

public class AP400{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class AP400

class Worker{
  public void doIt(){
    Collection ref = new TreeSet();
    Populator.fillIt(ref);
    Iterator iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(
                       Collection ref){
    ref.add(new Integer(4));
    ref.add(new Integer(4));
    ref.add(new Integer(3));
    ref.add(new Integer(2));
    ref.add(new Integer(1));
  }//end fillIt()
}//end class populator
```

Figure 5.449: Listing 6. Complete program listing.

-end-

5.4.2 Java4020: What is a Collection[249]

5.4.2.1 Table of Contents

[249] This content is available online at <http://cnx.org/content/m46136/1.2/>.

5.4.2.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

The purpose of this module is to explain some of the details surrounding the use of a Java collection for creating data structures. The module also discusses the interfaces and some of the concrete implementations in the Java Collections Framework.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [250] in Oracle's Java Tutorials [251] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.2.3 Preview

Collection is not only the name of a Java interface, it is also the term given to an object that groups multiple elements into a single unit.

I will discuss the advantages of passing collections between methods as type **Collection** .

I will summarize the core interfaces in the Collections Framework and show you how they are related.

I will very briefly discuss some of the concrete implementations of the interfaces that are provided by the framework.

And finally, I will introduce you to the three kinds of things that are part of a collections framework.

5.4.2.4 Generics

The code in this series of modules is written with no thought given to Generics [252] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

[250] http://docs.oracle.com/javase/tutorial/collections/index.html
[251] http://docs.oracle.com/javase/tutorial/index.html
[252] http://docs.oracle.com/javase/tutorial/java/generics/index.html

5.4.2.5 Discussion

5.4.2.5.1 What is a collection?

Just to see if you are awake today, let's start with a little quiz.

What is a collection insofar as Java programming is concerned?

- A. Something they gather in plates at church.
- B. An object that groups multiple elements into a single unit.
- C. The name of a Java interface.
- D. None of the above.

If you answered A, you are probably reading an article on the wrong website by mistake. If you answered *Both B and C* , then you are off to a good start on this module.

Collection is the name of a Java interface. This interface is an integral part of the *Java Collections Framework* . **Collection** is one of two top-level interfaces in the framework. The other top-level interface in the framework is named **Map** .

According to The Java Tutorial from Oracle, a collection *(sometimes called a container)* is also an object that groups multiple elements into a single unit.

Typical collection objects might contain information about employees in the company telephone book, all the purchase orders issued during the past year, or the transactions occurring in a person's checking account.

5.4.2.5.2 Slightly different terminology

Note that this terminology may be somewhat different from what you are accustomed to. For example, if you speak of your *coin collection,* you are probably speaking about the actual coins rather than the container that holds the coins.

This is an important distinction. The usage of the term *collection* in the Collection Framework usually refers to the container and not to the contents of the container. In the framework, the contents are usually referred to as *elements* .

5.4.2.5.3 Store references rather than objects

The collections in the framework always store references to objects, rather than storing the objects themselves. One consequence of this is that primitive values cannot be stored in a collection without first encapsulating them in an object. *(Standard wrapper classes are provided for encapsulating all primitive types.)*

5.4.2.5.4 Stored as type Object

Furthermore, the references are always stored as type **Object** . Prior to Java version 1.5, when you retrieved an element from a collection, you frequently needed to downcast it before you could gain access to the members of the object to which the reference refers. Version 1.5 introduced Generics [253] into the Java programming environment, which eliminated that requirement, *(provided that you use the more complex syntax required by Generics)* .

5.4.2.5.5 Moving data among methods

In addition to their use for storing, retrieving, and manipulating data, collections are also used to move data among methods.

One of the primary advantages of the Collections Framework is the ability to pass a collection to a method as the generic interface type **Collection** . The receiving method doesn't need to know the actual type of the object referred to by the incoming reference in order to call its methods.

[253] http://docs.oracle.com/javase/tutorial/java/generics/index.html

5.4.2.5.6 Polymorphic behavior

The receiving method can call *(on the reference to the Collection object)* any of the methods declared in the **Collection** interface, with confidence that the behavior of the method will be appropriate for the actual type of **Collection** object involved. *(That is polymorphic behavior.)*

5.4.2.5.7 Core collection interfaces

If you have been working with the framework, you might be inclined to think that all of the interfaces in the following list are members of the *core collection interfaces.*

- Collection
- Set
- List
- Queue
- Deque
- SortedSet
- Map
- SortedMap
- Iterator

However, that is not the case.

While the **Iterator** interface is heavily used in conjunction with collections, according to The Java Tutorial from Oracle, it is not one of the *core collection interfaces.*

The core collection interfaces identified by the Oracle book are shown below, with indentation showing the superinterface-subinterface relationships among the interfaces.

- **Collection**
  - Set
    - SortedSet
  - List
  - Queue
  - Deque
- **Map**
  - SortedMap

As you can see, as mentioned earlier, **Collection** and **Map** are the two top-level interfaces.

You should probably commit the above list of interfaces and their relationships to memory. You might find that helpful when navigating the Oracle documentation.

5.4.2.5.8 Concrete implementations

In addition to interfaces, the framework provides several concrete implementations of the interfaces defined in the framework. *(A concrete implementation is a class that implements one or more of the interfaces.)*

Are you still awake? If so, see if you can answer the following question.

True or False? Each of the following classes provides an implementation of one of the interfaces that make up the Java Collections Framework. If False, which items don't belong in the list.

- AbstractSet
- AbstractList
- AbstractMap
- HashSet

- TreeSet
- LinkedList
- Vector
- ArrayList
- HashMap
- Hashtable
- WeakHashMap
- TreeMap
- Iterator
- Attributes
- RenderingHints

Hopefully your answer was False, but even so, that isn't the complete answer.

5.4.2.5.9 Iterator is not a class

To begin with, **Iterator** is not a class. I told you that a couple of paragraphs back. It is an interface. Therefore, it has no place in the above list of classes.

5.4.2.5.10 What about Attributes and RenderingHints?

You may also have wondered if the classes named **Attributes** and **RenderingHints** belong on the list. Note that I didn't restrict the above list to only those classes that might be considered part of the framework, so this was sort of a trick question. *(Of course you could have looked them up in the Oracle documentation just like I did.)*

While these two classes are not really a part of the core Java Collections Framework, they do implement interfaces that are part of the framework.

The **RenderingHints** class implements the **Map** interface, and is used in conjunction with the **Graphics2D** class. The **Attributes** class also implements the **Map** interface,

5.4.2.5.11 What is a Collections Framework?

According to The Java Tutorial from Oracle, *"A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain three things."*

Those three things are:

- Interfaces
- Implementations
- Algorithms

This is probably a good place to close off the discussion for this module. The next module will take up at this point, providing a more in-depth discussion of the interfaces, implementations, and algorithms that make up the framework.

5.4.2.6 Summary

I started out by telling you that a collection is not only the name of a Java interface *(Collection)* but is also an object that groups multiple elements into a single unit.

Java **Collection** objects don't store objects or primitive values directly. Rather, they store references to objects. Further, all such references are stored as the type **Object** . However, the use of the Generics syntax can eliminate the need to downcast the reference in order to gain access to the members of the object to which it refers. *(Generics also provide other useful properties as well.)*

If you need to store primitive values in a collection, you will first need to wrap those values in appropriate objects. Standard wrapper classes are provided for all the primitive types.

Collections are not only useful for storing and retrieving data, they are also useful for moving data among methods.

Because a collection can be passed to a method as type **Collection** , all of the methods declared in the **Collection** interface can be called on the incoming reference in a polymorphic manner.

In addition to the interfaces defined in the Collections Framework, the framework also provides various concrete implementations of the interfaces for many of the commonly-used data structures. This makes it possible for you to conveniently use the framework without the requirement to define new **Collection** classes.

There are eight core interfaces in the Collections Framework. Although the **Iterator** interface is often used with collections, it is not one of the core interfaces.

I ended the module by telling you that there are basically three things in a collections framework: interfaces, implementations, and algorithms.

5.4.2.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4020: What is a Collection
- File: Java4020.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.3 Java4030: Purpose of Framework Interfaces[254]

5.4.3.1 Table of Contents

[254] This content is available online at <http://cnx.org/content/m46140/1.2/>.

5.4.3.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

There are eight core interfaces in the *Collections Framework* . Each interface declares several methods and provides a contract that applies to each declared method. The method declarations and their associated contracts specify the general behavior of matching methods in the classes that implement the interfaces.

The purpose of this module is to provide a brief explanation of those interfaces.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [255] in Oracle's Java Tutorials [256] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.3.3 Preview

At least three things are included in a collections framework:

- interfaces
- implementations
- algorithms

This module will discuss the purpose of the interfaces in the Collections Framework. Future modules will discuss implementations and algorithms.

5.4.3.4 Introduction

In an earlier module, we learned that the Collections Framework contains eight core interfaces with the parent-child relationships **shown below** :

- **Collection**
  - Set
    - SortedSet
  - List
  - Queue
  - Deque
- **Map**
  - SortedMap

[255] http://docs.oracle.com/javase/tutorial/collections/index.html
[256] http://docs.oracle.com/javase/tutorial/index.html

5.4.3.5 Generics

The code in this series of modules is written with no thought given to Generics [257] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.3.6 Discussion

5.4.3.6.1 Purpose of framework interfaces

A *collection* is an object of some type, and that type is defined in a generic sense by one or more interfaces that make up the Collections Framework.

5.4.3.6.2 What is a data type?

All data types specify the operations that can be performed on an entity of that type. *(Data types also specify the kinds of data that can be stored in an entity of that type, but that is not germane to this discussion.)*

5.4.3.6.3 Interface is a type

An object in Java can often be considered to be of several different types. One of those types is determined by any interfaces implemented by the class from which the object was instantiated. Framework collection objects in Java are instantiated from classes that implement the core interfaces of the Collections Framework.

Thus, a Java interface in the Collections Framework specifies the type of such an object, and provides a generic representation of the operations that apply across different implementations of the interface.

5.4.3.6.4 Collection interface declares several methods

The **Collection** interface declares several methods. This is not unusual. From a technical standpoint, all interfaces declare none, one, or more methods. Most interfaces declare multiple methods. *(Interfaces can also declare constants, but that is not germane to this discussion.)*

In general, there is no technical requirement for a specification of the behavior of the interface methods when implemented in a class. In fact, because a method that is declared in an interface is abstract, it specifically refrains from defining the behavior of the method. The interface definition simply declares the interfaces for all the methods that it declares.

We have now arrived at one of the differences that distinguish the Collections Framework from *"just a bunch of interfaces."* That difference is *contracts* .

5.4.3.6.5 An extra step

The Oracle documentation for the **Collection** interface goes a step beyond the minimum technical requirements for an interface. The documentation describes the general behavior that *must be exhibited* by each of the methods belonging to an object instantiated from a class that implements the **Collection** interface. This is sometimes referred to as a *contract* .

Therefore, if you define a class that implements the **Collection** interface in a manner consistent with the *Collections Framework* , it is important that you make certain that each of your methods behaves as described in the Oracle documentation. In other words, you must be careful to comply with the contract defined for those methods. If you don't do that, a user can't rely on objects instantiated from your class to exhibit proper behavior.

[257] http://docs.oracle.com/javase/tutorial/java/generics/index.html

5.4.3.6.6 The add method in Collection

For example, the **Collection** interface declares a method named **add** that receives an incoming reference of a generic type *(see Generics [258])* and returns a **boolean** .. Here is some text from the Oracle documentation describing the required behavior *(contract)* of the **add** method for any class that implements the **Collection** interface.

"Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add null elements, and others will impose restrictions on the type of elements that may be added.

Collection classes should clearly specify in their documentation any restrictions on what elements may be added. If a collection refuses to add a particular element for any reason other than that it already contains the element, it must throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns."

As you can see, the behavior is defined in a very general way. There is no indication as to how that behavior is to be achieved.

5.4.3.6.7 The add method in Set

As you can see from the list above (p. 1204) , the **Set** interface extends the **Collection** interface. In keeping with the general form of object-oriented design, **Set** is more specialized than **Collection** . Therefore, **Set** makes the contract for the **add** method more specific for objects of type **Set** . Here is some text from the Oracle documentation describing the contract of the **add** method for any class that implements the **Set** interface.

"Adds the specified element to this set if it is not already present (optional operation). More formally, adds the specified element e to this set if the set contains no element e2 such that (e==null ? e2==null : e.equals(e2)).

If this set already contains the element, the call leaves the set unchanged and returns false. In combination with the restriction on constructors, this ensures that sets never contain duplicate elements.

The stipulation above does not imply that sets must accept all elements; sets may refuse to add any particular element, including null, and throw an exception, as described in the specification for Collection.add. Individual set implementations should clearly document any restrictions on the elements that they may contain."

5.4.3.6.8 How do the contracts differ?

The contract for the **add** method, as declared in the **Collection** interface, does not prohibit duplicate elements, but does make the provision for interfaces that extend **Collection** to prohibit duplicate elements.

The contract for the **add** method in the **Set** interface does prohibit duplicate elements.

5.4.3.6.9 What about the List interface?

Here is some text from the Oracle documentation describing the contract of the **add** method for any class that implements the **List** interface.

"Appends the specified element to the end of this list (optional operation).

Lists that support this operation may place limitations on what elements may be added to this list. In particular, some lists will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. List classes should clearly specify in their documentation any restrictions on what elements may be added."

[258] http://docs.oracle.com/javase/tutorial/java/generics/index.html

As you can see, the contract for the **add** method declared in the **List** interface, *(which extends Collection)* , does not prohibit duplicate elements. However, it does have some other requirements that don't apply to **Set** objects. For example, it must add new methods at the end of the list.

5.4.3.6.10 A major difference

This is one of the major differences between lists and sets in the Java Collection Framework. Both **List** objects and **Set** objects are collections, because both of the interfaces extend the **Collection** interface. However, the **Set** interface contract prohibits duplicate elements while the **List** interface contract does not prohibit duplicate elements.

5.4.3.6.11 Designing a framework

In theory, it should be possible *(but perhaps not very practical)* to define a framework consisting solely of interface definitions and associated contracts for methods and algorithms. Then each user could implement the interfaces however they see fit, provided that they comply with the contracts. *(This might not be very practical, however, because every user of the framework would then be required to implement the interfaces, which would entail a lot of work.)*

5.4.3.6.12 Concrete implementations

Fortunately, Oracle didn't stop work after defining the interfaces and contracts for the Java Collections Framework. Rather, they also provided us with several useful classes that implement the interfaces in the framework. Thus, we can instantiate and use objects of those classes immediately without having to define them ourselves. Here is a list of some of the concrete implementation classes in the Java Collections Framework:

- HashSet
- TreeSet
- LinkedList
- Vector
- ArrayList
- HashMap
- Hashtable
- WeakHashMap
- TreeMap

In addition, Oracle provided us with several partial implementation classes including **AbstractSet** , **AbstractList** , and **AbstractMap** , which are intended to serve a starting point for new implementations that we choose to define. According to Oracle, these classes provide *a skeletal implementation of the Set, List, and Map interfaces to minimize the effort required to implement those interfaces.*

5.4.3.7 Summary

There are eight core interfaces in the Collections Framework.

As is always the case, each of the core interfaces defines a data type. Each interface declares several methods. In addition, each interface provides a contract that applies to each declared method. The contracts become more specific as we traverse down the interface inheritance hierarchy.

Objects instantiated from classes that implement the interfaces can be considered to be of the interface type or any ancestor interface in the interface's hierarchical family tree.

The method declarations and their associated contracts in the interfaces specify the general behavior of matching methods in the classes that implement the interfaces.

The framework provides several concrete implementations of the interfaces that we can use to instantiate new objects to use as data structures or data containers.

The framework also provides several abstract implementations that we can use as a starting point for defining our own implementations.

5.4.3.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4030: Purpose of Framework Interfaces
- File: Java4030.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.4 Java4040: Purpose of Framework Implementations and Algorithms[259]

5.4.4.1 Table of Contents

[259] This content is available online at <http://cnx.org/content/m46137/1.3/>.

5.4.4.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module explains how the core collection interfaces in the Java Collections Framework allow collections to be manipulated without regard for how they are implemented. The framework provides nine or more concrete implementations of the interfaces. The framework also provides various algorithms for manipulating the data in the collections.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [260] in Oracle's Java Tutorials [261] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.4.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.4.2.1.1 Listings

[260] http://docs.oracle.com/javase/tutorial/collections/index.html
[261] http://docs.oracle.com/javase/tutorial/index.html

5.4.4.3 Preview

At least three things are included in the Java *Collections Framework:*

- interfaces
- implementations
- algorithms

The previous module discussed the purpose of the interfaces. This module will discuss the purpose of the implementations and the algorithms in the Collections Framework.

5.4.4.4 Generics

The code in this series of modules is written with no thought given to Generics [262] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.4.5 Introduction

We learned in an earlier module that the framework provides at least nine concrete implementations of the interfaces in the framework. These nine implementation classes are available for immediate instantiation to produce objects to satisfy your collection needs.

We also learned that the framework provides at least three incomplete implementations. These classes are available for you to use as a starting point in defining your own implementations. Default implementations of many of the interface methods are provided in the incomplete implementations.

5.4.4.6 Discussion and sample code

5.4.4.6.1 Purpose of implementations

The *implementations* in the Java Collections Framework are the concrete definitions of the classes that implement the *core collection interfaces* . For example, concrete implementations in the Java Collections Framework are provided by at least the following nine classes.

- HashSet
- TreeSet
- LinkedList
- ArrayList
- Vector
- HashMap
- WeakHashMap
- TreeMap
- Hashtable

5.4.4.6.1.1 Available for immediate use

These classes are available for immediate use to instantiate collection objects.

As you can see, there are two classes that obviously fall into the *Set* category, two that obviously fall into the *List* category, and three that obviously fall into the *Map* category. You can learn more about the detailed characteristics of those classes in the standard Java documentation and in The Java Tutorials [263] .

This leaves two additional classes whose names don't readily divulge the category to which they belong.

[262] http://docs.oracle.com/javase/tutorial/java/generics/index.html
[263] http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

5.4.4.6.1.2 Vector and Hashtable classes

The classes **Vector** and **Hashtable** were part of Java even before the Java Collections Framework became available. The **Vector** class can be used to instantiate objects that fall in the general *List* category.

The **Hashtable** class can be used to instantiate objects that fall in the *Map* category.

These two classes have been upgraded to make them compatible with the Collections Framework.

5.4.4.6.1.3 Abstract implementations

In addition to the concrete implementations listed above, the following three classes partially implement the interfaces, but are not intended for instantiation. Rather, they are intended to be extended into new concrete classes that you define.

- AbstractSet
- AbstractList
- AbstractMap

Therefore, by either using one of the three classes listed above as a starting point, or by starting from scratch and fully implementing one or more of the interfaces, you can provide new concrete implementations to augment the framework to include collections that meet your special needs. If you do that, be sure to satisfy the contract requirements of the Collections Framework in addition to the technical requirements imposed by implementing interfaces.

5.4.4.6.2 Purpose of algorithms

Algorithms are methods *(not necessarily exposed)* that provide useful capabilities, such as searching and sorting. For example, the **Collection** interface declares an exposed method named **contains** .

5.4.4.6.2.1 The contains method

The contract for the **contains** method requires that the method:

- receives an incoming reference of type **Object** as a parameter
- searches the collection looking for an element that matches the incoming reference
- returns true if the collection on which the method is called contains the specified element and returns false otherwise.

5.4.4.6.2.2 Different classes, different implementations

You can safely call the **contains** method on any object instantiated from a class that properly implements the **Collection** interface, even if you don't know the actual type of the collection object.

The manner in which the search will be performed will probably differ from one concrete implementation of the interface to the next. For example, a **TreeSet** object will perform the search very rapidly with a time cost of only log(n) where n is the number of elements. On the other hand, for the same number of elements, because of a different underlying data structure, a search on an **ArrayList** object will probably require more time than a search on a **TreeSet** object. As the number of elements increases, the difference in time cost between the two will also increase.

5.4.4.6.3 A sample program

Consider the sample program shown in Listing 1 (p. 1212) . This program compares the search speed of the **ArrayList** class and the **TreeSet** class. A detailed discussion of the program follows Listing 1 (p. 1212) .

Listing 1. SpeedTest01.

```
    /*File SpeedTest01
Copyright 2001 R.G.Baldwin
************************************/

import java.util.*;

public class SpeedTest01{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class SpeedTest01

class Worker{
  public void doIt(){
    int size = 2000000;
    //Create a TreeSet object
    Collection aTree = new TreeSet();

    //Populate the TreeSet object with
    // random values.  The add() method
    // for a set rejects duplicates.
    Random rnGen = new Random();
    for(int ct = 0; ct < size; ct++){
      aTree.add(new Double(rnGen.nextDouble()));
    }//end for loop

    //Create and populate an ArrayList
    // object with the same random
    // values
    Collection aList = new ArrayList(aTree);

    //Extract a value near the center
    // of the ArrayList object to use
    // as a test case.
    Object testVal = ((List)aList).get(size/2);

    //Search for the test value in each
    // of the collection objects.
    // Measure and display the time
    // required to perform the search
    // in each case.
    long start = new Date().getTime();
    boolean found = aList.contains(testVal);
    long stop = new Date().getTime();
    System.out.println(found + " " + (stop - start));

    start = new Date().getTime();
    for(int x = 0; x < 100000; x++){
      found = aTree.contains(testVal);
    }//end for loop
    stop = new Date().getTime();
    System.out.println(found + " " + (stop - start)/100000.0);

  }//end doIt()
}// end class Worker
```

5.4.4.6.3.1 Instantiate and populate a TreeSet object

The program begins by instantiating a **TreeSet** object and populating it with approximately 2,000,000 elements as shown in Listing 2 (p. 1213) . The values encapsulated in the objects referred to by the elements in the collection are produced by a random number generator.

Recall that the **add** method of a **Set** object rejects duplicate elements, so there may be fewer than 2,000,000 elements in the object after it is populated, depending on how many of the random values are duplicates.

Listing 2. Beginning of the doIt method.

```
  public void doIt(){
int size = 2000000;

Collection aTree = new TreeSet();

Random rnGen = new Random();
for(int ct = 0; ct < size; ct++){
  aTree.add(new Double(rnGen.nextDouble()));
}//end for loop
```

Figure 5.451: Listing 2. Beginning of the doIt method.

5.4.4.6.3.2 Instantiate and populate an ArrayList object

One of the capabilities of the Collection Framework is to create a new **Collection** object and populate it with the contents of an existing **Collection** object of a different *(or the same)* actual type.

The code in Listing 3 (p. 1213) instantiates an **ArrayList** object and populates it with the contents of the existing **TreeSet** object. As a result, we then have two different **Collection** objects of different actual types containing the same elements.

Listing 3. Instantiate and populate an ArrayList object.

```
Collection aList = new ArrayList(aTree);
```

Figure 5.452: Listing 3. Instantiate and populate an ArrayList object.

5.4.4.6.3.3 Identify a target element

The objective of this program is to compare the times required to search for and to find an element in each of the collections. Thus, we need a target element to search for.

The code in Listing 4 (p. 1214) extracts a value near the center of the **ArrayList** object using an index to find and extract the value. This is a very fast operation on a **List** object. This value is saved in **testVal** to be used later for test purposes.

Note that the reference to the **ArrayList** object was saved as type **Collection** *(and not as type **ArrayList**)* in Listing 3 (p. 1213) above.

Note also that it was necessary to cast that reference to type **List** in Listing 4 (p. 1214) in order to call the **get** method on the reference. This is because the **Collection** interface does not declare a method named **get** . Rather, the **get** method is added to the **List** interface to define a more specialized form of collection.

> *(Author's note: This program was originally written before the introduction of Generics. The above requirement may not be true if the program were to be rewritten making proper use of Generics.)*

Listing 4. Identify a target element.

```
Object testVal = ((List)aList).get(size/2);
```

Figure 5.453: Listing 4. Identify a target element.

5.4.4.6.3.4 Search for the test value in each collection

The code in Listing 5 (p. 1215) calls the **contains** method to search for the test value in each of the collections. It uses the system clock to measure the time required to find the element in each case. *(I will assume that you understand how to use the **Date** class for this purpose, and won't provide a detailed explanation.)*

Listing 5. Search for the test value in each collection.

```
    long start = new Date().getTime();
  boolean found = aList.contains(testVal);
  long stop = new Date().getTime();
  System.out.println(found + " " + (stop - start));

  start = new Date().getTime();
  for(int x = 0; x < 100000; x++){
    found = aTree.contains(testVal);
  }//end for loop
  stop = new Date().getTime();
  System.out.println(found + " " + (stop - start)/100000.0);

}//end doIt()
```

Figure 5.454: Listing 5. Search for the test value in each collection.

5.4.4.6.3.5 Program output

Running the program several times produced the following range of output values:

- First output value ranged from "true 93" to "true 109"
- Second output value ranged from "true 0.00031" to "true 0.00046"

The first output value applies to the **ArrayList** object, and the second output value applies to the **TreeSet** object.

As we would expect, the test value was successfully found in both cases; hence the display of true in both cases.

5.4.4.6.3.6 Time required to search the ArrayList collection

The output indicates that approximately 100 milliseconds were required to find the test value in the **ArrayList** object.

5.4.4.6.3.7 Time required to search the TreeSet collection

The time required to find the test value in the **TreeSet** object was so small that it wasn't even measurable within the granularity of the system clock *(other experiments have caused me to believe that the granularity of the system clock on this machine is at least sixteen milliseconds)* . Hence, the original reported time required to find the test value in the **TreeSet** object was zero.

In order to get a measurable time value to search the **TreeSet** object, I had to wrap the invocation of the **contains** method in a for-loop and search for the same value 100,000 times in succession. Thus, the time required to find the test value in the **TreeSet** object was approximately 0.00030 milliseconds as compared to 100 milliseconds for the **ArrayList** object.

> *(I'll let you do the arithmetic to see if this makes sense in terms of the expected time cost to search the two different types of collections. Don't forget the extra overhead of the for-loop.)*

5.4.4.6.3.8 Different implementations

This is a graphic demonstration that even though both objects can be treated as type **Collection**, and the **contains** method can be called on either object in a polymorphic manner, the actual implementations of the two objects and the implementations of the **contains** methods in those two objects are different.

Each type of collection has advantages and disadvantages, depending on your needs.

5.4.4.6.3.9 Polymorphic behavior applies

The important point is that if you receive a reference to the collection object as type **Collection**, you can call the **contains** method on that reference without regard to the underlying structure of the collection object. This is because *polymorphic* behavior applies.

Very briefly, polymorphic behavior means that the actual method that is executed is the appropriate method for that type of object regardless of the actual type *(class)* of the reference to the object. This is one of the great advantages of using the Java Collections Framework and passing collection objects among methods as interface types.

5.4.4.6.4 Sorting algorithms

Some of the implementations of the Java Collection Framework maintain their elements in a random order, and other implementations maintain their elements in a sorted order. Thus, the framework also provides sorting algorithms. However, the sorting algorithms used to maintain the order of the collections are not exposed in the way that the search algorithm is exposed *(via the **contains** method)*. Rather, the sorting algorithms are implicit in those implementations that need them, and are absent from those implementations that don't need them.

5.4.4.6.5 Now for a little quiz

Let's see if you are still awake. Select the words in one pair of parentheses in the following statement that causes the statement to be true.

> *The interfaces in the Collections Framework make it possible to manipulate the contents of collections in a manner that is (dependent on) (independent of) the underlying implementation of each collection.*

5.4.4.6.5.1 And the answer is ...

The interfaces in the Collections Framework make it possible to manipulate the contents of collections in a manner that is *i **ndependent of*** the underlying implementation of each collection. That is the beauty of basing the framework on interfaces that declare polymorphic methods.

5.4.4.6.5.2 Drive home the point

I placed this question here to drive home the point that the methods declared in the **Collection** interface can be called on collection objects in a *polymorphic* manner.

That is to say, as a user of an object instantiated from a class that properly implements the **Collection** interface *(according to the contracts of the Collections Framework)*, you can call the methods declared in that interface on a reference to the object and be confident that the actual method that is called will be the version that is appropriate for the class from which the object was instantiated. This is polymorphic behavior.

In the event that you need to call a method that is not declared in the **Collection** interface *(such as the get() method in Listing 4 (p. 1214) above)*, you can pass the reference as one of the more specialized sub-interfaces of **Collection**, such as **Set**.

(Author's note: Once again, this document was originally written before the release of Generics. The use of the more specialized sub-interfaces described above may not be required if the program is written making proper use of Generics.)

5.4.4.6.6 Benefits of using the Collections Framework

The Java Tutorial [264] from Oracle lists and explains the benefits of using the Java Collections Framework, including the following.

- It reduces programming effort
- It increases program speed and quality
- It allows interoperability among unrelated APIs
- It reduces the effort to learn and use new APIs
- It reduces effort to design new APIs
- It fosters software reuse

For a detailed explanation of these benefits, I am simply going to refer you directly to The Java Tutorial [265] .

5.4.4.7 Run the program

I encourage you to copy the code from Listing 1 (p. 1212) and paste it into a Java source code file. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.4.8 Summary

Let's recap some of what we have learned in this and the previous modules.

The core collection interfaces in the Collections Framework are shown below.

- Collection
  - Set
    - SortedSet
  - List
  - Queue
  - Deque
- Map
  - SortedMap

The basic purpose of the core collection interfaces in the Java Collections Framework is to allow collections to be manipulated without regard for how the collections are implemented, provided of course that the implementations comply with the contracts.

The framework provides at least the following nine concrete implementations *(classes)* of the interfaces shown above:

- HashSet
- TreeSet

[264] http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html
[265] http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

- LinkedList
- ArrayList
- Vector
- HashMap
- WeakHashMap
- TreeMap
- Hashtable

For example, the classes **TreeSet** and **ArrayList** are concrete implementations of the **Collection** interface as shown in the above list.

> *(Actually, they are concrete implementations of sub-interfaces of Collection. The Collections Framework doesn't provide any direct implementations of the Collection interface.)*

A collection object instantiated from the class **TreeSet** and a collection object instantiated from the class **ArrayList** can each be viewed as being of the interface type **Collection** .

Methods having the same signatures can be used to manipulate either collection with confidence that the behavior of the method will be appropriate for the actual type of collection involved.

The framework also provides the following incomplete implementations of the core interfaces:

- AbstractSet
- AbstractList
- AbstractMap

The purpose of these implementations is to provide you with a starting point for defining your own concrete implementations for more specialized collections.

5.4.4.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java0440: Purpose of Framework Implementations and Algorithms
- File: Java0440.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.5 Java4050: Core Collection Interfaces[266]

5.4.5.1 Table of Contents

5.4.5.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

The Java *Collections Framework* defines eight core interfaces, in two distinct trees. You will learn about the inheritance structure and the purpose of those interfaces. You will also learn how the interfaces declare polymorphic methods that apply to implementations of the interfaces, and you will learn about the optional methods of the **Collection** and **Map** interfaces.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [267] in Oracle's Java Tutorials [268] . The modules in this collection are intended to supplement and not to replace those tutorials.

[266] This content is available online at <http://cnx.org/content/m46138/1.2/>.
[267] http://docs.oracle.com/javase/tutorial/collections/index.html
[268] http://docs.oracle.com/javase/tutorial/index.html

5.4.5.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.5.2.1.1 Listings

- Listing 1 (p. 1221) . The program named Ap401.
- Listing 2 (p. 1222) . The Populator class.
- Listing 3 (p. 1223) . Create and populate a TreeSet object.
- Listing 4 (p. 1224) . Create and populate an ArrayList object.

5.4.5.3 Preview

In earlier modules, you learned that at least three things are included in a collections framework:

- interfaces
- implementations
- algorithms

Earlier modules provided a general discussion of the purpose of the interfaces, implementations, and algorithms in the *Collections Framework* . This module takes that discussion further and illustrates the use of the *core collection interfaces.*

The Java Collections Framework defines eight core interfaces, in two distinct trees. You will learn the names and the inheritance structure of those interfaces. You will also learn about the purpose of some of those interfaces. You will see how the interfaces declare polymorphic methods that apply to implementations of the interfaces, and you will learn about the optional methods of the **Collection** interface.

5.4.5.4 Generics

The code in this series of modules is written with no thought given to Generics [269] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.5.5 Discussion and sample code

5.4.5.5.1 Illustration of core collection interfaces

We will begin this module with a little quiz. Take a look at the program shown in Listing 1 (p. 1221) and see if you can answer the following question.

What output does the program in Listing 1 (p. 1221) produce?

- A. Compiler Error
- B. Runtime Error
- C. 44321 44321
- D. 12344 12344
- E. 1234 44321
- F. 1234 4321
- D. None of the above.

[269] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Ap401.

```
    import java.util.TreeSet;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Ap401{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap401

class Worker{
  public void doIt(){
    Collection ref = new TreeSet();
    Populator.fillIt(ref);
    Iterator iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.print(" ");

    ref = new ArrayList();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();
  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(new Integer(4));
    ref.add(new Integer(4));
    ref.add(new Integer(3));
    ref.add(new Integer(2));
    ref.add(new Integer(1));
  }//end fillIt()
}//end class populator
```

Figure 5.455: Listing 1. The program named Ap401.

If you selected the following answer, then you are correct.

E. 1234 44321

The program in Listing 1 (p. 1221) illustrates the basic purpose of the core collection interfaces in the Java Collections Framework. That purpose is to allow collections to be manipulated without regard for how the collections are implemented.

5.4.5.5.1.1 Multiple list implementations

For example, there is more than one way to implement a list. Two common ways involve arrays and linked structures. If two lists are implemented in different ways, but both satisfy the requirements of the core collection interfaces, they can each be manipulated the same way regardless of the details of their implementation.

5.4.5.5.1.2 TreeSet and ArrayList

A collection of type **TreeSet** and a collection of type **ArrayList** are instantiated in the program in Listing 1 (p. 1221) . Each of the collections is viewed as being of the interface type **Collection** . A method named **add** is used to populate each collection with the same values in the same order.

5.4.5.5.1.3 Behavior is different but appropriate

The behavior of the **add** method is appropriate, and different in each of the two cases, with the final contents of each collection being determined by the respective behavior of the **add** method for that type of collection.

5.4.5.5.2 The fillIt method

The code in the fragment shown in Listing 2 (p. 1222) defines a *static* method named **fillIt** of the class named **Populator** . This is a class of my own design intended solely to illustrate the primary point of this program.

The method named **fillIt** receives an incoming reference to a collection object as type **Collection** . The method calls the **add** method on the incoming reference five times in succession to add five elements to the collection. These elements are added without regard for the actual type or underlying implementation of the collection. *(As written, the **fillIt** method has no way of knowing the underlying implementation.)*

Listing 2. The Populator class.

```
    class Populator{
  public static void fillIt(Collection ref){
    ref.add(new Integer(4));
    ref.add(new Integer(4));
    ref.add(new Integer(3));
    ref.add(new Integer(2));
    ref.add(new Integer(1));
  }//end fillIt()
}//end class populator
```

Figure 5.456: Listing 2. The Populator class.

The **fillIt** method will be used to populate two collections of different types with the same data values in the same order.

5.4.5.5.3 Create and populate a TreeSet object

Consider the code fragment shown in Listing 3 (p. 1223) .

Listing 3. Create and populate a TreeSet object.

```
    Collection ref = new TreeSet();
Populator.fillIt(ref);
Iterator iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next());
}//end while loop
```

Figure 5.457: Listing 3. Create and populate a TreeSet object.

The code in Listing 3 (p. 1223) instantiates an object of type **TreeSet** , and passes that object's reference to the **fillIt** method as type **Collection** . As described above, the **fillIt** method adds five elements to the collection, in random order with two of the elements being duplicates.

> *"Note that this program does not use the syntax for Generics. Therefore, if you copy and compile this program, you will probably see a warning regarding unchecked or unsafe operations"*

5.4.5.5.3.1 Display the collection's contents

Then the code in Listing 3 (p. 1223) gets an **Iterator** object on the collection and uses the iterator to display the contents of the collection.

5.4.5.5.3.2 TreeSet object is type SortedSet

The **TreeSet** class implements one of the core collection interfaces named **SortedSet** . **SortedSet** is a sub interface of **Set** . One of the characteristics of a **Set** object is that it doesn't allow duplicate elements. One of the characteristics of a **SortedSet** object is that, by default, it maintains its elements in ascending natural order. Since the **TreeSet** class implements both of these interfaces, it is both a **Set** and a **SortedSet** , and exhibits the characteristics of both interfaces.

Because the underlying structure of the **TreeSet** class doesn't allow duplicates, and the underlying structure maintains its elements in ascending order, the code in Listing 3 (p. 1223) produces the following text on the screen:

1234

5.4.5.5.4 Create and populate an ArrayList object

Now consider the code fragment shown in Listing 4 (p. 1224) .

Listing 4. Create and populate an ArrayList object.

```
   ref = new ArrayList();
Populator.fillIt(ref);
iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next());
}//end while loop
```

Figure 5.458: Listing 4. Create and populate an ArrayList object.

The code in Listing 4 (p. 1224) instantiates a new collection of type **ArrayList** , and passes that object's reference to the same **fillIt** method, once again as type **Collection** .

The code in the **fillIt** method adds five elements having the same values as before to the collection and adds them in the same order as before. The added elements are references to **Integer** objects encapsulating the same values as were earlier added to the **TreeSet** collection. Although they are physically different objects, the result is that essentially the same data is added to both collections.

5.4.5.5.4.1 Display the collection's contents

Then, as before, the code in Listing 4 (p. 1224) gets an iterator and uses it to access and display the contents of the **ArrayList** collection.

The **ArrayList** class implements the **List** interface, which does not prohibit duplicate elements, and does not maintain its elements in sorted order. Therefore, in this case, the following text was displayed:

44321

All five element values are displayed, including the duplicate, in the order in which they were added to the list.

5.4.5.5.5 The important point

The important point is that although the **fillIt** method calls the same method name (**add**) on each of the collection objects, the behavior of that method is different in each case. In both cases, the behavior is appropriate for the underlying data structure. Furthermore, the underlying data structure isn't even known to the **fillIt** method.

5.4.5.5.5.1 No duplicate elements in ascending order

In the first case, where the underlying data structure was a **TreeSet** object *(type **SortedSet**)*, the duplicate element was eliminated, and the elements were stored so as to be accessible in ascending order.

5.4.5.5.5.2 Duplicates allowed with no sorting

In the second case, where the underlying data structure was an **ArrayList** object *(type **List**)*, all five elements, including the duplicate element were stored in the collection. Furthermore, they were stored and later retrieved in the same order in which they were added.

5.4.5.5.6 Structure of the core interfaces

The *core collection interfaces* in the Java Collections Framework do not all extend from a common root interface.

Rather, the inheritance structure of the core interfaces is shown below. Indentation is used to indicate the parent-child relationships among the interfaces.

- Collection
  - Set
    - SortedSet
  - List
  - Queue
  - Deque
- Map
  - SortedMap

5.4.5.5.7 A Map is not a true Collection

As you can see, that there is no common root interface. Rather, there are two distinct trees, one rooted by **Collection** and the other rooted by **Map** . According to The Java Tutorial from Oracle, *"a Map is not a true Collection."* I will have more to say about this in a future module.

5.4.5.5.8 Some operations are optional

Every class that implements an interface in the tree rooted in **Collection** is not required to support all of the methods *(operations)* declared in the **Collection** interface.

Rather, some of the methods in the **Collection** interface are designated as "optional operation" in the documentation. *(See the list of optional methods for the Collection interface below.)*

According to the contract for the Collections Framework, if a given implementation doesn't support a specific method, it must throw an **UnsupportedOperationException** . The author of the implementation is responsible for providing documentation that identifies the optional operations that the implementation does and does not support.

5.4.5.5.8.1 Support for optional operations

This should not be an issue unless you are either defining your own implementation, or using an implementation defined by someone other than the programmers at Oracle. All of the general-purpose implementations from Oracle appear to support all of the optional operations.

5.4.5.5.8.2 Optional Collection operations

The following list shows the optional operations in the **Collection** interface. Each of these methods has the ability to modify the contents of the collection.

- add()
- addAll()
- clear()
- remove()
- removeAll()
- retainAll()

5.4.5.5.8.3 Optional Map operations

The following list shows the optional operations in the **Map** interface. Each of these methods also has the ability to modify the contents of the map.

- clear()
- put()
- putAll()
- remove()

5.4.5.5.8.4 Many methods are not optional

In both cases, the interface declares numerous other methods that are not optional. Generally, the non-optional methods don't have the ability to modify the collection. For example, the **get** method of the **Map** interface is not optional. Although the **get** method receives an incoming *key* and returns the *value* to which the key maps, the method doesn't have the ability to modify the contents of the collection.

5.4.5.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1221) and paste it into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.5.7 Summary

A collections framework contains at least the following items:

- interfaces
- implementations
- algorithms

The *Java Collections Framework* defines eight core interfaces, in two distinct trees. One tree is rooted in **Collection** and the other is rooted in **Map** .

The basic purpose of the core interfaces is to make it possible for collections to be manipulated without regard for how they are implemented, so long as the implementation satisfies the contracts of the interfaces.

When the same method name *(and signature)* is called on references to collections of different types, the behavior of the method is likely to be different for each collection. However, in each case, that behavior will be appropriate for the type of collection object on which the method is called. This is polymorphic behavior.

Six of the methods declared in the **Collection** interface are optional insofar as being supported by implementing classes is concerned. The optional methods all have the ability to modify the contents of the collection. Those implementing classes that don't support an optional method must throw an **UnsupportedOperationException** if that method is called on an object of the class. Similarly four of the methods declared in the **Map** interface are optional.

Many methods declared in the **Collection** interface are not optional. Generally, the non-optional methods don't have the ability to modify the collection.

5.4.5.8 What's next?

In the next module, I will discuss and illustrate some of the details of the core interfaces and the general-purpose implementations in the Java Collections Framework. For example, I will discuss the difference between a *set* and a *list* . I will also discuss the difference between *ordered* and *sorted* . I will discuss the fact that additional stipulations are applied as you progress down the framework interface hierarchy. In order to help you learn and retain the material, I will provide a couple of short quizzes.

5.4.5.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4050: Core Collection Interfaces
- File: Java4050.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.6 Java4060: Duplicate Elements, Ordered Collections, Sorted Collections, and Interface Specialization[270]

5.4.6.1 Table of Contents

[270] This content is available online at <http://cnx.org/content/m46141/1.2/>.

5.4.6.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [271] in Oracle's Java Tutorials [272] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.6.3 Preview

You learned in earlier modules that the Java Collections Framework defines eight core interfaces, in two distinct trees. One of the trees, which consists of six interfaces, is rooted in the interface named **Collection** . The other tree, which consists of two interfaces, is rooted in the interface named **Map** .

You learned the names and the inheritance structure of those interfaces. You also learned about their purpose. You saw how the interfaces declare polymorphic methods that apply to implementations of the interfaces, and you learned about the optional methods of the **Collection** interface and the **Map** interface.

In this module you will learn that all of the implementations of the interfaces on the **Collection** side of the Java Collections Framework *(the Collection hierarchy)* implement one of the sub-interfaces of the **Collection** interface. *(A similar discussion regarding the* ***Map*** *side of the Java Collections framework will be deferred until a future module.)*

You will learn that a **Set** object cannot contain duplicate elements, but a **List** object can contain duplicate elements.

You will learn about the difference between *ordered* collections and *sorted* collections. You will also learn about *ascending order* and the *natural ordering* of objects.

In addition, you will learn how more specialized stipulations are placed on interfaces as you progress down the interface inheritance hierarchy of the Java Collections Framework.

5.4.6.4 Generics

The code in this series of modules is written with no thought given to Generics [273] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

[271] http://docs.oracle.com/javase/tutorial/collections/index.html
[272] http://docs.oracle.com/javase/tutorial/index.html
[273] http://docs.oracle.com/javase/tutorial/java/generics/index.html

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.6.5 Discussion

5.4.6.5.1 We will start with a quiz

I am going to begin this module with a quiz just to make sure that you are still awake. Is the following statement True or False?

> *The* ***TreeSet*** *class is a direct implementation of the* ***Collection*** *interface.*

The answer is False.

The **TreeSet** class is not a direct implementation of the **Collection** interface. Rather, the **TreeSet** class is a direct implementation of the **SortedSet** interface. The **SortedSet** interface extends the **Set** interface, and the **Set** interface extends the **Collection** interface.

The interface hierarchy for the Java Collections Framework is shown below:

- Collection
  - Set
    - SortedSet
  - List
  - Queue
  - Deque
- Map
  - SortedMap

As you learned in an earlier module, the Java Collections Framework doesn't have a single root. As shown above, there are two distinct trees in the framework – The **Collection** hierarchy and the **Map** hierarchy.

5.4.6.5.2 The root of the Collection hierarchy

The **Collection** interface is the root of the collection hierarchy. The Java Collections Framework doesn't provide any direct implementations of the **Collection** interface. All of the implementations of the interfaces in the **Collection** hierarchy implement one of the sub-interfaces of the **Collection** interface.

5.4.6.5.2.1 What does Oracle say about this?

Here is what the Oracle documentation has to say on the topic of the **Collection** interface:

"The SDK does not provide any direct implementations of this interface: it provides implementations of more specific sub-interfaces like ***Set*** *and* ***List*** *. This interface is typically used to pass collections around and manipulate them where maximum generality is desired."*

The Oracle documentation also states:

"Bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly."

5.4.6.5.3 What about duplicate elements?

Some implementations of **Collection** allow duplicate elements, and others do not. Implementations of the **List** interface *(such as* ***ArrayList*** *)* allow duplicate elements. Implements of **Set** and **SortedSet** *(such as* ***TreeSet*** *)* do not allow duplicate elements. This was illustrated in an earlier module.

A sample program in that earlier module created two collection objects and applied the polymorphic **add** method to add the same elements to each collection. One of the collection objects was of type **ArrayList** , and the other collection object was of type **TreeSet** . The elements added to each collection contained one pair of duplicate elements. The duplicate element was automatically excluded from the **TreeSet** object, but was retained in the **ArrayList** object.

5.4.6.5.4 What is a set?

According to Oracle, a **Set** is a *"collection that contains no duplicate elements ... this interface models the mathematical set abstraction."*

An object of type **Set** is typically used to model collections such as Social Security numbers, where duplicates are not allowed.

5.4.6.5.5 What is a list?

Also according to Oracle, a **List** is *"An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list."*

5.4.6.5.6 Ordered is not the same as sorted

Note that an *ordered* collection is not the same as a *sorted* collection.

The fact that the collection is ordered derives from the fact that each element in the collection has a specific position specified by an index.

In a sorted collection, the position of each element is determined by its value relative to the values of its predecessors and successors.

Oracle goes on to say, *"Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all."*

5.4.6.5.7 Is ascending sort order always required?

Not all implementations of the **Collection** interface maintain the elements in ascending sort order. Some may, and others do not. For example, as discussed above, implementations of the **List** interface *(such as **ArrayList**)* do not maintain their elements in sorted order at all. In other words, the position of an element in an **ArrayList** does not depend on the value of the element.

On the other hand, implementations of the interface named **SortedSet** *(such as **TreeSet**)* and the interface named **SortedMap** do maintain their elements in sorted order. However, that order is not necessarily ascending.

When an object is instantiated from a class that implements the **SortedSet** interface, the sorting order for that object can be established by providing an object instantiated from a class that implements the **Comparator** interface. In that case, the author of the class that implements the **Comparator** interface determines the order imposed on the elements in the collection.

5.4.6.5.7.1 Does case matter in String objects?

For example, if your **SortedSet** object contains references to **String** objects, the natural ascending sort would take the difference between upper case and lower case characters into account.

However, you might prefer that case be ignored when establishing the sorted order. You can accomplish this by providing an object of a class that implements the **Comparator** interface and which defines the **compare method** and the **equals** method in such a way as to eliminate case considerations for comparisons of **String** objects.

5.4.6.5.8 Sub-interfaces have more stipulations

As you progress down the inheritance hierarchy, you find that additional stipulations apply at each level of inheritance. As an example, according to Oracle, *"The Set interface places additional stipulations, beyond those inherited from the Collection interface, on the contracts of all constructors and on the contracts of the* ***add*** *,* ***equals*** *and* ***hashCode*** *methods."*

The important point is that specific sub-interfaces of the **Collection** interface can define requirements that do not apply to all sub-interfaces of the **Collection** interface.

5.4.6.5.8.1 Stipulations on set

For example, the **add** method of the **Set** interface stipulates the following:

> *"Adds the specified element to this set if it is not already present."*

On the other hand, the **add** method of the **Collection** interface simply states:

> *"Ensures that this collection contains the specified element."*

Thus, the contract for the **add** method of an object of a class that implements the **Set** interface is more specialized than the contract for the **add** method of an object of a class that implements the **Collection** interface.

An additional stipulation on the constructor for a **Set** object is that all constructors must create a set that contains no duplicate elements.

5.4.6.5.8.2 Stipulations on SortedSet

The **SortedSet** interface extends the **Set** interface. The **SortedSet** interface contains the following stipulation that makes it more specialized than a **Set** .

"A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see ***Comparable*** *), or by a* ***Comparator*** *provided at sorted set creation time."*

5.4.6.5.9 We will end with a quiz

I'm going to finish this module with several questions in the form of a quiz. To ensure that this is a learning experience, I will provide an explanation in addition to the answer for each question.

5.4.6.5.9.1 Question 1

True or False? A collection that implements the **List** interface maintains its elements in ascending alphanumeric order.

The answer to question 1 is False. Unlike collections that implement the **SortedSet** interface, the order of the elements in a collection that implements the **List** interface is not based on the values of the objects referred to by the elements in the list.

5.4.6.5.9.2 Question 2

True or False? A collection that implements the **List** interface is an unordered collection.

The answer to question 2 is also False. A collection that implements the **List** interface is an ordered collection *(also known as a sequence)* . According to Oracle, *"The user of the interface has precise control over where in the list each element is inserted."* Elements can be inserted and retrieved on the basis of their integer index *(position in the list)* using the following methods:

- **add(int index, Object element)**

- **get(int index)**

Valid index values are positive integers that begin with zero. When the **add** method is used to insert an element at a specific position in the sequence, the element currently at that position *(if any)* and any subsequent elements are shifted toward higher index values to make room for the new element.

Another version of the **add** method takes a reference to an object as an incoming parameter and appends the specified element to the end of the collection.

The **get** method simply returns the element at the specified position in the collection.

The **List** interface also declares various other methods that can be used to manipulate the contents of the collection.

5.4.6.5.9.3 Question 3

True or False? A collection that implements the **List** interface is allowed to contain duplicate values.

The answer to question 3 is True. Unlike a collection that implements the **Set** interface, a collection that implements the **List** interface is typically allowed to contain duplicate values. More formally, according to Oracle, *"lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all."*

5.4.6.5.9.4 Question 4

True or False? The contracts of the methods in the **List** interface are the same as the contracts of the methods inherited from the **Collection** interface.

The answer to question 4 is False. According to Oracle, *"The **List** interface places additional stipulations, beyond those specified in the **Collection** interface, on the contracts of the **iterator** , **add** , **remove** , **equals** , and **hashCode** methods."*

For example, the **iterator** method *(for both the **List** and **Collection** interfaces)* returns an iterator over the elements in the collection. For the **Collection** interface, there are no guarantees concerning the order in which the elements are returned by the methods of the **Iterator** object.

On the other hand, the **iterator** method for the **List** interface returns an iterator over the elements in the collection in proper sequence, where the sequence is determined by the numeric index. In other words, when you call the methods of the **Iterator** object on a **List** , the elements will be returned in the proper sequence as determined by a numeric index.

Similarly, according to Oracle, the **SortedSet** interface *"guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see **Comparable**), or by a **Comparator** provided at sorted set creation time."*

5.4.6.6 Summary

In this module you learned that all of the implementations of the interfaces in the **Collection** hierarchy implement one of the sub-interfaces of the **Collection** interface. You learned that a **Set** object cannot contain duplicate elements, but a **List** object can contain duplicate elements.

You learned about the difference between *ordered* collections and *sorted* collections. You also learned about *ascending order* and the *natural ordering* of objects. In addition, you learned how more specialized stipulations are placed on interfaces as you progress down the interface inheritance hierarchy of the Java Collections Framework.

5.4.6.7 What's next?

The **SortedSet** interface *"guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see **Comparable**), or by a **Comparator** provided at sorted set creation time."* In the next module, I will show you how to use the **Comparator** interface to control the sorted order of your collections.

5.4.6.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4060: Duplicate Elements, Ordered Collections, Sorted Collections, and Interface Specialization
- File: Java4060.htm
- Published: 04/18/13
- Revised: 05/07/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.7 Java4070: The Comparable Interface, Part 1[274]

5.4.7.1 Table of Contents

[274] This content is available online at <http://cnx.org/content/m46142/1.2/>.

5.4.7.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java *Collections Framework* in particular.

This is also the first module in a sub-collection on the **Comparable** interface. The purpose of the modules in this sub-collection is to teach you about the interactions between the **Comparable** interface and the Collections Framework, particularly with respect to the **Set** , **SortedSet** , and **SortedMap** interfaces of the Collections Framework.

This module explains the *(lack of)* interaction between the **Comparable** interface and the Java Collections Framework with respect to collections of type **List** .

In addition to studying these modules, I strongly recommend that you study the Collections Trail [275] in Oracle's Java Tutorials [276] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.7.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.7.2.1.1 Listings

[275] http://docs.oracle.com/javase/tutorial/collections/index.html
[276] http://docs.oracle.com/javase/tutorial/index.html

5.4.7.3 Preview

In this module, I will begin discussing the interaction between the **Comparable** interface and the Collections Framework.

5.4.7.3.1 Generics

The code in this module is written with no thought given to Generics [277] . As a result, if you copy and compile this code, you will probably get a warning about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.7.3.2 Specialization

I will provide a concrete example of the specialization that occurs while moving down the interface hierarchy from **Collection** to **List** . I will show an example of using two different overloaded versions of the **add** method to add new elements to an **ArrayList** object. One version is declared in the **Collection** interface and both versions are declared in the **List** interface.

5.4.7.3.3 To cast, or not to cast

I will illustrate the use of a *cast* to change the type of a reference from **Collection** to **List** , in order to call a version of the **add** method that is declared only in the **List** interface.

This version of the program, *(in which the **add** method actually does an insert)* makes it possible for the user to control the location of each individual element added to a **List** . The fact that the location of each element can be controlled in a **List** is what causes a **List** to be an *ordered* collection.

I will illustrate that a cast is not required on a reference being treated as type **Collection** in order to call the version of the **add** method that is declared in the **Collection** interface. This version of the **add** method supports the addition of new elements only at the end of the **List** .

5.4.7.3.4 Comparable interface not required for a List

Finally, I will show that it is not necessary for objects to implement the **Comparable** interface to make them eligible for inclusion in a **List** . I will tell you that it is necessary for objects to implement the **Comparable** interface to make them eligible for inclusion in a **SortedSet** , although I won't demonstrate that in this module.

5.4.7.4 Discussion and sample code

5.4.7.4.1 We will begin with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1236) ?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 4321
- E. 1234
- F. 12344
- G. None of the above.

[277] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparable01.

```
    //File Comparable01.java

import java.util.*;

public class Comparable01{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparable01

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new ArrayList();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();
  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(0,new MyClass(4));
    ref.add(1,new MyClass(4));
    ref.add(2,new MyClass(3));
    ref.add(3,new MyClass(2));
    ref.add(4,new MyClass(1));
  }//end fillIt()
}//end class Populator

class MyClass{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()

}//end MyClass
```


Figure 5.459: Listing 1. The program named Comparable01.

If your answer was **A. Compiler Error** , you were correct.

5.4.7.4.1.1 What caused the compiler error?

The compiler error was caused by the code shown in Listing 2 (p. 1237) .

Listing 2. The code with the problem.

```
 public static void fillIt(Collection ref){
ref.add(0,new MyClass(4));
```

Figure 5.460: Listing 2. The code with the problem.

The problem here is that the method named **fillIt** receives a reference to an object of the **ArrayList** class as the interface type **Collection** , and attempts to call the following overloaded method on that reference:

add(int index, Object element)

However, the **Collection** interface knows nothing about a method having that signature.

5.4.7.4.1.2 Implements Collection and List

The **ArrayList** class implements both the **Collection** interface and the **List** interface. As you may recall from earlier modules in this series, **List** is a sub-interface of **Collection** . The **List** interface declares the following overloaded versions of the **add** method:

- **add(Object o)**
- **add(int index, Object element)**

The second of these two methods, which is called in Listing 2 (p. 1237) , is unknown to the **Collection** interface. The **Collection** interface declares only the first version of the **add** method shown above.

5.4.7.4.1.3 Specialization

This is the result of specialization. A **List** object is a more-specialized collection than a **Collection** object.

Therefore, the version of the **add** method that requires two parameter cannot be called on a reference to an **ArrayList** object when that object is treated as the generic type **Collection** .

5.4.7.4.2 Modified program

Now, take a look at the modified version of the program as shown in Listing 3 (p. 1239) .

What output is produced by the program shown in Listing 3 (p. 1239) ?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 4321
- E. 1234
- F. 12344

- G. 443521
- H. None of the above.

Listing 3. The program named Comparable02.

```
    //File Comparable02.java

import java.util.*;

public class Comparable02{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparable02

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new ArrayList();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();
  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ((List)ref).add(0,new MyClass(4));
    ((List)ref).add(1,new MyClass(4));
    ((List)ref).add(2,new MyClass(3));
    ((List)ref).add(3,new MyClass(2));
    ((List)ref).add(4,new MyClass(1));
    ((List)ref).add(3,new MyClass(5));
  }//end fillIt()
}//end class populator

class MyClass{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()
```



```
}//end MyClass
```

Figure 5.461: Listing 3. The program named Comparable02.

If your answer was **G. 443521** , you are correct.

5.4.7.4.2.1 The corrected code

This version of the program illustrates a mechanism for correcting the problem in the earlier program shown in Listing 1 (p. 1236) . The updated code that corrected the problem is shown in Listing 4 (p. 1240) .

Listing 4. The corrected code.

```
    class Populator{
  public static void fillIt(Collection ref){
    ((List)ref).add(0,new MyClass(4));
    ((List)ref).add(1,new MyClass(4));
    ((List)ref).add(2,new MyClass(3));
    ((List)ref).add(3,new MyClass(2));
    ((List)ref).add(4,new MyClass(1));
    ((List)ref).add(3,new MyClass(5));
  }//end fillIt()
}//end class populator
```

Figure 5.462: Listing 4. The corrected code.

The incoming parameter to the **fillIt** method in Listing 4 (p. 1240) is a reference to an object instantiated from the **ArrayList** class. That reference is passed to the **fillIt** method as type **Collection** , which is legal because the **ArrayList** class implements both the **Collection** interface and the **List** interface.

5.4.7.4.2.2 Casting to type List

The code in Listing 4 (p. 1240) uses a cast to convert the incoming reference from type **Collection** to type **List** . Because the version of the **add** method that is used in Listing 4 (p. 1240) is declared in the **List** interface, and because the **ArrayList** class correctly implements the **List** interface, that version of the **add** method can be called on the reference to the **ArrayList** object when it is treated as the interface type **List** . Hopefully this is review material for you at this point. If not, you may need to go back and study some of my earlier modules.

5.4.7.4.3 The List contract for the add method

Listing 4 (p. 1240) also illustrates part of the contract for this version of the **add** method in the **List** interface. This version of the **add** method makes it possible to specify the position of each element added to the **ArrayList** object.

> *(A List is an ordered collection because the user has control over the location of each element in the collection relative to the other elements in the collection.)*

5.4.7.4.3.1 Controlling the locations of the elements

In Listing 4 (p. 1240) , the elements are added to the **ArrayList** object in increasing element order during the first five invocations of the **add** method. However, the sixth invocation of the **add** method adds a new element at index position 3.

5.4.7.4.3.2 Add method actually does an insert

A portion of the contract for this version of the **add** method in the **List** interface is as follows:

> *"Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices)."*

Thus, the new element is inserted at that position, and the other elements are pushed up, as required, toward higher index values to make room for the new element.

5.4.7.4.4 The Vector class

Here is an interesting side note. The Java **Vector** class has been around longer than the Collections Framework. Somewhere along the way, the **Vector** class was upgraded to cause it to become a concrete implementation of the **Collection** interface and the **List** interface.

As a result of the upgrade, the **Vector** class now provides an implementation of the **add** method described above. Except for the order of the parameters, that **add** method appears to have the same behavior as the older method named:

insertElementAt(Object elem, int index)

You can insert elements into a **Vector** object by calling the **add** method on that object while treating it as type **List** . However, since the older **insertElementAt** method is not declared in the **List** interface, you cannot insert an element into the **Vector** object by calling the **insertElementAt** method while treating it as a **List** . In order to call that method, you must treat it as type **Vector** .

5.4.7.4.5 More on the List contract

Another portion of the contract for a **List** object is that the **iterator** method

> *"Returns an iterator over the elements in this list in proper sequence."*

As a result, the code shown in Listing 5 (p. 1241) , along with the overridden **toString** method of the **MyClass** class causes the program to display the elements in the following order:

443521 .

Listing 5. Display using an iterator.

```
    iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next());
}//end while loop
```

Figure 5.463: Listing 5. Display using an iterator.

5.4.7.4.6 Duplicates are allowed in a List

One final thing that is worthy of note in this program is that a **List** objects allows duplicates. Hence, the populated collection contains references to two separate objects that are equal to one another in the sense that they both contain the same values in their instance variables.

5.4.7.4.7 One more sample program

Let's take a look at one more sample program. What output is produced by the program shown in Listing 6 (p. 1243) ?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 4321
- E. 1234
- F. 12344
- G. None of the above.

Listing 6. The program named Comparable03.

```
    //File Comparable03.java

import java.util.*;

public class Comparable03{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparable03

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new ArrayList();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();
  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class populator

class MyClass{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()

}//end MyClass
```


Figure 5.464: Listing 6. The program named Comparable03.

If you selected **C. 44321** , you are correct.

5.4.7.4.7.1 No need to cast to type List

As shown in Listing 7 (p. 1244) , this program takes a different approach to solving the problem originally exposed in the program shown in Listing 1 (p. 1236) .

Listing 7. No need to cast to type List.

```
    class Populator{
  public static void fillIt(
                       Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class populator
```

Figure 5.465: Listing 7. No need to cast to type List.

This program does not change the type of the incoming reference to the **ArrayList** object in the **fillIt** method. Rather, it continues to treat the incoming reference as type **Collection** , and calls the version of the **add** method that is declared in the **Collection** interface. This avoids the requirement to cast the incoming reference to type **List** .

The contract for this version of the **add** method in the **List** interface is

"Appends the specified element to the end of this list (optional operation)."

As a result, the new elements are added to the collection in increasing index order. Since an iterator on a **List** returns the elements in increasing index order, this program displays the elements in the same order that they are added.

5.4.7.4.8 What happened to the Comparable interface?

By now, you are probably wondering what all of this has to do with the **Comparable** interface, because I haven't mentioned that interface since the introductory comments at the beginning of the module.

5.4.7.4.8.1 Comparable interface is not required for a List

Actually, the purpose of this module is to illustrate the lack of any requirement to make use of the **Comparable** interface with **List** objects. In particular, the purpose is to illustrate that this is one of the features that differentiates between a **List** object and a **Set** or **SortedSet** object.

A **List** can be used as a container for other objects regardless of whether or not those objects implement the **Comparable** interface. However, in the next module, we will see that objects must implement the **Comparable** interface in order to be eligible for inclusion in collections that implement the **SortedSet** interface.

This and the next several modules are intended to provide you with an understanding of the interaction between the **Comparable** interface, the **Comparator** interface, and the Collections Framework.

5.4.7.4.8.2 No requirement to compare

Because a **List** makes no attempt to eliminate duplicate elements, or to sort the elements on the basis of their values, there is no requirement to *compare* objects when placing them in a **List** . Therefore, objects whose references are stored in a **List** are not required to implement the **Comparable** interface *(but they may implement the Comparable interface without causing any harm)* .

5.4.7.4.8.3 Comparison is required for a SortedSet

Because a **SortedSet** does eliminate duplicates and does sort the elements on the basis of their values, there is a requirement to *compare* each new element with the existing elements in a **SortedSet** whenever a new element is added to the collection. Therefore, objects whose references are stored in a **SortedSet** are required to implement the **Comparable** interface.

5.4.7.5 Run the program

I encourage you to copy the code from Listing 1 (p. 1236) , Listing 3 (p. 1239) , and Listing 6 (p. 1243) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.7.6 Summary

In this module, I began discussing the interaction between the **Comparable** interface and the Collections Framework.

I provided a concrete example of the specialization that occurs when moving down the interface hierarchy from **Collection** to **List** . I showed an example of using two different overloaded versions of the **add** method to add new elements to an **ArrayList** object. One version is declared in the **Collection** interface and both versions are declared in the **List** interface.

I illustrated the use of a *cast* to change the type of a reference from **Collection** to **List** , in order to call a version of the **add** method that is declared only in the **List** interface. This version makes it possible for the user to control the location of each individual element added to a **List** .

I illustrated that a cast is not required on a reference being treated as type **Collection** in order to call the version of the **add** method that is declared in the **Collection** interface. This version of the **add** method supports the addition of new elements only at the end of the **List** .

Finally, I explained that it is not necessary for objects to implement the **Comparable** interface to make them eligible for inclusion in a **List** .

Although I didn't demonstrate it, I told you that it is necessary for objects to implement the **Comparable** interface to make them eligible for inclusion in a **SortedSet** .

5.4.7.7 What's next?

The next module will begin exploring the interaction between the **Comparable** interface and the **SortedSet** interface of the *Collections Framework.*

5.4.7.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4070: The Comparable Interface, Part 1
- File: Java4070.htm

- Published: 04/19/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.8 Java4080: The Comparable Interface, Part 2[278]

5.4.8.1 Table of Contents

[278] This content is available online at <http://cnx.org/content/m46143/1.1/>.

5.4.8.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module explains why the elements stored in a **TreeSet** collection must be references to objects instantiated from a class that implements the **Comparable** interface. The module also briefly discusses an alternative approach using the **Comparator** interface.

The module shows you how to implement the **Comparable** interface for a new class definition, explains the "natural ordering of the elements" for a class, and discusses the "consistent with equals" requirement. Finally, the module shows you how to define a new class whose objects are eligible for inclusion in a **TreeSet** collection.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [279] in Oracle's Java Tutorials [280] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.8.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.8.2.1.1 Listings

- Listing 1 (p. 1249) . The program named Comparable04.
- Listing 2 (p. 1250) . The code with the problem.
- Listing 3 (p. 1253) . The program named Comparable05.
- Listing 4 (p. 1254) . Beginning of the class named MyClass.
- Listing 5 (p. 1255) . The compareTo method.
- Listing 6 (p. 1255) . The overridden equals method.

5.4.8.3 Preview

In this module, I will teach you why the elements stored in a **TreeSet** collection must be references to objects instantiated from a class that implements the **Comparable** interface. (*In a subsequent module, I will teach you about an alternative approach that makes use of the* ***Comparator*** *interface.)*

I will provide an example of implementing the **Comparable** interface for a new class definition, and will teach you about the *natural ordering of the elements* for a class.

I will teach you the meaning of the *consistent with equals* requirement and show you how to satisfy that requirement for a new class definition.

Finally, I will show you how to define a new class whose objects are eligible for inclusion in a **TreeSet** collection.

5.4.8.4 Discussion and sample code

5.4.8.4.1 Generics

The code in this module is written with no thought given to Generics [281] . As a result, if you copy and compile this code, you will probably get a warning about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

[279] http://docs.oracle.com/javase/tutorial/collections/index.html
[280] http://docs.oracle.com/javase/tutorial/index.html
[281] http://docs.oracle.com/javase/tutorial/java/generics/index.html

5.4.8.4.2 Begin with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1249) ?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 4321
- E. 1234
- F. 12344
- G. None of the above.

Listing 1. The program named Comparable04.

```
    //File Comparable04.java

import java.util.*;

public class Comparable04{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparable04

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new TreeSet();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class Populator

class MyClass{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()
```



```
}//end MyClass
```

Figure 5.466: Listing 1. The program named Comparable04.

If your answer was **B. Runtime Error** , you were correct.

5.4.8.4.2.1 What caused the runtime error?

The runtime error was caused by the code shown in Listing 2 (p. 1250) .

Listing 2. The code with the problem.

```
   class Populator{
 public static void fillIt(Collection ref){
   ref.add(new MyClass(4));
```

Figure 5.467: Listing 2. The code with the problem.

5.4.8.4.2.2 Why did this code produce a runtime error?

This code produced a runtime error for the following reasons.

The incoming parameter of the **fillIt** method is a reference to an object of type **TreeSet** but it is received as type **Collection** . The **TreeSet** class implements the **Collection** , **Set** , and **SortedSet** interfaces. *(In this module, we will be primarily interested in the **Set** and **SortedSet** interfaces.)*

The contract for the **add** method of the **Set** interface reads partially as follows:

> *"Adds the specified element to this set if it is not already present ... If this set already contains the specified element, the call leaves this set unchanged and returns false. ... this ensures that sets never contain duplicate elements."*

5.4.8.4.2.3 What does this mean?

This means that whenever the **add** method is called on a **Set** object, the **add** method must have a way of determining if the element being added is a duplicate of an element that already exists in the collection. This means that it must be possible for the **add** method to *compare* the new element with all of the existing elements to determine if the new element is a duplicate of any of the existing elements.

5.4.8.4.2.4 The compareTo method

The documentation for the **TreeSet** class states the following:

> *"... the **Set** interface is defined in terms of the equals operation, but a **TreeSet** instance performs all key comparisons using its **compareTo** (or **compare**) method ..."*

What this means is that insofar as the handling of duplicate elements is concerned, *(with the possible exception given below involving a **Comparator**)*, in order for a reference to an object to be included in a **TreeSet** collection, the class from which that object is instantiated must implement the **Comparable** interface.

5.4.8.4.2.5 A possible exception

Note that one of the constructors for the **TreeSet** class makes it possible to instantiate a new object by passing a parameter that is a reference to an object that implements the **Comparator** interface.

The **Comparator** interface declares a method named **compare** , which compares its two arguments for order. The text in the above excerpt from the Oracle documentation suggests that when this parameterized constructor is used, it may not be necessary for the objects included in the **TreeSet** collection to implement the **Comparable** interface.

I won't discuss that possibility in this module, but I will discuss it in a future module that discusses the use of the **Comparator** interface. For purposes of this module, I will concentrate on the use of a **TreeSet** collection that does not receive a reference to a **Comparator** object when it is instantiated.

5.4.8.4.2.6 The SortedSet interface

The **TreeSet** class also implements the **SortedSet** interface. The documentation for the **SortedSet** interface states the following:

> *"A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see* ***Comparable*** *), or by a* ***Comparator*** *provided at sorted set creation time."*

5.4.8.4.2.7 Natural ordering of the elements

The key term to note in the above quotation is the term *natural ordering of its elements* . This takes us back to the **Comparable** interface, for which the documentation states:

> *"This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's* ***compareTo*** *method is referred to as its natural comparison method."*

5.4.8.4.2.8 Conclusion regarding traversal

The conclusion is, in order for the iterator to be able to traverse the set according to the *natural ordering of its elements,* the elements stored in an object that implements the **SortedSet** interface must be instantiated from a class that implements the **Comparable** interface *(unless a* ***Comparator*** *is provided when the* ***SortedSet*** *object is instantiated.)*

5.4.8.4.2.9 The bottom line

The bottom line is, because the class named **MyClass** in Listing 1 (p. 1249) does not implement the **Comparable** interface, objects of that class are not eligible for use with a **TreeSet** collection *(unless a* ***Comparator*** *is provided when the* ***TreeSet*** *object is instantiated).*

A **Comparator** was not provided when the **TreeSet** object was instantiated in Listing 1 (p. 1249) . Therefore, the attempt in Listing 2 (p. 1250) , to add a **MyClass** object to the **TreeSet** collection resulted in a **ClassCastException** being thrown at runtime. The runtime error reads partially as follows:

> *"Exception ... java.lang.ClassCastException: MyClass cannot be cast to java.lang.Comparable"*

5.4.8.4.3 The solution

To solve this problem, we must modify the definition of the class named **MyClass** to make it implement the **Comparable** interface *(assuming that we don't provide a* ***Comparator*** *when the* ***TreeSet*** *object is instantiated).*

This is accomplished in the modified version of the program shown in Listing 3 (p. 1253) .

Listing 3. The program named Comparable05.

```
    //File Comparable05.java
import java.util.*;

public class Comparable05{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparable05

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new TreeSet();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class Populator

class MyClass implements Comparable{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()

  public int compareTo(Object o){
    if(!(o instanceof MyClass))
        throw new ClassCastException();
    if(((MyClass)o).data < data)
      return 1;
```

5.4.8.4.3.1 The corrected code

The important code to note in this modified version of the program is the new definition of the class named **MyClass** . The other code in the program is essentially the same as in the previous version of the program.

The beginning portion of the new definition for **MyClass** is shown in Listing 4 (p. 1254) .

Listing 4. Beginning of the class named MyClass.

```
  class MyClass implements Comparable{
int data;

MyClass(){
  data = 0;
}//end noarg constructor

MyClass(int data){
  this.data = data;
}//end parameterized constructor

public String toString(){
  return "" + data;
}//end overridden toString()
```

Figure 5.469: Listing 4. Beginning of the class named MyClass.

The code shown in Listing 4 (p. 1254) is identical to the code in the previous version with one major exception. This version of the class definition implements the **Comparable** interface. That means that this class must provide a concrete definition for the following method, which is the only method declared in the **Comparable** interface:

public int compareTo(Object o)

5.4.8.4.3.2 The compareTo method

The description of the **compareTo** method in the Oracle documentation begins as follows:

> *"Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object."*

Beyond this, there are a number of additional stipulations that I won't repeat here. You can view them in the Oracle documentation if you are interested in that level of detail.

Listing 5 (p. 1255) shows my implementation of the **compareTo** method. Although this implementation satisfies the general description given above, I haven't taken the time to test it fully to confirm that it meets all of the additional stipulations provided by Oracle.

Listing 5. The compareTo method.

```
  public int compareTo(Object o){
if(!(o instanceof MyClass))
    throw new ClassCastException();
if(((MyClass)o).data < data)
  return 1;
if(((MyClass)o).data > data)
  return -1;
else return 0;
}//end compareTo()
```

Figure 5.470: Listing 5. The compareTo method.

5.4.8.4.3.3 Consistent with equals

The Oracle documentation strongly emphasizes the need to make certain that a class' natural ordering is *consistent with equals*, and provides the rules for meeting that requirement.

Further, the documentation for the **TreeSet** class reads partially as follows:

> *"Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be* ***consistent with equals*** *if it is to correctly implement the Set interface. ..."*

5.4.8.4.3.4 Meeting the consistent with equals requirement

In order to satisfy the rules and to cause the *natural ordering* of the **MyClass** class to be *consistent with equals* , it was necessary to override the **equals** method inherited from the **Object** class. My overridden version of the **equals** method is shown in Listing 6 (p. 1255) .

Listing 6. The overridden equals method.

```
  public boolean equals(Object o){
if(!(o instanceof MyClass))
    return false;
if(((MyClass)o).data == data)
  return true;
else return false;
}//end overridden equals()
}//end MyClass
```

Figure 5.471: Listing 6. The overridden equals method.

As was the case in defining the **compareTo** method, there are also a large number of stipulations involved in properly overriding the **equals** method. I will simply refer you to the Oracle documentation if you are interested in reading about those stipulations.

5.4.8.4.3.5 The program output

Given all of the above, this program compiles and executes correctly, producing the following output.

1234

Note that duplicate elements were eliminated, and the iterator traversed the set in ascending element order, sorted according to the natural ordering of the elements, as required for a **SortedSet** collection.

5.4.8.5 Run the program

I encourage you to copy the code from Listing 1 (p. 1249) and Listing 3 (p. 1253) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.8.6 Summary

I explained why the elements stored in a **TreeSet** collection must be references to objects instantiated from a class that implements the **Comparable** interface. (*In a future module, I will teach you about an alternative approach that makes use of the* ***Comparator*** *interface.)*

I provided an example of implementing the **Comparable** interface for a new class definition, and I taught you about the *natural ordering of the elements* for a class.

I taught you the meaning of the *consistent with equals* requirement and showed you how to satisfy that requirement for a new class definition.

I showed you how to define a new class whose objects are eligible for inclusion in a **TreeSet** collection.

5.4.8.7 What's next?

In the next module, I will discuss the use of the **Comparator** interface in order to achieve a sorting order that is different from the *natural ordering* of the elements in a sorted collection.

5.4.8.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4080: The Comparable Interface, Part 2
- File: Java4080.htm
- Published: 04/19/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.9 Java4090: The Comparator Interface, Part 1[282]

5.4.9.1 Table of Contents

[282] This content is available online at <http://cnx.org/content/m46189/1.1/>.

5.4.9.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module discusses and illustrates the use of the **Comparator** interface. The sorting order established by a **Comparator** may be different or may be the same as the natural order. A **Comparator** can be used to establish a sorting order for objects that don't have a natural ordering. The use of a **Comparator** is an alternative to the implementation of the **Comparable** interface.

This module is also the first module in a series of modules on the **Comparator** interface. The purpose of the modules in this series is to teach you about the interactions between the **Comparator** interface and the Collections Framework , particularly with respect to the **Set** , **SortedSet** , and **SortedMap** interfaces of the Collections Framework. This module discusses **Set** and **SortedSet** . A discussion of **SortedMap** will be deferred to a future module.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [283] in Oracle's Java Tutorials [284] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.9.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.9.2.1.1 Listings

5.4.9.3 Preview

Previous modules have discussed the use of the **Comparable** interface. This module discusses and illustrates the use of the **Comparator** interface.

The **Comparable** interface establishes *natural ordering.* The sorting order established by a **Comparator** may be different or may be the same as the *natural order.*

A **Comparator** can be used to establish a sorting order for objects that don't have a *natural ordering* .

The use of a **Comparator** is an alternative to the implementation of the **Comparable** interface. A **TreeSet** object instantiated with the benefit of a **Comparator** object doesn't require the objects in its collection to implement **Comparable** .

[283] http://docs.oracle.com/javase/tutorial/collections/index.html
[284] http://docs.oracle.com/javase/tutorial/index.html

5.4.9.4 Discussion and sample code

5.4.9.4.1 Generics

The code in this module is written with no thought given to Generics [285] . As a result, if you copy and compile this code, you will probably get a warning about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.9.4.2 The Comparable interface

Previous modules have discussed the use of the **Comparable** interface to establish the *natural ordering* of elements in a sorted set. Although the name of the **Comparable** interface is similar to the name of the **Comparator** interface, they are different interfaces. Don't be confused by the similarity of the names.

5.4.9.4.3 The Comparator interface

This module will begin the discussion of an alternative approach to sorting, using the **Comparator** interface to establish sorting order. The discussion will be continued in future modules.

The sorting order established by a **Comparator** may be different from the *natural ordering* . The **Comparator** interface can also be used to establish sorting order for objects that do not implement the **Comparable** interface and therefore do not have a *natural ordering* .

5.4.9.4.4 Beginning with a quiz

Let's begin with a little quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1260) ?

- A. Compiler Error
- B. Runtime Error
- C. 44321
- D. 4321
- E. 1234
- F. 12344
- G. None of the above.

[285] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparator02.

```
    //File Comparator02.java
//Copyright 2001, R.G.Baldwin
import java.util.*;
import java.io.Serializable;

public class Comparator02{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator02

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new TreeSet(new TheComparator());
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next());
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class Populator

class MyClass{
  int data;

  MyClass(){
    data = 0;
  }//end noarg constructor

  MyClass(int data){
    this.data = data;
  }//end parameterized constructor

  public String toString(){
    return "" + data;
  }//end overridden toString()

}//end MyClass

class TheComparator
```

If your answer was **E. 1234** , then you are correct.

5.4.9.4.5 Eligibility for inclusion in a TreeSet

The **TreeSet** class implements the **SortedSet** interface.

In an earlier module, I told you that in order to be eligible for inclusion in a **TreeSet** collection, an object must be instantiated from a class that implements the **Comparable** interface.

At that time, I also told you that it is possible to instantiate a new **TreeSet** object using a constructor that receives an incoming reference to a **Comparator** object, in which case it is not necessary for the objects in the collection to implement the **Comparable** interface.

5.4.9.4.5.1 Using a Comparator object

The program in Listing 1 (p. 1260) takes this latter approach. The main purpose of this program is to illustrate the use of a **Comparator** object as an alternative to implementation of the **Comparable** interface.

5.4.9.4.5.2 Passing Comparator to TreeSet constructor

The code fragment in Listing 2 (p. 1261) shows the instantiation of a new **TreeSet** object, passing an anonymous object of type **TheComparator** as a parameter to the constructor for **TreeSet** . Shortly, we will see that the class named **TheComparator** implements the **Comparator** interface. Therefore, an object instantiated from that class is a **Comparator** object.

Listing 2. Passing Comparator to TreeSet constructor.

```
    Collection ref;
ref = new TreeSet(new TheComparator());
Populator.fillIt(ref);
```

Figure 5.473: Listing 2. Passing Comparator to TreeSet constructor.

5.4.9.4.5.3 Passing the TreeSet to a Populator method

The code fragment in Listing 2 (p. 1261) also shows the reference to the **TreeSet** object being stored in a reference variable of the interface type **Collection** . The reference to the **TreeSet** object is passed as type **Collection** to a method named **fillIt** .

The purpose of the **fillIt** method is to instantiate some objects of type **MyClass** , and to store those object references in the **TreeSet** collection.

5.4.9.4.5.4 The fillIt method

The code fragment in Listing 3 (p. 1262) shows the entire method named **fillIt** . This method instantiates five objects from the class named **MyClass** and adds those object's references to the **TreeSet** collection.

Listing 3. The fillIt method.

```
    class Populator{
  public static void fillIt(Collection ref){
    ref.add(new MyClass(4));
    ref.add(new MyClass(4));
    ref.add(new MyClass(3));
    ref.add(new MyClass(2));
    ref.add(new MyClass(1));
  }//end fillIt()
}//end class Populator
```

Figure 5.474: Listing 3. The fillIt method.

5.4.9.4.5.5 Similar to previous program

This is essentially the same code that we saw in a sample program in a previous module. In that module, we saw that it was necessary for the class named **MyClass** to implement the **Comparable** interface. Otherwise, the **add** method would throw a runtime exception.

5.4.9.4.5.6 MyClass does not implement Comparable

In that program, however, the **TreeSet** object was instantiated without benefit of a **Comparator** object.

As you can see in the code fragment in Listing 4 (p. 1263) , the class named **MyClass** in this program does not implement the **Comparable** interface.

5.4.9.4.5.7 Comparator eliminates requirement for Comparable

Furthermore, the **add** method in Listing 3 (p. 1262) does not throw a runtime exception. That is because the **TreeSet** object was instantiated with the benefit of a **Comparator** object.

The use of a **Comparator** object in the instantiation of the **TreeSet** object eliminates the requirement for objects stored in the **TreeSet** collection to implement the **Comparable** interface.

Listing 4. The class named MyClass.

```
  class MyClass{
int data;

MyClass(){
  data = 0;
}//end noarg constructor

MyClass(int data){
  this.data = data;
}//end parameterized constructor

public String toString(){
  return "" + data;
}//end overridden toString()
}//end MyClass
```

Figure 5.475: Listing 4. The class named MyClass.

5.4.9.4.6 The class named TheComparator

That brings us to the class named **TheComparator** from which the **Comparator** object was instantiated and passed to the constructor for the **TreeSet** object in Listing 2 (p. 1261) . The declaration for the class named **TheComparator** is shown in Listing 5 (p. 1263) .

Listing 5. Beginning of the class named TheComparator.

```
class TheComparator
implements Comparator,Serializable{
```

Figure 5.476: Listing 5. Beginning of the class named TheComparator.

As you can see, the class named **TheComparator** implements both the **Comparator** interface and the **Serializable** interface.

5.4.9.4.6.1 Implementing the Comparator interface

By implementing the **Comparator** interface, an object instantiated from the class is eligible to be passed to the constructor for a **TreeSet** object, which requires an incoming parameter of type **Comparator** .

5.4.9.4.6.2 Implementing the Serializable interface

Here is what Oracle has to say about implementing the **Serializable** interface:

> *"Note: It is generally a good idea for comparators to implement java.io.Serializable, as they may be used as ordering methods in serializable data structures (like TreeSet, TreeMap). In order for the data structure to serialize successfully, the comparator (if provided) must implement Serializable."*

Since the **Serializable** interface doesn't declare any methods, implementing the interface simply requires a declaration that the interface is being implemented.

5.4.9.4.6.3 Methods of the Comparator interface

The **Comparator** interface declares the two methods listed below:

- public int **compare** (Object o1, Object o2)
- public boolean **equals** (Object obj)

As is always the case when implementing interfaces, a class that implements the **Comparator** interface must provide concrete definitions for both of these methods.

5.4.9.4.6.4 The compare method

The beginning of the **compare** method is shown in Listing 6 (p. 1264) .

Listing 6. Beginning of the compare method.

```
  public int compare(Object o1,Object o2){
if(!(o1 instanceof MyClass))
    throw new ClassCastException();
if(!(o2 instanceof MyClass))
    throw new ClassCastException();
```

Figure 5.477: Listing 6. Beginning of the compare method.

The purpose of a **Comparator** is to compare the values stored in the instance variables of two objects and to return a value indicating which object is *greater* .

5.4.9.4.7 Specialization is required

Generally speaking, therefore, a **Comparator** object must be specialized to deal with a particular type of object. That type could be

- A specific class from which the object is instantiated,
- A specific interface implemented by the class from which the object is instantiated, or perhaps
- A specific superclass of the class from which the object is instantiated.

The code in Listing 6 (p. 1264) confirms that both of the objects to be compared are of the correct type, which in this case is type **MyClass** .

5.4.9.4.7.1 Must gain access to instance variables

Regardless of how the type is established, the code in the **compare** method of the **Comparator** object must gain access to the instance variables of the two objects passed to the **compare** method as type **Object** . This normally requires that a downcast be performed on the incoming object references.

5.4.9.4.7.2 Specialized for type MyClass

This **Comparator** is specialized to compare two objects of the class named **MyClass** . The first two statements in Listing 6 (p. 1264) above confirm that both of the incoming objects are of type **MyClass** . If either object is not of that type, an exception is thrown.

5.4.9.4.7.3 General behavior of compare method

The general description of the behavior of the **compare** method as provided by Oracle is shown below:

> *"Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second."*

5.4.9.4.7.4 Implementation of required behavior

This behavior is accomplished by the code shown in Listing 7 (p. 1265) . In this case, the comparison is based solely on the values of the instance variable named **data** in each of the two objects.

Depending on which object contains the larger value in its instance variable, a value of 1 or -1 is returned. If the two values are equal, a value of 0 is returned.

> *(Note that it is up to the author of the* ***compare*** *method to decide what constitutes larger. This gives the author of the method a great deal of control over the results of a sorting operation.)*

Listing 7. Implementation of required behavior.

```
      if(((MyClass)o1).data < ((MyClass)o2).data)
    return -1;
  if(((MyClass)o1).data > ((MyClass)o2).data)
    return 1;
  else return 0;
}//end compare()
```

Figure 5.478: Listing 7. Implementation of required behavior.

5.4.9.4.7.5 Other stipulations

The documentation for the **compare** method contains several other stipulations regarding the behavior of the method. While I believe that this version of the **compare** method meets all of those stipulations, I haven't taken the time to test it fully. Therefore, it is possible that it may not meet all of the stipulations in terms of its behavior.

5.4.9.4.8 The equals method

Every new class inherits a default version of the **equals** method from the class named **Object.** Therefore, a new class that implements the **Comparator** interface already has such a method. The new class is free to override the inherited version, or to simply make use of the inherited version. Here is what Oracle has to say on the subject:

"Note that it is always safe not to override Object.equals(Object). However, overriding this method may, in some cases, improve performance by allowing programs to determine that two distinct Comparators impose the same order."

5.4.9.4.8.1 Overridden equals method

I decided, for purposes of illustration, to go ahead and override the **equals** method. However, my overridden version, as shown in Listing 8 (p. 1266) isn't very significant. It simply confirms that an object being compared for equality to a **Comparator** object is instantiated from the same class.

Since the **Comparator** object doesn't contain any instance variables, there isn't much more to be tested for equality.

Listing 8. The overridden equals method.

```
  public boolean equals(Object o){
if(!(o instanceof TheComparator))
    return false;
else return true;
 }//end overridden equals()
}//end class TheComparator
```

Figure 5.479: Listing 8. The overridden equals method.

5.4.9.4.9 The program output

Finally, the code shown in Listing 9 (p. 1266) uses an **Iterator** to display the contents of the populated **TreeSet** object.

Listing 9. Display the contents of the TreeSet object.

```
   iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next());
}//end while loop
```

Figure 5.480: Listing 9. Display the contents of the TreeSet object.

The output produced by this code fragment is shown below.

1234

As you can see, the duplicate elements having the value 4 were eliminated as would be expected for a **Set** object. In addition, the **Comparator** was used to accomplish the following contract of a **SortedSet** object:

> *"A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see* **Comparable** *), or by a* **Comparator** *provided at sorted set creation time."*

In this case, the sorted order was controlled by the **Comparator** object, and not by the *natural ordering* of the elements. The *natural ordering* is controlled by implementation of the **Comparable** interface, and the elements in this collection did not implement the **Comparable** interface. Therefore, objects of the class named **MyClass** do not have a natural order in this program.

However, the code in this version of the **Comparator** produced an output order that matches the ascending natural order that one might expect for objects of type **MyClass** . Future modules will show you how to design the **Comparator** object to produce different output orders, such as descending order for example.

5.4.9.5 Run the program

I encourage you to copy the code Listing 2 (p. 1261) and paste it into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.9.6 Summary

This module has discussed and illustrated the use of the **Comparator** interface.

The sorting order established by a **Comparator** may be different or may be the same as the *natural ordering* for a collection of objects .

A **Comparator** can be used to establish a sorting order for objects that don't have a *natural ordering* .

The use of a **Comparator** is an alternative to the implementation of the **Comparable** interface.

A **TreeSet** object instantiated with the benefit of a **Comparator** object doesn't require the objects in its collection to implement **Comparable** .

5.4.9.7 What's next?

In the next module, I will illustrate the use of a **Comparator** to eliminate the effect of case when sorting **String** objects.

5.4.9.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4090: The Comparator Interface, Part 1
- File: Java4090.htm
- Published: 05/07/13

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.10 Java4100: The Comparator Interface, Part 2[286]

5.4.10.1 Table of Contents

5.4.10.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

[286] This content is available online at <http://cnx.org/content/m46190/1.1/>.

This module shows you how to use a **Comparator** object to achieve natural *(ascending)* order on a set of names added as **String** objects to a **TreeSet** collection while ignoring the case used to write the names.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [287] in Oracle's Java Tutorials [288] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.10.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.10.2.1.1 Listings

5.4.10.3 Preview

In this module, I will show you how to use a **Comparator** object to achieve a *natural ordering* of a set of names *(**String** objects)* added to a **TreeSet** collection while ignoring the case used to write the names. *(The natural ordering for **String** objects is ascending.)*

5.4.10.4 Discussion and sample code

5.4.10.4.1 Generics

The code in this module is written with no thought given to Generics [289] . As a result, if you copy and compile this code, you will probably get a warning about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.10.4.2 Beginning with a quiz

Let's begin with a quiz to test your prior knowledge of the Java Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1270) ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Bill Tom JOE BILL TOM
- D. Tom TOM Joe JOE Bill BILL
- E. Joe Bill Tom
- F. None of the above.

[287] http://docs.oracle.com/javase/tutorial/collections/index.html
[288] http://docs.oracle.com/javase/tutorial/index.html
[289] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparator03.

```
    //File Comparator03.java
//Copyright 2001 R.G.Baldwin
import java.util.*;
import java.io.Serializable;

public class Comparator03{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator03

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;
    System.out.println("Natural ordering");
    ref = new TreeSet();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

    System.out.println("Comparator in use");
    ref = new TreeSet(new TheComparator());
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator

class TheComparator
        implements Comparator,Serializable{

  public int compare(Object o1,Object o2) {
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();
```

If your answer was **None of the above** , you are correct.

5.4.10.4.2.1 The program output

The output produced by the program shown in Listing 1 (p. 1270) is four lines long as shown below. *(Note that the bullets shown below do not appear in the actual program output.)*

- **Natural ordering**
- **BILL Bill JOE Joe TOM Tom**
- **Comparator in use**
- **Bill Joe Tom**

5.4.10.4.2.2 From the previous module

In the previous module, I introduced you to the essentials of defining and using a **Comparator** for controlling the sort order of the elements contained in a **TreeSet** collection.

In that module, I explained the difference between *natural ordering* and the *sort ordering* produced through the use of a **Comparator** object.

However, what I showed you generally replicated the *natural ordering* , and therefore, wasn't too exciting.

5.4.10.4.2.3 Doing more with a Comparator

In this and several subsequent modules, I am going to show you some of the things that you can do with a **Comparator** object. By using a **Comparator** object, you can achieve comparisons and sort orders that are different from the *natural ordering* for a given element type.

5.4.10.4.3 Two steps in the program

The program shown in Listing 1 (p. 1270) goes through two major steps.

5.4.10.4.3.1 The first step

First it populates a **TreeSet** collection with the names of six people without using a **Comparator** . Then it displays the contents of that collection using an iterator. That produces the following output *(without the bullets)* :

- **Natural ordering**
- **BILL Bill JOE Joe TOM Tom**

As you will see later, the names were added to the collection in a different order than the output order shown above.

In this step, each of the six names that were added to the collection were displayed after they were arranged into their *natural ordering* .

In case you are unfamiliar with this aspect of character encoding, upper-case characters appear before lower-case characters in the *natural ordering* of characters in the Unicode character set. Therefore, the names consisting of all upper-case characters appear in the output ahead of the same names consisting of a mixture of upper-case and lower-case characters.

5.4.10.4.3.2 The second step

Then the program shown in Listing 1 (p. 1270) instantiates a new **TreeSet** object, providing a **Comparator** for use in comparing and managing the sort order of the elements.

The program populates the new **TreeSet** collection with the same set of six names in the same order as before. After the collection is populated, its contents are displayed producing the following output *(without the bullets)* :

- **Comparator in use**
- **Bill Joe Tom**

5.4.10.4.3.3 Duplicate names eliminated from the set

Three of the names appear in the output in the same order as the *natural ordering* shown earlier. However, the duplicate names are eliminated and only three names appear.

This is because a **Comparator** was used by the **TreeSet** object to compare the elements as they were added. The **Comparator** was designed to eliminate the distinction between upper-case and lower-case characters.

5.4.10.4.3.4 Does Joe equal JOE?

For the earlier case that didn't use a **Comparator** , the names **Joe** and **JOE** were considered to be different elements. Therefore, after population, both names appeared in the collection.

When the **Comparator** was used to eliminate the distinction between upper-case and lower-case characters, the names **Joe** and **JOE** were considered to be duplicates. As a result, only the first of the two was allowed into the collection and the second of the two was rejected.

5.4.10.4.4 Let's see some code

The code shown in Listing 2 (p. 1272) is the code that was used

- To instantiate a **TreeSet** object without a **Comparator** ,
- To populate the collection, and
- To display the contents of the collection after it was populated.

Listing 2. Create, populate, and display a TreeSet collection.

```
    ref = new TreeSet();
Populator.fillIt(ref);

iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.482: Listing 2. Create, populate, and display a TreeSet collection.

5.4.10.4.4.1 The Populator code

The code in Listing 3 (p. 1273) was used to populate the collection in both cases, both with, and without a **Comparator** *(to be discussed later).*

Listing 3. The class named Populator.

```
   class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator
```

Figure 5.483: Listing 3. The class named Populator.

5.4.10.4.4.2 Populating the collection with String objects

Note that in Listing 3 (p. 1273) , unlike earlier modules, I did not use a class of my own design from which to instantiate the objects used to populate the collection. Rather, I used the **String** class from the standard library.

The **String** class implements the **Comparable** interface. Therefore, objects instantiated from the **String** class have a *natural ordering* when placed in a collection.

Because the **compareTo** method of the **String** class, *(which implements the Comparable interface)* considers upper-case and lower-case characters to be different, there were no duplicate elements added to the collection when only the **compareTo** method was used to compare elements. The six **String** objects were simply arranged so that the iterator would return references to those objects in sorted order. This produced the output shown below:

BILL Bill JOE Joe TOM Tom

5.4.10.4.4.3 A TreeSet with a Comparator

The code shown in Listing 4 (p. 1274) was used to instantiate a new **TreeSet** object. A **Comparator** object's reference was passed to the **TreeSet** constructor. The **Comparator** object *(instead of the **compareTo** method)* was subsequently used for comparing and controlling the sorting order of the elements in the **TreeSet** collection.

Listing 4. A TreeSet with a Comparator.

```
    ref = new TreeSet(new TheComparator());

Populator.fillIt(ref);

iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.484: Listing 4. A TreeSet with a Comparator.

The code in Listing 4 (p. 1274) was also used to populate the collection, and to display the contents of the collection after it was populated.

5.4.10.4.4.4 Populating the TreeSet collection

As before, the **fillIt** method shown in Listing 5 (p. 1274) was used to populate the collection. The same six names as before were added to the **TreeSet** collection. However, the result of adding those six names was determined by the behavior of the **compare** method in the **Comparator** object used by the **TreeSet** object for managing the collection. *(Three of the names were rejected as duplicates.)*

Listing 5. Populating the TreeSet collection.

```
  public static void fillIt(Collection ref){
ref.add("Joe");
ref.add("Bill");
ref.add("Tom");
ref.add("JOE");
ref.add("BILL");
ref.add("TOM");
}//end fillIt()
```

Figure 5.485: Listing 5. Populating the TreeSet collection.

5.4.10.4.4.5 Beginning of the Comparator class

The code in Listing 6 (p. 1275) shows the beginning of the class from which the **Comparator** object was instantiated. Note that this class implements the **Comparator** interface, and therefore defines a concrete version of the method named **compare** .

Listing 6. Beginning of the Comparator class.

```
   class TheComparator
          implements Comparator,Serializable{

  public int compare(Object o1,Object o2){
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();
```

Figure 5.486: Listing 6. Beginning of the Comparator class.

Listing 6 (p. 1275) doesn't contain the interesting part of this class. The code in Listing 6 (p. 1275) simply throws an exception if the **compare** method receives any incoming parameters of types other than **String** .

5.4.10.4.4.6 The interesting code

The interesting code in the method named **compare** is shown in Listing 7 (p. 1275) .

Listing 7. The interesting code in the compare method.

```
     int result =
   ((String)o1).toUpperCase().
         compareTo(((String)o2).
                        toUpperCase());
   return result;
}//end compare()
```

Figure 5.487: Listing 7. The interesting code in the compare method.

The code in Listing 7 (p. 1275) makes use of two methods of the **String** class to compare the two incoming objects.

5.4.10.4.4.7 Convert to upper-case

The method named **toUpperCase** is used to produce a version of each of the incoming strings that consists of upper-case characters only. In other words, lower-case characters in each of the two strings are replaced by the corresponding upper-case characters. This conversion occurs before the strings are compared.

For example, the string **Joe** is converted to **JOE** inside the **compare** method, before the actual comparison is made. This results in the two strings containing **Joe** and **JOE** being considered to be duplicates. If one of them is already in the collection when an attempt is made to add the other, the second will be rejected as a duplicate.

5.4.10.4.4.8 Making the comparison

Then the **compareTo** method of the **String** class is used to make the actual comparison. *(Note that this is the same method that is used to make the comparison in the absence of a* ***Comparator*** *object. However, in the case of the* ***Comparator*** *object, the case of the strings is modified before they are passed to the* ***compareTo*** *method.)*

This code calls the **compareTo** method on the upper-case version of the string represented by **o1** , passing the upper-case version of the string represented by **o2** as a parameter. Here is part of what Oracle has to say about the behavior of the **compareTo** method.

> *"Returns: the value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string."*

5.4.10.4.4.9 Just what I was looking for

That is exactly the behavior that I was looking for, so all that I needed to do after calling the **compareTo** method on the upper-case versions of the two strings was to return the value that was returned by the **compareTo** method.

(Note, while writing this module and explaining the behavior of this program, I discovered that I could have used a method of the ***String*** *class named* ***compareToIgnoreCase*** *to accomplish the same thing with a little less work.)*

5.4.10.4.4.10 The results

When the **TreeSet** object used the **Comparator** object to compare and arrange the elements in the collection, the three *duplicate* names were eliminated and the iterator delivered references to the remaining three names in the following order:

Bill Joe Tom

5.4.10.5 Run the program

I encourage you to copy the code from Listing 1 (p. 1270) and paste it into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.10.6 Summary

In this module, I showed you how to use a **Comparator** object to achieve a *natural ordering* of a set of names *(* ***String*** *objects)* added to a **TreeSet** collection while ignoring the case used to write the names. *(Natural ordering for* ***String*** *objects is ascending.)*

5.4.10.7 What's next?

In the next module, I will show you how to use a **Comparator** to cause a **TreeSet** collection containing references to **String** objects to be sorted in descending order while preserving differences in case.

5.4.10.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4100: The Comparator Interface, Part 2
- File: Java4100.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.11 Java4110: The Comparator Interface, Part 3[290]

5.4.11.1 Table of Contents

[290] This content is available online at <http://cnx.org/content/m46191/1.1/>.

5.4.11.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

The module shows you how to use a **Comparator** to cause a **TreeSet** collection to be sorted in *descending* order while preserving the impact of differences in case.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [291] in Oracle's Java Tutorials [292] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.11.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.11.2.1.1 Listings

- Listing 1 (p. 1280) . The program named Comparator04.
- Listing 2 (p. 1281) . A new TreeSet object with a Comparator.
- Listing 3 (p. 1282) . The fillIt method.
- Listing 4 (p. 1282) . Beginning of the class named TheComparator.
- Listing 5 (p. 1283) . The most interesting code.

5.4.11.3 Preview

In this module, I will teach you how to use a **Comparator** to cause a **TreeSet** collection to be sorted in descending order while preserving the impact of differences in case. We might refer to this as *reverse natural order*. In other words, the sorting order is the same as the *natural order* except that the order is descending instead of ascending.

5.4.11.4 Generics

The code in this series of modules is written with no thought given to Generics [293] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.11.5 Discussion and sample code

5.4.11.5.1 Beginning with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1280) ?

- A. Compiler Error
- B. Runtime Error
- C. BILL Bill JOE Joe TOM Tom
- D. Tom TOM Joe JOE Bill BILL
- E. Joe Bill Tom

[291] http://docs.oracle.com/javase/tutorial/collections/index.html
[292] http://docs.oracle.com/javase/tutorial/index.html
[293] http://docs.oracle.com/javase/tutorial/java/generics/index.html

- F. None of the above.

Listing 1. The program named Comparator04.

```
    //File Comparator04.java
//Copyright 2001, R.G.Baldwin

import java.util.*;
import java.io.Serializable;

public class Comparator04{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator04

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new TreeSet(new TheComparator());
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator

class TheComparator implements Comparator,Serializable{

  public int compare(Object o1,Object o2){
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();

    int result = ((String)o1).compareTo(((String)o2));
    return result*(-1);
  }//end compare()
```



```
  public boolean equals(Object o){
    if(!(o instanceof TheComparator))
        return false;
    else return true;
```

If you selected the following, you are correct:
D. Tom TOM Joe JOE Bill BILL

5.4.11.5.1.1 Similar to previous programs

The overall structure of this program is very similar to programs that I have discussed in previous modules. Therefore, I will concentrate on those aspects of this program that differentiate it from the programs in previous modules.

5.4.11.5.1.2 A new TreeSet object with a Comparator

The code in Listing 2 (p. 1281) instantiates a new **TreeSet** object, by providing a reference to an anonymous object that implements the **Comparator** interface. That object is instantiated from the class named **TheComparator** . It is the **Comparator** object that will be of most interest to us in this module.

Listing 2. A new TreeSet object with a Comparator.

```
   Collection ref;

ref = new TreeSet(new TheComparator());
Populator.fillIt(ref);

iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.489: Listing 2. A new TreeSet object with a Comparator.

5.4.11.5.1.3 Populating the TreeSet collection

After the **TreeSet** object is instantiated, it is passed to a method named **fillIt** where the **TreeSet** collection is populated with the names of several people.

5.4.11.5.1.4 Display the contents of the TreeSet collection

As shown by the code in Listing 2 (p. 1281) , after the **TreeSet** collection is populated, an **Iterator** is obtained for that collection and used to display the contents of the collection. The output produced by the program is **shown below** :
Tom TOM Joe JOE Bill BILL

5.4.11.5.2 Analyzing the contents of the TreeSet collection

We will need to compare this output with the names used to populate the collection to appreciate the true significance of the use of the **Comparator** object.

At this point, it is worth pointing out that the six names contained in the collection are returned by the iterator in *descending order* , taking the significance of upper and lower case into account. In other words,

names beginning with letters that are high in the alphabet occur before names beginning with letters that are lower in the alphabet. In addition, names containing lower case characters appear before the same names containing only upper case characters.

5.4.11.5.2.1 Method used to populate the collection

Listing 3 (p. 1282) shows the method named **fillIt** that was used to populate the collection with references to six **String** objects. As you can see, the names weren't added in any particular order.

As you can also see by comparing Listing 3 (p. 1282) with the output shown above (p. 1281) , all six names that were added to the collection were displayed in the output, but in a different order from the order in which they were added. *(Names with the same spelling but different case were not considered to be duplicates insofar as the contract for the set was concerned.)*

Listing 3. The fillIt method.

```
    public static void fillIt(Collection ref){
  ref.add("Joe");
  ref.add("Bill");
  ref.add("Tom");
  ref.add("JOE");
  ref.add("BILL");
  ref.add("TOM");
}//end fillIt()
```

Figure 5.490: Listing 3. The fillIt method.

5.4.11.5.2.2 Implementing the Comparator interface

That brings us to the class from which the **Comparator** object was instantiated. The beginning portion of that class is shown in Listing 4 (p. 1282) .

Listing 4. Beginning of the class named TheComparator.

```
  class TheComparator implements Comparator,Serializable{

public int compare(Object o1,Object o2){
  if(!(o1 instanceof String))
      throw new ClassCastException();
  if(!(o2 instanceof String))
      throw new ClassCastException();
```

Figure 5.491: Listing 4. Beginning of the class named TheComparator.

The code in Listing 4 (p. 1282) isn't particularly interesting. That code simply throws an exception if either of the references passed to the **compare** method refer to an object of some type other than **String** .

5.4.11.5.2.3 The interesting code

The most interesting code is shown in Listing 5 (p. 1283) . The first statement in Listing 5 (p. 1283) uses the **compareTo** method of the **String** class to compare the two objects in an **ascending** *natural ordering* sense. The behavior of this method is more formally described as follows:

> *"Returns: the value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string."*

Listing 5. The most interesting code.

```
   int result = ((String)o1).compareTo(((String)o2));

return result*(-1);
```

Figure 5.492: Listing 5. The most interesting code.

5.4.11.5.2.4 Converting to reverse natural order

The most interesting line of code in this entire program is the **return** statement shown in Listing 5 (p. 1283) . This line of code changes the sign on the value returned by the **compareTo** method before returning it as the return value for the **compare** method.

The effect of changing the sign is to return a value that causes the **TreeSet** collection to arrange the elements in **reverse** *natural order* instead of the normal **ascending** *natural order.*

As a result, the use of an iterator to access and display the contents of the collection produces the following output:

Tom TOM Joe JOE Bill BILL

For comparison, if the names were arranged in **ascending** *natural order,* the output would be as shown below:

BILL Bill JOE Joe TOM Tom

5.4.11.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1280) and paste it into your Java text editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.11.7 Summary

In this module, I taught you how to use a **Comparator** to cause a **TreeSet** collection to be sorted in *reverse natural order*. In other words, the sorting order is the same as the *natural order* except that the order is descending instead of ascending.

5.4.11.8 What's next?

In the next module, I will show you how to use a **Comparator** object to sort the contents of an array.

5.4.11.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name:Java4110: The Comparator Interface, Part 3
- File: Java4110.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.12 Java4120: The Comparator Interface, Part 4[294]

5.4.12.1 Table of Contents

[294] This content is available online at <http://cnx.org/content/m46192/1.1/>.

5.4.12.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module shows you how to extract the contents of a collection into an array, and how to use a **Comparator** object to sort the contents of the array into reverse natural order. The module also shows you how to sort the contents of the array into natural order without the use of a **Comparator** object.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [295] in Oracle's Java Tutorials [296] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.12.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.12.2.1.1 Listings

[295] http://docs.oracle.com/javase/tutorial/collections/index.html
[296] http://docs.oracle.com/javase/tutorial/index.html

5.4.12.3 Preview

The primary purpose of recent modules in this series was to teach you about the interactions between the **Comparator** interface and the Collections Framework.

This module departs somewhat from that primary purpose and teaches you how to use a **Comparator** object to sort the contents of an array containing references to objects. Technically speaking, an array is not part of the core Collections Framework. However, it is definitely a first cousin to the Framework.

5.4.12.3.1 An array is a container

As you should already know, an array is a container that can be used to store a collection of primitive values or a collection of references to objects.

The **Collection** interface declares a method named **toArray** , which can be called on a **Collection** object to *"return an array containing all of the elements in this collection whose runtime type is that of the specified array"* .

5.4.12.3.2 An opportune time

Since you are studying this sub-series of modules to learn about the uses of the **Comparator** interface, this seems like an opportune time to teach you how to get an array from a collection, and how to use the **Comparator** interface to sort the contents of the array. *(While I'm at it, I will also teach you how to sort the elements in an array of object references into natural order without the use of a Comparator object.)*

5.4.12.4 Generics

The code in this series of modules is written with no thought given to Generics [297] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.12.5 Discussion and sample code

5.4.12.5.1 Beginning with a quiz

See if you can write down the output produced by the program shown in Listing 1 (p. 1287) .

[297] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparator 05.

```
    //File Comparator05.java
//Copyright 2001, R.G.Baldwin
import java.util.*;
import java.io.Serializable;

public class Comparator05{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator05

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;
    Object[] array;

    ref = new Vector();
    Populator.fillIt(ref);
    iter = ref.iterator();
    System.out.println("Collection data");
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

    array = ref.toArray();
    System.out.println("Raw array data");
    display(array);

    //Sort the array into natural order
    // and display it.
    Arrays.sort(array);
    System.out.println("Natural order sorted " +
                                 "array data");
    display(array);

    //Sort the array into custom order
    // and display it.
    Arrays.sort(array, new TheComparator());
    System.out.println("Custom order sorted " +
                                "array data");
    display(array);

    iter = ref.iterator();
    System.out.println("Collection data");
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();
```



```
  }//end doIt()

  static void display(Object[] array){
    for(int i = 0; i < array.length;i++){
```

If you wrote down the following for the program output, you already understand most of the material covered in this module and you can probably skip this module and move on to the next module.

NOTE:

```
    Collection data
Joe Bill Tom JOE BILL TOM
Raw array data
Joe Bill Tom JOE BILL TOM
Natural order sorted array data
BILL Bill JOE Joe TOM Tom
Custom order sorted array data
Tom TOM Joe JOE Bill BILL
Collection data
Joe Bill Tom JOE BILL TOM
```

If you didn't write down the correct output for the program in Listing 1 (p. 1287) , you should probably continue with your study of this module.

5.4.12.5.1.1 Similar to previous programs

Although this program is somewhat more complex, the overall structure of this program is similar to programs that I have discussed in previous modules. Therefore, I will concentrate on those aspects of this program that differentiate it from the programs in previous modules.

5.4.12.5.1.2 A new Vector object

The code in Listing 2 (p. 1288) instantiates a new object of the **Vector** class and stores a reference to that object in the variable named **ref** .

Listing 2. A new Vector object.

```
ref = new Vector();
```

Figure 5.494: Listing 2. A new Vector object.

The **Vector** class was part of Java long before the Collections Framework was released. However, with the release of the Collections Framework, the **Vector** class was upgraded to implement the **Collection** interface and the **List** interface.

5.4.12.5.1.3 A Vector is a List

Therefore, a **Vector** is a **List** , and adheres to the various contracts of the **List** interface. For example, since it is not a **Set** , it doesn't prohibit duplicate elements. Because it is a **List** , it is an *ordered* collection. The position of each element in the collection is determined by a numeric index associated with the element and is independent of the value of the element.

5.4.12.5.1.4 The fillIt method

As has been the case in several of the programs in previous modules, the code in Listing 3 (p. 1289) passes the **Vector** object's reference to a method named **fillIt** where the **Vector** is populated with the names of several people.

Listing 3. Call the fillIt method.

```
Populator.fillIt(ref);
```

Figure 5.495: Listing 3. Call the fillIt method.

The code for the **fillIt** method is shown in Listing 4 (p. 1289) . As you can see, the names were added to the collection in no particular order relative to their values. *(The add method for the Vector class simply adds each new element to the end of the list.)*

Listing 4. The fillIt method.

```
    class Populator{
  public static void fillIt(
                      Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator
```

Figure 5.496: Listing 4. The fillIt method.

5.4.12.5.1.5 Iteration on a Vector

When an iterator is used to traverse the elements in a **Vector** collection, the elements are delivered by the iterator in ascending index order, beginning with the element stored at index 0.

The code in Listing 5 (p. 1290) gets and uses an iterator to display the contents of the populated collection.

Listing 5. Iteration on a Vector.

```
    iter = ref.iterator();
System.out.println("Collection data");
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.497: Listing 5. Iteration on a Vector.

5.4.12.5.1.6 The output

The code in Listing 5 (p. 1290) produces the following output:

NOTE:

```
    Collection data
Joe Bill Tom JOE BILL TOM
```

As you can see, this is the same order in which the names were added to the collection by the **fillIt** method in Listing 4 (p. 1289) .

5.4.12.5.2 The toArray method

The code in Listing 6 (p. 1290) is new to this module. This code calls the **toArray** method on the **Vector** object to extract the contents of the collection and store the elements in an array object of type **Object** .

Listing 6. Call the toArray method.

```
    array = ref.toArray();
```

Figure 5.498: Listing 6. Call the toArray method.

(Recall that the variable named ***array*** *was declared as a reference to an array object of type* ***Object*** *in Listing 1* (p. 1287) *.)*

5.4.12.5.2.1 The contract

According to the documentation for the **Vector** class, this version of the **toArray** method:

"Returns an array containing all of the elements in this Vector in the correct order."

The documentation for the **toArray** method of the **Collection** interface is a little more verbose, reading partially as follows:

"Returns an array containing all of the elements in this collection. If the collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

5.4.12.5.2.2 Elements are returned in ascending index order

By default, the iterator for a **Vector** returns its elements in ascending index order. Therefore, the **toArray** method for a **Vector** object must return the elements in the same order.

5.4.12.5.2.3 A "safe" array

Also, according to Oracle:

"The returned array will be "safe" in that no references to it are maintained by this collection. ... The caller is thus free to modify the returned array."

In the code in Listing 6 (p. 1290) above, the returned reference to an array object is assigned to a reference variable that previously contained null. Following the execution of the **toArray** method, that reference variable refers to an array object of type **Object** containing the same elements as the **Vector** collection, in ascending index order.

(Regarding the concept of a "safe" array, it is easy to demonstrate that the elements in the array refer to the same objects referred to by the elements in the Vector. Thus, using the references stored in the array to modify the objects to which they refer also modifies the objects referred to by the elements stored in the Vector. In other words, the elements in the array are copies of the elements in the Vector. The elements in the array refer to the original objects, and do no refer to copies of those objects. As usual when dealing with multiple references to objects, care should be taken to avoid inadvertently corrupting those objects.)

5.4.12.5.2.4 Display the contents of the array

The code in Listing 7 (p. 1291) passes the array object's reference to a method named **display** that displays the contents of the array in ascending index order.

Listing 7. Display the contents of the array.

```
    System.out.println("Raw array data");
display(array);
```

Figure 5.499: Listing 7. Display the contents of the array.

The output produced by the code in Listing 7 (p. 1291) is as shown below:

NOTE:

```
    Raw array data
Joe Bill Tom JOE BILL TOM
```

As you can see, this is the same data, in the same order, as the contents of the collection displayed earlier.

(The method named ***display*** *is a simple utility method that I won't discuss here because of its simplicity. You can view the display method in its entirety in Listing 1 (p. 1287) .)*

5.4.12.5.3 Sorting the array into *natural order*

The code in Listing 8 (p. 1292) is also new to this module. This code uses one of the overloaded **sort** methods of the **Arrays** class to sort the contents of the array into *natural order* .

Listing 8. Sorting the array into natural order.

```
Arrays.sort(array);
```

Figure 5.500: Listing 8. Sorting the array into natural order.

Here is part of what Oracle has to say about the **Arrays** class:

"This class contains various methods for manipulating arrays (such as sorting and searching)."

The class contains many overloaded versions of the **sort** method. Here is part of what Oracle has to say about the version of the **sort** method used in Listing 8 (p. 1292) above:

"Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface."

5.4.12.5.3.1 The Comparable interface and polymorphic behavior

Although the declared type of the array is **Object** , the array actually contains references to **String** objects.

The **String** class implements the **Comparable** interface. It is not necessary to cast the array to type **String** before passing it to the **Sort** method. *(The* ***Sort*** *method declares the incoming parameter as type* ***Object*** *.)*

The **sort** method treats the array elements as type **Comparable** and uses the **compareTo** method declared in that interface to perform any necessary comparisons required to carry out the sorting operation.

This is another example of the usefulness of polymorphism as implemented through the use of the Java interface. *(The* ***Comparable*** *interface and the* ***compareTo*** *method declared in that interface were discussed in detail in an earlier module.)*

5.4.12.5.3.2 Display the sorted array data

The code in Listing 9 (p. 1293) displays the contents of the array after those contents are sorted into *natural order* by the **sort** method in Listing 8 (p. 1292) above.

Listing 9. Display the sorted array data.

```
    System.out.println("Natural order sorted " +
                                        "array data");
display(array);
```

Figure 5.501: Listing 9. Display the sorted array data.

The output produced by Listing 9 (p. 1293) above is:

NOTE:

```
    Natural order sorted array data
BILL Bill JOE Joe TOM Tom
```

5.4.12.5.3.3 The *natural order* for String objects

I discussed the concept of *natural ordering* in a previous module with particular emphasis of the *natural order* for strings. You will recognize that the strings shown in the above output have been sorted into *natural order* according to the definition of the **compareTo** method of the **String** class.

5.4.12.5.4 Sort the array with a Comparator

The code in Listing 10 (p. 1293) is also new to this module. This code uses a different version of the overloaded **sort** method of the **Arrays** class to sort the array using the rules defined in the **compare** method of a **Comparator** object *(passed as a parameter to the sort method)*.

Listing 10. Sort the array with a Comparator.

```
    Arrays.sort(array, new TheComparator());
```

Figure 5.502: Listing 10. Sort the array with a Comparator.

5.4.12.5.4.1 What does Oracle have to say about this?

Here is part of what Oracle has to say about this version of the **sort** method of the **Arrays** class:

> *"Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array)."*

5.4.12.5.4.2 The class named TheComparator

Listing 11 (p. 1294) shows the class from which the **Comparator** object was instantiated.

This is essentially the same class that was used to instantiate a **Comparator** object in an earlier module. I discussed the **compare** method in detail in that module and won't repeat that discussion here.

Listing 11. The class named TheComparator.

```
    class TheComparator implements Comparator,Serializable{

  public int compare(Object o1,Object o2){
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();

    int result = ((String)o1).compareTo(((String)o2));
    return result*(-1);
  }//end compare()
}//end class TheComparator
```

Figure 5.503: Listing 11. The class named TheComparator.

Suffice it to say at this point that this **Comparator** object causes the elements in the array to be sorted into *reverse natural order* . That term was also explained in the previous module, so I won't discuss it further here.

5.4.12.5.4.3 Display the array contents again

The code in Listing 12 (p. 1294) was used to display the newly-sorted contents of the array.

Listing 12. Display the contents again.

```
    System.out.println("Custom order sorted " +
                                        "array data");
display(array);
```

Figure 5.504: Listing 12. Display the contents again.

The output produced by this code is:

NOTE:

```
    Custom order sorted array data
Tom TOM Joe JOE Bill BILL
```

You will recognize this as *reverse natural order* for the elements contained in the array.

5.4.12.5.4.4 Could have sorted differently

It is important to note that I could have caused the sorting order to be different from *reverse natural order* simply by defining the rules used for comparison in the **compare** method shown in Listing 11 (p. 1294) above. This makes it possible for you to sort array data into any order that you choose as long as you can write the sorting rules into the **compare** method of a class that implements the **Comparator** interface.

5.4.12.5.5 Display the collection data again

Finally, in order to show that none of this has disturbed the contents of the original collection, the code in Listing 13 (p. 1295) gets and uses an iterator to display the contents of the **Vector** collection.

Listing 13. Display the collection data again.

```
    iter = ref.iterator();
System.out.println("Collection data");
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.505: Listing 13. Display the collection data again.

The output produced by the code in Listing 13 (p. 1295) is:

NOTE:

```
    Collection data
Joe Bill Tom JOE BILL TOM
```

If you compare this with the output produced by the code at the beginning of the program, you will see that the iterator still returns the elements in the **Vector** in the same order that they were added. Thus, modifications to the array did not disturb the contents of the **Vector** collection.

5.4.12.5.6 The bottom line

The **toArray** method of the **Collection** interface makes it possible to extract a copy of the elements in a collection into an array and to manipulate those elements in whatever way you wish. As mentioned earlier, however, care should be exercised to make certain that the copies of the references to the original objects are not used to corrupt the objects.

The various versions of the **sort** method in the **Arrays** class make it possible to sort the contents of arrays in a variety of different ways.

5.4.12.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1287) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.12.7 Summary

In this module, I taught you how to extract the contents of a collection into an array and how to use a **Comparator** to sort the contents of the array into *reverse natural order* .

Although I elected to use *reverse natural order* for purposes of illustration, I could have sorted the array into some other order simply by defining the comparison rules in the **compare** method of the **Comparator** class differently.

In order to further expand your knowledge of array sorting, I also sorted the array into *natural order* without the use of a **Comparator** .

Sorting the contents of the array did not disturb the contents of the **Vector** collection from which the contents of the array were derived.

5.4.12.8 What's next?

In the next module, I will show you how to use the **sort** method of the **Collections** class along with a **Comparator** object to sort the contents of a **List** .

5.4.12.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4120: The Comparator Interface, Part 4
- File: Java4120.htm
- Published: 05/07/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.13 Java4130: The Comparator Interface, Part 5[298]

5.4.13.1 Table of Contents

[298] This content is available online at <http://cnx.org/content/m46193/1.1/>.

5.4.13.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module shows you how to use the **sort** method of the **Collections** class along with a **Comparator** object to sort the contents of a **List** into reverse natural order.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [299] in Oracle's Java Tutorials [300] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.13.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.13.2.1.1 Listings

[299] http://docs.oracle.com/javase/tutorial/collections/index.html
[300] http://docs.oracle.com/javase/tutorial/index.html

5.4.13.3 Preview

In this module, I will teach you how to use the **sort** method of the **Collections** class along with a **Comparator** object to sort the contents of a **List** into *reverse natural order* .

The methodology that I will teach you is completely general, and can be used to sort a list in a wide variety of ways, depending on how you define the **compare** method of a **Comparator** object.

Furthermore, the same **sort** method and the same **Comparator** object can be used to sort any implementation of a list, so long as the list properly implements the **List** interface.

5.4.13.4 Generics

The code in this series of modules is written with no thought given to Generics [301] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.13.5 Discussion and sample code

5.4.13.5.1 Beginning with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1299) ?

- A. Compiler Error
- B. Runtime Error
- C. BILL Bill JOE Joe TOM Tom
- D. Tom TOM Joe JOE Bill BILL
- E. Joe Bill Tom
- F. None of the above.

[301] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparator06.

```
    //File Comparator06.java
//Copyright 2001, R.G.Baldwin
import java.util.*;
import java.io.Serializable;

public class Comparator06{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator06

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new LinkedList();
    Populator.fillIt(ref);
    Collections.sort((List)ref, new TheComparator());
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator

class TheComparator implements Comparator,Serializable{

  public int compare(Object o1,Object o2){
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();

    //Do an upper-case comparison
    int result = ((String)o1).compareTo(((String)o2));
    return result*(-1);
  }//end compare()
}//end class TheComparator
```


Figure 5.506: Listing 1. The program named Comparator06.

The output produced by this program is shown below:

Tom TOM Joe JOE Bill BILL

If that was your answer, you probably already understand most of the material covered in this module. In that case, you might consider skipping this module and moving on to the next module. If that wasn't your answer, you should probably continue with your study of this module.

5.4.13.5.1.1 Similar to previous programs

The overall structure of the program in Listing 1 (p. 1299) is similar to programs that I have discussed in previous modules. Therefore, I will concentrate on those aspects of this program that differentiate it from the programs in previous modules.

5.4.13.5.1.2 A new LinkedList object

The code in Listing 2 (p. 1300) instantiates a new **LinkedList** object and passes that object's reference to a method named **fillIt** where it is populated with the names of several people.

Listing 2. A new LinkedList object.

```
   ref = new LinkedList();
Populator.fillIt(ref);
```

Figure 5.507: Listing 2. A new LinkedList object.

The **LinkedList** class is one of the concrete implementation classes of the Collections Framework. This class implements the **Collection** interface and the **List** interface. Here is part of what Oracle has to say about the **LinkedList** class:

> *"Linked list implementation of the List interface. Implements all optional list operations, and permits all elements (including null). In addition to implementing the List interface, the LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque)."*

5.4.13.5.1.3 Populating the List

The code in Listing 3 (p. 1301) shows the **fillIt** method that is used to populate the list with references to six different **String** objects.

The **add** method is used to add each new element to the end of the list. As you can see, the elements are added to the list in no particular order with respect to their values.

Listing 3. The fillIt method.

```
    class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator
```

Figure 5.508: Listing 3. The fillIt method.

5.4.13.5.1.4 Displaying the list

Although I didn't bother to do so in this program, if an iterator were to be used to access and display the elements in the list following the invocation of the **fillIt** method, the result would be as shown below:

Joe Bill Tom JOE BILL TOM

As you can see, this is the same as the order in which the elements are added to the list. The first element is added to the list at index value 0 and the sixth element is added to the list at index value 5.

5.4.13.5.1.5 Sort the list

The code shown in Listing 4 (p. 1301) is new to this module. This code uses the **sort** method of the **Collections** class, along with a **Comparator** object to sort the contents of the list.

Listing 4. Sort the list.

```
Collections.sort((List)ref, new TheComparator());
```

Figure 5.509: Listing 4. Sort the list.

*The sort method expects to receive an incoming parameter of type **List** . Therefore, it was necessary to cast the reference from type **Collection** to type **List** .*

5.4.13.5.1.6 A very important point

Unlike the programs in previous modules that simply extracted the contents of the collection into an array and sorted the array, this code actually rearranges the contents of the list according to the sorting rules.

(The programs in previous modules that sorted the array did not rearrange the contents of the list. Only the contents of the array were rearranged.)

Thus, the relationship between an element in the list and the index associated with that element can change as a result of the sorting operation shown in Listing 4 (p. 1301) .

Following the sort, when an iterator is used to access the elements, the elements will be returned by the iterator in the newly-sorted order.

5.4.13.5.2 The Collections class

Despite the similarity of the names, the **Collections** class is different from the **Collection** interface. Here is part of what Oracle has to say about the **Collections** class:

"This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends."

5.4.13.5.2.1 The sort method

The **Collections** class provides a large number of very interesting and useful methods, such as **binarySearch** , **copy** , **reverse** , and **reverseOrder** . *(The reverseOrder method will be examined in the next module.)*

One of the static methods of the **Collections** class is the **sort** method. One overloaded version of the **sort** method can be used to sort a list into the *natural ordering* of its elements. Another overloaded version sorts a list according to the order induced by a **Comparator** .

Here is part of what Oracle has to say about this second version of the **sort** method that uses a **Comparator** :

public static void sort(List list, Comparator c) *"Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be mutually comparable using the specified comparator ...*

The specified list must be modifiable, but need not be resizable. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the n2 log(n) performance that would result from attempting to sort a linked list in place."

5.4.13.5.2.2 Also uses an array

I find it interesting that the **sort** method uses an array as an intermediary in the sorting process. However, the difference between this approach and the approach involving arrays shown in previous modules is given by the following excerpt from the above quotation:

"iterates over the list resetting each element from the corresponding position in the array"

In other words, after sorting the array, the **sort** method uses the sorted results in the array to rearrange the positions of the elements in the list, resulting in a sorted list.

5.4.13.5.2.3 A flexible approach to sorting

Thus, the **sort** method of the **Collections** class can be used to sort the elements in a list using whatever set of comparison rules you program into the **compare** method of the **Comparator** object. Furthermore, it doesn't matter how the list is actually implemented so long as it properly implements the **List** interface.

5.4.13.5.3 The Comparator

The code in Listing 5 (p. 1303) shows the class from which the **Comparator** object was instantiated.

Listing 5. The Comparator.

```
    class TheComparator implements Comparator,Serializable{

  public int compare(Object o1,Object o2){
    if(!(o1 instanceof String))
        throw new ClassCastException();
    if(!(o2 instanceof String))
        throw new ClassCastException();

    int result = ((String)o1).compareTo(((String)o2));
    return result*(-1);
  }//end compare()
}//end class TheComparator
```

Figure 5.510: Listing 5. The Comparator.

I have presented and explained this class in previous modules, so I won't discuss it in detail again here. Suffice it for now to say that an object instantiated from this class will induce the list to be sorted into *reverse natural order* .

5.4.13.5.4 Display the sorted list

The code in Listing 6 (p. 1303) gets and uses an iterator to display the contents of the sorted list.

Listing 6. Display the sorted list.

```
       iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
```

Figure 5.511: Listing 6. Display the sorted list.

The output produced by the code in Listing 6 (p. 1303) is shown below:
Tom TOM Joe JOE Bill BILL
As you can see, this is *reverse natural order* as induced by the **Comparator** object.

5.4.13.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1299) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.13.7 Summary

In this module, I taught you how to use the sort method of the **Collections** class along with a **Comparator** object to sort the contents of a list.

By using this approach, you can sort the contents of list according to any set of comparison rules that you can program into the **compare** method of the **Comparator** object.

Furthermore, the ability to sort the list is independent of the actual implementation of the list, so long as the list properly implements the **List** interface. For example, the same **Comparator** object *(and the same code)* can be used to sort an **ArrayList** , a **LinkedList** , or a **Vector** , producing the same results regardless of which class the list object is instantiated from.

5.4.13.8 What's next?

In the next module, I will show you how to use a **Comparator** created by the **reverseOrder** method of the **Collections** class to sort a list into *reverse natural order* . I will also show you how to use the **reverse** method of the **Collections** class to reverse the order of the elements in a list.

5.4.13.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4130: The Comparator Interface, Part 5
- File: Java4130.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.14 Java4140: The Comparator Interface, Part 6[302]

5.4.14.1 Table of Contents

5.4.14.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module shows you how to use a **Comparator** created by the **reverseOrder** method of the **Collections** class to sort a list into reverse natural order. The module also shows you how to use the **reverse** method of the **Collections** class to reverse the order of the elements in a list.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [303] in Oracle's Java Tutorials [304] . The modules in this collection are intended to supplement and not to replace those tutorials.

[302] This content is available online at <http://cnx.org/content/m46194/1.1/>.
[303] http://docs.oracle.com/javase/tutorial/collections/index.html
[304] http://docs.oracle.com/javase/tutorial/index.html

5.4.14.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.14.2.1.1 Listings

5.4.14.3 Preview

In this module, I will teach you how to use a **Comparator** created by the **reverseOrder** method of the **Collections** class to sort a list into *reverse natural order* . I will also teach you how to use the **reverse** method of the **Collections** class to reverse the order of the elements in a list.

5.4.14.4 Generics

The code in this series of modules is written with no thought given to Generics [305] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.14.5 Discussion and sample code

5.4.14.5.1 Beginning with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework.

What output is produced by the program shown in Listing 1 (p. 1307) *(select one or more answers)* ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Bill Tom JOE BILL TOM
- D. BILL Bill JOE Joe TOM Tom
- E. TOM BILL JOE Tom Bill Joe
- F. Joe Bill Tom JOE TOM BILL
- G. Tom TOM Joe JOE Bill BILL
- H. Joe Bill Tom
- I. None of the above.

[305] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named Comparator07.

```
    //File Comparator07.java
//Copyright 2001, R.G.Baldwin
import java.util.*;

public class Comparator07{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class Comparator07

class Worker{
  public void doIt(){
    Iterator iter;
    Collection ref;

    ref = new ArrayList();
    Populator.fillIt(ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

    Collections.reverse((List)ref);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

    Comparator aComparator= Collections.reverseOrder();
    Collections.sort((List)ref, aComparator);
    iter = ref.iterator();
    while(iter.hasNext()){
      System.out.print(iter.next() + " ");
    }//end while loop
    System.out.println();

  }//end doIt()
}// end class Worker

class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator
```


Figure 5.512: Listing 1. The program named Comparator07.

5.4.14.5.1.1 And the answer is ...

The correct answer to the above question is C, E, and G. The output from the program is shown below:

NOTE:

```
    Joe Bill Tom JOE BILL TOM
TOM BILL JOE Tom Bill Joe
Tom TOM Joe JOE Bill BILL
```

If that was your answer, you probably already understand most of the material covered in this module. In that case, you might consider skipping this module and moving on to the next module. If that wasn't your answer, you should probably continue with your study of this module.

5.4.14.5.1.2 Similar to previous programs

The overall structure of this program in Listing 1 (p. 1307) is similar to programs that I have discussed in previous modules. Therefore, I will concentrate on those aspects of this program that differentiate it from the programs in previous modules.

5.4.14.5.1.3 A new ArrayList object

The code in Listing 2 (p. 1308) instantiates a new **ArrayList** object and passes that object's reference to a method named **fillIt** where it is populated with the names of several people.

Listing 2. A new ArrayList object.

```
ref = new ArrayList();
Populator.fillIt(ref);

iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.513: Listing 2. A new ArrayList object.

5.4.14.5.1.4 Displays the list contents

The code in Listing 2 (p. 1308) also gets an iterator on the list and uses that iterator to display the contents of the populated list. At that point in the program, the list contains the following elements in the order shown:

Joe Bill Tom JOE BILL TOM

You will recognize this as matching the order in which the elements were added to the list by the **fillIt** method shown in Listing 3 (p. 1309) .

Listing 3. The fillIt method.

```
    class Populator{
  public static void fillIt(Collection ref){
    ref.add("Joe");
    ref.add("Bill");
    ref.add("Tom");
    ref.add("JOE");
    ref.add("BILL");
    ref.add("TOM");
  }//end fillIt()
}//end class Populator
```

Figure 5.514: Listing 3. The fillIt method.

5.4.14.5.1.5 The ArrayList class

The **ArrayList** class is one of the concrete class implementations of the Collections Framework. This class implements both the **Collection** interface and the **List** interface. Therefore, it is both a collection and a list, and adheres to the contracts and stipulations of both interfaces.

Here is part of what Oracle has to say about the **ArrayList** class:

> *"Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. ... (This class is roughly equivalent to Vector, except that it is unsynchronized.)"*

5.4.14.5.1.6 The reverse method of the Collections class

The call to the **reverse** method shown in Listing 4 (p. 1309) is new to this module.

Listing 4. The reverse method of the Collections class.

```
Collections.reverse((List)ref);

iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.515: Listing 4. The reverse method of the Collections class.

5.4.14.5.2 The Collections class

A previous module discussed the **Collections** class, indicating that the class provides a number of static methods that can be used to manipulate collections. As a refresher, here is part of what Oracle has to say about the **Collections** class:

> *"This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends."*

You should recall that the **Collections** class is not the same as the **Collection** interface. Don't confuse the two.

5.4.14.5.2.1 The reverse method

One of the static methods in the **Collections** class is the method named **reverse** . Here is part of what Oracle has to say about the **reverse** method:

> *"Reverses the order of the elements in the specified list."*

Pretty simple, huh? But also very useful in some cases.

5.4.14.5.2.2 Contents of the list

After calling the reverse method on the list, the code in Listing 4 (p. 1309) above used an iterator to get and display the contents of the list. The contents of the list at that point in the program were as shown below:

TOM BILL JOE Tom Bill Joe

If you compare this with the previous output, you will see that the locations of the elements in the list are reversed. The element at index 0 was moved to index 5, the element at index 5 was moved to index 0, and the elements in between were moved accordingly.

5.4.14.5.2.3 The reverseOrder method

The code in Listing 5 (p. 1310) is also new to this module. This code calls the static **reverseOrder** method of the **Collections** class and stores the returned value in a reference variable of type **Comparator** .

Listing 5. The reverseOrder method.

```
Comparator aComparator= Collections.reverseOrder();
```

Figure 5.516: Listing 5. The reverseOrder method.

5.4.14.5.2.4 What does Oracle have to say about this?

Here is part of what Oracle has to say about the **reverseOrder** method:

> *"Returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface. (The natural ordering is the ordering imposed by the objects' own compareTo method.) This enables a simple idiom for sorting (or maintaining) collections (or arrays) of objects that implement the Comparable interface in reverse-natural-order."*

5.4.14.5.2.5 Reverse natural order

You will recall that in several previous modules, I have written a class from which I instantiated a **Comparator** object that was used to sort elements into *reverse natural order* . I chose that sorting order simply because I needed to illustrate how to define such a class, and in my specific cases, *reverse natural order* was relatively easy to implement. *(With a little more effort, I could have implemented a variety of different sorting orders.)*

In my design of those classes, I made no attempt to write a generic class that could do the job independently of the type of the elements to be sorted. Rather, my **Comparator** objects tended to be very type specific.

5.4.14.5.3 A type-independent Comparator

What we see here is much more general and sophisticated. The **Comparator** object returned by the **reverseOrder** method can be used to impose a *reverse natural order* on any collection of objects that implement the **Comparable** interface. Thus, the class from which the objects are instantiated doesn't matter, as long as those classes implement the **Comparable** interface. *(I also discussed the Comparable interface in some detail in an earlier module. You may want to refer back to that module to learn more about it.)*

5.4.14.5.3.1 The wonderful world of the Java interface

Here again, we see a manifestation of the benefits of polymorphism as implemented using the Java interface. *(I frequently tell my students that if they don't understand interfaces, they can't possibly understand Java.)*

5.4.14.5.3.2 Sorting the list

The code in Listing 6 (p. 1311) is not new to this module. An earlier module discussed the use of the **sort** method of the **Collections** class, along with a **Comparator** object to sort a list.

Listing 6. Sorting the list.

```
Collections.sort((List)ref, aComparator);
```

Figure 5.517: Listing 6. Sorting the list.

5.4.14.5.3.3 Source of Comparator object is new

The thing that is new to this module is the source of the **Comparator** object provided to the **sort** method in Listing 6 (p. 1311) .

In the previous modules, the **Comparator** object was obtained by instantiating an object from a class of my own design. Those classes implemented the **Comparator** interface.

In this case, a reference to a **Comparator** object was returned by the call to the **reverseOrder** method of the **Collections** class, and that reference was passed as a parameter to the **sort** method.

5.4.14.5.3.4 Don't know, don't care

The **sort** method doesn't care where the **Comparator** object comes from, as long as it properly implements the **Comparator** interface.

Regardless of the source of the **Comparator** object, the **sort** method will use that object to impose the sorting rules imposed by the **compare** method of the object. In this case, the sorting rules cause the list to be sorted into *reverse natural order* .

5.4.14.5.3.5 The output

The code in Listing 7 (p. 1312) gets and uses an iterator to display the contents of the list following the call to the **sort** method in Listing 6 (p. 1311) .

Listing 7. Produce the output.

```
iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(iter.next() + " ");
}//end while loop
```

Figure 5.518: Listing 7. Produce the output.

The output produced by the code in Listing 7 (p. 1312) is shown below:

Tom TOM Joe JOE Bill BILL

You will recognize this as *reverse natural order* for the elements in the list.

5.4.14.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1307) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.14.7 Summary

In this module, I taught you how to use a **Comparator** created by the **reverseOrder** method of the **Collections** class to sort a list into *reverse natural order* . The **Comparator** object is generic, and can be used to sort any list of objects that implement the **Comparable** interface.

I also taught you how to use the **reverse** method of the **Collections** class to reverse the order of the elements in a list.

5.4.14.8 What's next?

In the next module, I am going to dig a little deeper into the implications of using the **toArray** method declared in the **Collection** interface.

5.4.14.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4140: The Comparator Interface, Part 6
- File: Java4140.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.15 Java4150: The toArray Method, Part 1[306]

5.4.15.1 Table of Contents

[306] This content is available online at <http://cnx.org/content/m46197/1.1/>.

5.4.15.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module shows you how to use the simpler version of the overloaded **toArray** method that is declared in the **Collection** interface. The module also explains why you need to exercise care when using the elements stored in the resulting array to avoid corrupting the state of the objects referred to by the elements in the collection.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [307] in Oracle's Java Tutorials [308] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.15.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.15.2.1.1 Listings

[307] http://docs.oracle.com/javase/tutorial/collections/index.html
[308] http://docs.oracle.com/javase/tutorial/index.html

5.4.15.3 Preview

In earlier modules, I used the **toArray** method, declared in the **Collection** interface, to copy elements from a collection into an array. However, in those modules, I didn't take the time to fully explain how to use the method. Also, I didn't fully explain the precautions that you need to take when you use the method.

The **Collection** interface declares the following two overloaded versions of the **toArray** method:

NOTE:

```
public Object[] toArray()

public Object[] toArray(Object[] a)
```

In this module, will teach you how to use the first *(simpler)* version of the **toArray** method. I will also show why you need to exercise care when using the elements stored in the array to avoid corrupting the state of the objects referred to by the elements in the collection.

I will teach you how to use the second *(more complex)* version of the **toArray** method in the next module.

5.4.15.4 Generics

The code in this series of modules is written with no thought given to Generics [309] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.15.5 Discussion and sample code

5.4.15.5.1 Beginning with a quiz

Let's begin with a quiz to test your prior knowledge of the Collections Framework. To take this quiz, examine the program shown in Listing 1 (p. 1316) and write down the output produced by the program.

[309] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named ToArray01.

```
    //File ToArray01.java
//Copyright 2001, R.G.Baldwin
import java.util.*;
import javax.swing.*;

public class ToArray01{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class ToArray01
//===================================//

class Worker{
  public void doIt(){
    Collection ref;

    //Create, populate, and display the
    // contents of a collection
    ref = new LinkedList();
    Populator.fillIt(ref);
    System.out.println("Collection contents");
    showCollection(ref);

    //Get collection contents into the
    // array and display the new
    // contents of the array.
    Object[] array = ref.toArray();
    System.out.println("New array contents");
    showArray(array);

    //Modify a property of an object
    // referred to by one of the
    // elements in the array. Display
    // array contents after
    // modification
    System.out.println("Modified array contents");
    ((JComponent)array[0]).setToolTipText("XX");
    showArray(array);

    //Display the contents of the
    // collection
    System.out.println("Collection contents");
    showCollection(ref);
  }//end doIt()
//-----------------------------------//

  //Utility method for displaying
  // array contents
  void showArray(Object[] array){
    for(int i = 0; i < array.length;i++){
      if(array[i] == null){
        System.out.print("null ");
      }else{
        System.out.print(((JComponent)array[i]).
                                  getToolTipText() + " ");
```

5.4.15.5.1.1 And the answer is ...

The correct answer to the quiz is the program output shown below:

NOTE:

```
Collection contents
B0 B1 L2 B3 B4 L5
New array contents
B0 B1 L2 B3 B4 L5
Modified array contents
XX B1 L2 B3 B4 L5
Collection contents
XX B1 L2 B3 B4 L5
```

If that was your answer, you probably already understand most of the material covered in this module. In that case, you might consider skipping this module and moving on to the next module. If that wasn't your answer, you should probably continue with your study of this module.

5.4.15.5.2 A new LinkedList collection

The code in Listing 2 (p. 1317) creates and populates a new **LinkedList** object and saves the object's reference as the interface type **Collection** . The collection is populated by passing the **LinkedList** object's reference to a method named **fillIt** .

The code in Listing 2 (p. 1317) also displays the contents of the **LinkedList** after it has been populated. The list is displayed by passing the **LinkedList** object's reference to a method named **showCollection** .

Listing 2. A new LinkedList collection.

```
    Collection ref;
ref = new LinkedList();

Populator.fillIt(ref);

System.out.println("Collection contents");
showCollection(ref);
```

Figure 5.520: Listing 2. A new LinkedList collection.

5.4.15.5.2.1 The LinkedList class

The **LinkedList** class is one of the concrete class implementations of the *Collections Framework* . This class implements the **Collection** interface and the **List** interface. Thus, it adheres to the contracts and stipulations of the **List** interface.

Here is part of what Oracle has to say about this class:

"Linked list implementation of the List interface. Implements all optional list operations, and permits all elements (including null). In addition ..."

5.4.15.5.2.2 Populating the LinkedList collection

The beginning of the static **fillIt** method, used to populate the collection, is shown in Listing 3 (p. 1318) .

Listing 3. Beginning of the fillIt method.

```
 public static void fillIt(Collection ref){
ref.add(new JButton());
ref.add(new JButton());
ref.add(new JLabel());
ref.add(new JButton());
ref.add(new JButton());
ref.add(new JLabel());
```

Figure 5.521: Listing 3. Beginning of the fillIt method.

As shown in Listing 3 (p. 1318) , the **fillIt** method begins by calling the **add** method six times in succession, passing references to new anonymous objects as a parameter to the **add** method.

5.4.15.5.2.3 Four buttons and two labels

Four of the objects are instantiated from the class named **JButton** . Two of the objects are instantiated from the class named **JLabel** .

Both **JButton** and **JLabel** belong to the **javax.swing** package. Further, both are subclasses of the class named **JComponent** .

5.4.15.5.2.4 The toolTipText property

Finally, both classes have a property named **toolTipText** , which can be set and accessed by calling the following methods on a reference to the object:

NOTE:

```
void setToolTipText(String text)

String getToolTipText()
```

5.4.15.5.2.5 Why am I using Swing GUI components?

I really don't plan to do anything special with these Swing GUI components. Rather, I chose to use them for illustration purposes simply because they possess the characteristics that I need for this module, and the next module as well. Those characteristics are:

- Both classes subclass the class named JComponent *(a common superclass below the Object class).*
- Both classes inherit a property *(toolTipText)* that can be used to identify them later.

5.4.15.5.3 Making the objects distinguishable

After the code in Listing 3 (p. 1318) has been executed, the buttons and labels are indistinguishable on the basis of the *null* value of their **toolTipText** property.

The code in Listing 4 (p. 1319) deals with this issue. This code uses the **setToolTipText** method to store a unique **String** value in the **toolTipText** property of the object referred to by each of the elements in the collection.

Listing 4. Making the objects distinguishable.

```
    Iterator iter = ref.iterator();
  int cnt = 0;
  JComponent refVar;

  while(iter.hasNext()){
    refVar = (JComponent)iter.next();
    if(refVar instanceof JButton){
      refVar.setToolTipText("B"+cnt++);
    }else{
      refVar.setToolTipText("L" + cnt++);
    }//end else
  }//end while loop

}//end fillIt()
```

Figure 5.522: Listing 4. Making the objects distinguishable.

5.4.15.5.3.1 Identifying the buttons and labels

In addition to storing a unique value in the **toolTipText** property of the object referred to by each element, the code in Listing 4 (p. 1319) also makes it possible to distinguish between the **JButton** objects and the **JLabel** objects. This is accomplished by including an upper-case "B" in the property value for each **JButton** , and including an upper-case "L" in the property value for each **JLabel** button.

5.4.15.5.3.2 Why populate this way?

This approach to population is, admittedly, a little bit of an overkill for illustrating what I want to illustrate in this program. However, I plan to use the same **fillIt** method in the sample program in the next module, and it won't be an overkill there.

5.4.15.5.3.3 Display the collection

The code in Listing 2 (p. 1317) above calls the **showCollection** method to display the contents of the populated **LinkedList** collection. The **showCollection** method is shown in Listing 5 (p. 1320)

Listing 5. The showCollection method.

```
  void showCollection(Collection ref){
Iterator iter = ref.iterator();
while(iter.hasNext()){
  System.out.print(((JComponent)iter.next()).
                            getToolTipText() + " ");
}//end while loop
System.out.println();
}//end showCollection
```

Figure 5.523: Listing 5. The showCollection method.

By now, you should have no difficulty understanding the code in Listing 5 (p. 1320) . This code gets an iterator on the incoming reference of type **Collection** . The code then uses that iterator to gain access to each element in succession, displaying the **String** value of the **toolTipText** property belonging to a particular object during each iteration.

5.4.15.5.3.4 Downcast is required

Note that the **next** method of the **Iterator** interface returns a reference to the next element in the collection, as type **Object** . *(Remember, Generics were not used to populate this collection.)*

In order to call the **getToolTipText** method on the returned reference, the reference must be downcast to type **JComponent** . Since both **JButton** and **JLabel** extend **JComponent** , and the **getToolTipText** method is declared in the **JComponent** class, it is not necessary to be concerned as to whether an object is type **JButton** or type **JLabel** to display the value of the **toolTipText** property. *(This is an example of polymorphic behavior based on class inheritance.)*

5.4.15.5.3.5 The output for the collection

The output produced by the code in Listing 2 (p. 1317) is shown below:

NOTE:

```
Collection contents
B0 B1 L2 B3 B4 L5
```

By examining the "B" and "L" characters in this output, you can identify the **JButton** objects and the **JLabel** objects.

5.4.15.5.4 Copy collection elements into an array

The code in Listing 6 (p. 1321) shows how to use the simple version of the **toArray** method to create an array of type **Object** that contains a copy of each element in the **LinkedList** collection.

Listing 6. Copy collection elements into an array.

```
Object[] array = ref.toArray();
```

Figure 5.524: Listing 6. Copy collection elements into an array.

5.4.15.5.4.1 The toArray method

Here is some of what Oracle has to say about this version of the **toArray** method:

> *"Returns an array containing all of the elements in this collection. If the collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.*
>
> *The returned array will be "safe" in that no references to it are maintained by this collection. ... The caller is thus free to modify the returned array."*

I will have some more to say about the *safe* aspects of the array shortly.

5.4.15.5.4.2 Display the array contents

The code in Listing 7 (p. 1321) calls a method named **showArray** to cause the current contents of the array to be displayed.

Listing 7. Display the array contents.

```
   System.out.println("New array contents");
showArray(array);
```

Figure 5.525: Listing 7. Display the array contents.

The entire **showArray** method is shown in Listing 8 (p. 1322) .

Listing 8. The showArray method.

```
  void showArray(Object[] array){
for(int i = 0; i < array.length;i++){
  if(array[i] == null){
    System.out.print("null ");
  }else{
    System.out.print(((JComponent)array[i]).
                              getToolTipText() + " ");
  }//end else
}//end for loop
System.out.println();
}//end showArray()
```

Figure 5.526: Listing 8. The showArray method.

5.4.15.5.4.3 The showArray method

The behavior of the **ShowArray** method is straightforward. The method uses a **for** loop to access each of the elements stored in the array in increasing index order.

A test is made to determine if the element contains a null reference. If so, then the word **null** is displayed for that element. If not, the **getToolTipText** method is used to access and display the value of the **toolTipText** property for each element in the array.

5.4.15.5.4.4 The output for the array

The output produced by the code in Listing 8 (p. 1322) is shown below:

NOTE:

```
New array contents
B0 B1 L2 B3 B4 L5
```

As you can see, *(except for the String that identifies the type of output)* this is an exact match to the output produced when the contents of the collection were displayed.

5.4.15.5.5 How "safe" is the array?

While it is "safe" to modify the contents of the array as explained in the quotation from Oracle earlier, there is still some danger here that you need to be aware of.

Java collections do not store objects. Rather, Java collections store references to objects. In Java, it is entirely possible to have two or more references to the same object.

5.4.15.5.5.1 Array contains copies of references to objects

Each element in the array is a copy of an element in the collection.

Therefore, at this point, for each object being managed by the collection, at least two references exist that refer to that object. One copy is contained in the collection. The other copy is contained in the array.

If you use a reference stored in the array to modify the state of one of those objects, that modification is made to the object that is also referenced by an element in the collection. This may or may not be what you intend. It's not necessarily a problem as long as you understand what is going on and be careful how you use the references stored in the array.

5.4.15.5.5.2 Modifying the state of an object

The code shown in Listing 9 (p. 1323) calls the **setToolTipText** method on the reference stored in the first element in the array to modify the state of the object to which that reference refers. Then the code calls the **showArray** method to display the contents of the array.

Listing 9. Modifying the state of an object.

```
    System.out.println("Modified array contents");
((JComponent)array[0]).setToolTipText("XX");
showArray(array);
```

Figure 5.527: Listing 9. Modifying the state of an object.

The **toolTipText** property value for each of the objects referred to by the remaining elements is left undisturbed.

5.4.15.5.5.3 Display array contents after object modification

The output produced by the code in Listing 9 (p. 1323) is shown below:

NOTE:

```
Modified array contents
XX B1 L2 B3 B4 L5
```

As you can see, except for the first element, this is a match for the display of the array contents before the state of the object referred by the first element was modified. However, the **toolTipText** property for the object referred to by the first element now contains the string "XX", instead of the string "B0" as before.

5.4.15.5.5.4 Display the contents of the collection again

The code in Listing 10 (p. 1324) displays the state of each of the objects referred to by the elements in the **LinkedList** collection.

Listing 10. Display the contents of the collection again.

```
   System.out.println("Collection contents");
showCollection(ref);
```

Figure 5.528: Listing 10. Display the contents of the collection again.

The output produced by Listing 10 (p. 1324) is shown below:

NOTE:

```
Collection contents
XX B1 L2 B3 B4 L5
```

As you can see, the state of the object referred to by the reference stored in the first element of the collection is also changed. The **toolTipText** property for that object now contains the string "XX" instead of "B0" as before.

5.4.15.5.6 The bottom line

It is safe to modify the contents of the array, even to replace the references in the array with references to other objects. Such a replacement has no impact on the contents of the collection.

However, it is also possible to use the elements of the array to modify the state of the objects referred to by the elements in the collection.

If this is what you intend to do, that's great. However, if that is not what you intend to do, that may be a problem. So, the bottom line is, be careful what you do with the elements in the array.

5.4.15.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1316) . Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.15.7 Summary

In this module, I taught you how to use the simpler version of the overloaded **toArray** method, declared in the **Collection** interface, to copy the elements from a collection into an array of type **Object** .

I also showed why you need to exercise care when using the elements stored in the array, to avoid corrupting the state of the objects referred to by the elements in the collection.

5.4.15.8 What's next?

In the next module, I will teach you how to use the other, more complex version of the overloaded **toArray** method.

5.4.15.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4150: The toArray Method, Part 1
- File: Java4150.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.4.16 Java4160: The toArray Method, Part 2[310]

5.4.16.1 Table of Contents

[310] This content is available online at <http://cnx.org/content/m46198/1.1/>.

5.4.16.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) in general and the Java Collections framework in particular.

This module shows you how to use the more-complex version of the **toArray** method declared in the **Collection** interface. The module discusses issues regarding the type of the array and the types of the objects referred to by the elements in the collection. The module also discusses issues regarding the relative sizes of the array and the collection. Finally, the module reaffirms that you need to exercise caution when using the elements stored in the array, to avoid corrupting the state of the objects referred to by the elements in the collection.

In addition to studying these modules, I strongly recommend that you study the Collections Trail [311] in Oracle's Java Tutorials [312] . The modules in this collection are intended to supplement and not to replace those tutorials.

5.4.16.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

5.4.16.2.1.1 Listings

- Listing 1 (p. 1328) . The program named ToArray02.
- Listing 2 (p. 1329) . A populated array.

[311] http://docs.oracle.com/javase/tutorial/collections/index.html
[312] http://docs.oracle.com/javase/tutorial/index.html

- Listing 3 (p. 1330) . A new LinkedList collection.
- Listing 4 (p. 1331) . The fillIt method.
- Listing 5 (p. 1332) . Copy collection elements into an array.
- Listing 6 (p. 1336) . Modify an object .

5.4.16.3 Preview

The **Collection** interface declares the following two overloaded versions of the **toArray** method:

NOTE:

```
public Object[] toArray()

public Object[] toArray(Object[] a)
```

In the previous module, I taught you how to use the first *(simpler)* of the two methods. I also discussed the need to exercise care when using the elements stored in the returned array to avoid corrupting the state of the objects referred to by elements in the collection.

In this module, I will teach you how to use the second *(more-complex)* version of the **toArray** method declared in the **Collection** interface. I will discuss issues regarding the type of the array and the types of the objects referred to by the elements in the collection. I will also discuss issues regarding the relative sizes of the array and the collection.

Finally, I will reaffirm that you need to exercise care when using the elements stored in the array, to avoid corrupting the state of the objects referred to by the elements in the collection.

5.4.16.4 Generics

The code in this series of modules is written with no thought given to Generics [313] . As a result, if you copy and compile the code, you will probably get warnings about *unchecked or unsafe operations* .

While you will ultimately need to understand how to use Generics, that is a very complex topic. An understanding of Generics is beyond the scope of this course. Therefore, for purposes of this course, you can simply ignore those warnings.

5.4.16.5 Discussion and sample code

5.4.16.5.1 Beginning with a quiz

As has been the case in the last few modules, let's begin with a quiz to test your prior knowledge of the Collections Framework. To take this quiz, examine the program shown in Listing 1 (p. 1328) and write down the output produced by that program.

[313] http://docs.oracle.com/javase/tutorial/java/generics/index.html

Listing 1. The program named ToArray02.

```
    //File ToArray02.java
//Copyright 2001, R.G.Baldwin
import java.util.*;
import javax.swing.*;

public class ToArray02{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class ToArray02
//===================================//

class Worker{
  public void doIt(){
    Collection ref;

    //Create, populate, and display
    // the contents of an array
    JComponent[] array = new JComponent[8];
    for(int cnt=0;cnt<8;cnt++){
      array[cnt] = new JButton();
      array[cnt].setToolTipText("" + (cnt+10));
    }//end for loop
    System.out.println();
    showArray(array,"Original array contents");


    //Create, populate, and display the
    // contents of a collection
    ref = new LinkedList();
    Populator.fillIt(ref);
    showCollection(ref,"Collection contents");

    //Get collection contents into the
    // array and display the new
    // contents of the array.
    array = (JComponent[])ref.toArray(array);
    showArray(array,"New array contents");

    //Modify a property of an object
    // referred to by one of the
    // elements in the array. Display
    // array contents after
    // modification
    ((JComponent)array[0]).setToolTipText("XX");
    showArray(array,"Modified array contents");

    //Display the contents of the collection
    showCollection(ref,"Collection contents");
  }//end doIt()
//-----------------Available for free at Connexions <http://cnx.org/content/col11441/1.121>

  //Utility method for displaying
  // array contents
  void showArray(Object[] array,String title){
```

5.4.16.5.1.1 And the answer is ...

The correct answer to the quiz is the program output shown below:

NOTE:

```
Original array contents
10 11 12 13 14 15 16 17
Collection contents
B0 B1 L2 B3 B4 L5
New array contents
B0 B1 L2 B3 B4 L5 null 17
Modified array contents
XX B1 L2 B3 B4 L5 null 17
Collection contents
XX B1 L2 B3 B4 L5
```

If that was your answer, you probably already understand most of the material covered in this module. In that case, you might consider skipping this module and moving on to some more productive activity. If that wasn't your answer, you should probably continue with your study of this module.

5.4.16.5.1.2 Similar to previous program

Except for the use of a different version of the **toArray** method, the overall structure of the program in Listing 1 (p. 1328) is similar to the program in the previous module. Therefore, I will concentrate on those aspects of this program that differentiate it from the program in the previous module.

5.4.16.5.1.3 A populated array

Unlike the program in the previous module, the code in Listing 2 (p. 1329) creates and populates an eight-element array of type **JComponent** . This is the array that will be re-populated by the **toArray** method later in the program. The array is populated with a set of initial element values at this point to make it obvious when it is re-populated *(overwritten elements)* by the **toArray** method later.

Listing 2. A populated array.

```
    JComponent[] array = new JComponent[8];

for(int cnt=0;cnt<8;cnt++){
  array[cnt] = new JButton();
  array[cnt].setToolTipText("" + (cnt+10));
}//end for loop
System.out.println();
showArray(array,"Original array contents");
```

Figure 5.530: Listing 2. A populated array.

The **JButton** class, the **JLabel** class, and the **setToolTipText** method were discussed in detail in the previous module, so I won't repeat that discussion here.

5.4.16.5.1.4 Display the array contents

After the array is populated by the code in Listing 2 (p. 1329) , a reference to the array object is passed to the **showArray** method *(also in Listing 2 (p. 1329))* to display the contents of the array.

With the exception of some minor changes implemented in this program to make the use of the **showArray** method more compact, this is the same **showArray** method used in the previous module. Therefore, I won't discuss that method further in this module. The output produced by the code in Listing 2 (p. 1329) is as follows:

NOTE:

```
Original array contents
10 11 12 13 14 15 16 17
```

As you can see, each of the eight elements in the array was initialized with an easily-recognizable and unique value, (*which may be overwritten by the* ***toArray*** *method later).*

5.4.16.5.1.5 A new LinkedList collection

The code in Listing 3 (p. 1330) creates and populates a new **LinkedList** collection. The collection is populated by passing the **LinkedList** object's reference to a method named **fillIt** .

The code in Listing 2 (p. 1329) also displays the contents of the **LinkedList** collection after it has been populated. The list is displayed by passing the **LinkedList** object's reference to a method named **showCollection** .

Listing 3. A new LinkedList collection.

```
    ref = new LinkedList();
Populator.fillIt(ref);
showCollection(ref,"Collection contents");
```

Figure 5.531: Listing 3. A new LinkedList collection.

Except for a couple of minor changes to the **showCollection** method, the code to create, populate, and display the collection is the same as the code in the previous module.

5.4.16.5.1.6 Populating the LinkedList collection

A couple of points regarding the **fillIt** method *(shown in Listing 4 (p. 1331))* are worthy of note.

Listing 4. The fillIt method.

```
  public static void fillIt(Collection ref){
ref.add(new JButton());
ref.add(new JButton());
ref.add(new JLabel());
ref.add(new JButton());
ref.add(new JButton());
ref.add(new JLabel());

Iterator iter = ref.iterator();
int cnt = 0;
JComponent refVar;
while(iter.hasNext()){
  refVar = (JComponent)iter.next();
  if(refVar instanceof JButton){
    refVar.setToolTipText("B"+cnt++);
  }else{
    refVar.setToolTipText("L" + cnt++);
  }//end else
}//end while loop

}//end fillIt()
```

Figure 5.532: Listing 4. The fillIt method.

The **fillIt** method begins by calling the **add** method six times in succession, passing references to new anonymous objects *(of types JButton and JLabel)* as a parameter to the **add** method.

5.4.16.5.1.7 Four buttons and two labels

Four of the objects are instantiated from the class named **JButton** . The other two objects are instantiated from the class named **JLabel** .

Both **JButton** and **JLabel** belong to the **javax.swing** package. Further, both are subclasses of the class named **JComponent** .

5.4.16.5.1.8 The toolTipText property

Finally, both classes have a property named **toolTipText** , which can be set and accessed by calling the following methods on a reference to the object:

NOTE:

```
void setToolTipText(String text)

String getToolTipText()
```

5.4.16.5.1.9 JButton and JLabel

I chose to use objects of these two classes for illustration purposes simply because they possess the characteristics that I need for this module. Those characteristics are:

- Both classes subclass the class named JComponent *(a common superclass below the Object class)*.
- Both classes inherit a property *(toolTipText)* that can be used to identify them later.

5.4.16.5.1.10 Making the objects distinguishable

After adding the objects' references to the collection, the code in Listing 4 (p. 1331) uses the **setToolTipText** method to store a unique **String** value in the **toolTipText** property of the object referred to by each of the elements in the collection.

5.4.16.5.1.11 Identifying the buttons and labels

In addition to storing a unique value in the **toolTipText** property of the object referred to by each element, the code in Listing 4 (p. 1331) also makes it possible to distinguish between the **JButton** objects and the **JLabel** objects. This is accomplished by including an upper-case "B" in the property value for each **JButton** , and including an upper-case "L" in the property value for each **JLabel** button.

5.4.16.5.1.12 Display the collection

The code in Listing 3 (p. 1330) above calls the **showCollection** method to display the contents of the populated **LinkedList** collection. The output produced by the code in Listing 3 (p. 1330) is shown below:

NOTE:

```
Collection contents
B0 B1 L2 B3 B4 L5
```

Each term in the output is the **String** value of the **toolTipText** property for a particular object. Hence, there are six terms in the output, one for each element in the collection.

5.4.16.5.1.13 Copy collection elements into an array

Having completed the preliminaries, we have now reached the point that is the main thrust of this module.

The code in Listing 5 (p. 1332) shows how to use the more-complex version of the **toArray** method to copy the elements in the collection into an array.

Listing 5. Copy collection elements into an array.

```
    array = (JComponent[])ref.toArray(array);
showArray(array,"New array contents");
```

Figure 5.533: Listing 5. Copy collection elements into an array.

The code in Listing 5 (p. 1332) also causes the contents of the array to be displayed after it receives the elements from the collection.

The first statement in Listing 5 (p. 1332) causes the first seven elements in the array to be overwritten with element values from the collection *(plus one null value)*.

The second statement in Listing 5 (p. 1332) causes the contents of the array to be displayed.

5.4.16.5.2 The toArray method

The most important thing to note about Listing 5 (p. 1332) is that a reference to an array object is passed as a parameter to the **toArray** method. *(The simpler version of the toArray method, discussed in the previous module, doesn't take any parameters.)*

5.4.16.5.2.1 The essential difference

The essential difference between the two overloaded versions of the **toArray** method has to do with the origin of the array into which the **toArray** method copies the elements from the collection.

With the simpler version of the **toArray** method that takes no parameters, the **toArray** method creates a new array object of type **Object**, populates it, and returns that object's reference as type **Object**.

5.4.16.5.2.2 Type is not an issue for the simpler version

Since the new array object is of type **Object**, *(when the rules for Generics are not adhered to)* there are no issues regarding type compatibility between the type of the array and the types of the elements stored in the collection. A reference to an object of any type can be stored in an array of the generic type **Object[]**.

5.4.16.5.2.3 Size is not an issue for the simpler version

Also, since the array is created when it is needed by the simpler version of the **toArray** method, there are also no size issues. The array is created to be of the correct size to contain copies of all of the elements in the collection.

5.4.16.5.3 More-complex version presents some issues

With the more-complex version of the **toArray** method *(shown in Listing 5 (p. 1332))*, the programmer must provide the array object that will be populated by the **toArray** method. In this situation, there are size issues as well as type issues to be dealt with.

5.4.16.5.3.1 The type issue

Here is some of what the Oracle documentation for the **LinkedList** class has to say about the type issue for this version of the **toArray** method:

> *"Returns an array containing all of the elements in this list in the correct order. The runtime type of the returned array is that of the specified array. ... Throws: ArrayStoreException - if the runtime type of (the specified array) is not a supertype of the runtime type of every element in this list.*

In other words, the type of the array passed as a parameter to the **toArray** method must be a superclass of the classes from which all of the objects being managed by the collection were instantiated.

5.4.16.5.3.2 Two types of objects in this collection

In this program, the collection is managing objects of the types **JButton** and **JLabel** . Each of these types is a subclass of the class named **JComponent** . For that reason, the type of array that I instantiated and passed to the **toArray** method is **JComponent[]** .

5.4.16.5.3.3 The size issue

Here is some of what the Oracle documentation for the **LinkedList** class has to say about the size issue for this version of the **toArray** method.

> *"If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list. If the list fits in the specified array with room to spare (i.e., the array has more elements than the list), the element in the array immediately following the end of the collection is set to null. This is useful in determining the length of the list only if the caller knows that the list does not contain any null elements."*

5.4.16.5.3.4 So, what did I do?

Knowing all of this in advance, I purposely caused the size of the **JComponent** array to be larger *(by two elements)* than the number of elements in the collection. Therefore, the array that I passed to the **toArray** method was populated and a reference to that populated array was returned.

> *(Had my array been smaller than the number of elements in the collection, the toArray method would have created and populated a new array of type JComponent and would have returned a reference to that new array object. In that case, my array would have been used by the toArray method only for the purpose of determining the runtime type of my array.)*

5.4.16.5.3.5 More information from Oracle

Here is some additional information about the **toArray** method provided by the Oracle documentation for the **Collection** interface:

> *"If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order."*

Because the iterator for a **LinkedList** object returns the elements in increasing index order, the **toArray** method, in this case, copies the element at each index position in the collection into the element at the same index position in the array. Thus, reference values are copied from each element in the collection into the first six elements in the array.

5.4.16.5.4 The output

The output produced by the code in Listing 5 (p. 1332) is shown below:

NOTE:

```
New array contents
B0 B1 L2 B3 B4 L5 null 17
```

You will note that the first six elements in the array match the six elements in the collection *(the initial values placed in the array earlier when the array was instantiated have been overwritten).*

You will also note that the value of the seventh element in the array *(index value 6)* has been overwritten with a null reference.

5.4.16.5.4.1 Demonstrates same array was used

Note finally that the last element in the array was not overwritten. It still contains the value placed there when the array object was instantiated. This demonstrates that the array that I passed to the **toArray** method was populated with the collection data, and a reference to that array was returned by the **toArray** method.

5.4.16.5.4.2 What if the array was too small?

Had my array been too small, it would have been discarded by the **toArray** method. The **toArray** method would have created and populated a new array object of the correct size and runtime type, and would have returned a reference to that new array.

5.4.16.5.4.3 Not difficult to demonstrate

Although this is not demonstrated by this program, it is easy to modify the program to demonstrate this feature.

A **String** representation of the array object can be displayed using a **System.out.println(array)** statement before and after the array is passed to the **toArray** method.

5.4.16.5.4.4 Array as large as or larger than collection

For the cases where my array contained six, seven, or eight elements, and the collection contained six elements, the **String** representations of the array object before and after the call to the **toArray** method were the same. For one case, those **String** representations were as follows:

NOTE:

```
[Ljavax.swing.JComponent;@49ba38
[Ljavax.swing.JComponent;@49ba38
```

In other words, the reference variable named **array** referred to the same array object before and after the call to the **toArray** method.

5.4.16.5.4.5 Array smaller than the collection

When I reduced the size of the array to five elements, keeping the size of the collection at six elements, the before and after **String** representations of the array object were as follows:

NOTE:

```
[Ljavax.swing.JComponent;@506411
[Ljavax.swing.JComponent;@21807c
```

In this case, the reference to the array object returned by the **toArray** method was different from the reference that was passed to the **toArray** method. In other words, the returned reference referred to a different array object than was referred to by the reference that was passed to the **toArray** method.

5.4.16.5.5 Modify an object

As in the program in the previous module, the code shown in Listing 6 (p. 1336) modifies the value of the **toolTipText** property of the object whose reference is stored in index 0 of the array.

Listing 6. Modify an object .

```
    ((JComponent)array[0]).setToolTipText("XX");
showArray(array,"Modified array contents");

showCollection(ref,"Collection contents");
```

Figure 5.534: Listing 6. Modify an object .

The code in Listing 6 (p. 1336) also displays the contents of the array and the contents of the collection after the modification is made.

The output produced by the code in Listing 6 (p. 1336) is shown below:

NOTE:

```
Modified array contents
XX B1 L2 B3 B4 L5 null 17
Collection contents
XX B1 L2 B3 B4 L5
```

5.4.16.5.5.1 Now for the caution

Note that the value of the **toolTipText** property of the object referred to by the reference at index 0 of the array, and the same property of the object referred to by the reference at index 0 of the collection was overwritten by "XX". *(This is true because both references refer to the same object.)*

This is the case regardless of which version of the **toArray** method is used. Therefore, the same cautions discussed in the previous module apply here as well.

5.4.16.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1328) , Paste the code into your Java editor. Then compile and execute it.

Run the program and observe the results. Experiment with the code. Make changes, run the program again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

5.4.16.7 Summary

In this module, I taught you how to use the more-complex version of the two overloaded versions of the **toArray** method, declared in the **Collection** interface, to copy the elements from a collection into an array of type **JComponent** .

I discussed issues regarding the type of the array and the type of the objects referred to by the elements in the container. I also discussed issues regarding the size of the array as compared to the number of elements in the collection.

Finally, I reaffirmed that you need to exercise care when using the elements stored in the array, to avoid corrupting the state of the objects referred to by the elements in the collection.

5.4.16.8 What's next?

For now, at least, this module concludes the series of modules on the Java Collections Framework. If I have time later, I will come back and add more modules to teach you how to use the **Map** and **SortedMap** interfaces, and the concrete class implementations of those interfaces.

5.4.16.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java4160: The toArray Method, Part 2
- File: Java4160.htm
- Published: 05/07/13

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.5 Practice Programs

5.5.1 Java OOP: ITSE 2321 Practice Group 1[314]

5.5.1.1 ITSE 2321 Object-Oriented Programming - Practice Group 1

5.5.1.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Oracle's Standard Edition JDK Version 1.7 or later.

Some of the programs in this group require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [315] .

5.5.1.1.2 Input Image Files

Links are provided within the individual program specifications for downloading any image files that may be required to write, compile, and test your programs.

5.5.1.1.3 Solution source code files

Links are provided within the individual program specifications for downloading source code files that contain the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [316] .

5.5.1.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

5.5.1.1.5 New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX.

[314] This content is available online at <http://cnx.org/content/m44252/1.7/>.
[315] http://cnx.org/content/m44148/latest/
[316] http://cnx.org/content/m44148/latest/

5.5.1.1.6 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click on the input image file link(s) and use the capabilities of your browser to download and save the image file(s).
2. Click on the Java solution source code link(s) and use the capabilities of your browser to download and save the source code file(s).
3. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
4. Write, compile, and execute a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

In addition to the hints listed above, I will precede the detailed specifications for each program with a discussion that contains hints about the concepts and skills that you will probably need to successfully write the program.

In order to write this or any other Java program of substance, you will need to know how to use [317] the Java Platform, Standard Edition API Specification [318] as well as the documentation for the Guzdial-Ericson Multimedia Class Library [319] .

You may find other useful hints in my online tutorials and slides for this course as well as in the YouTube video lectures for this course.

5.5.1.1.7 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [320] .

5.5.1.1.8 Program Specifications

5.5.1.1.8.1 Program 1

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob01.java [321] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library.

- How to design and implement an algorithm that will transform Prob01.jpg [322] into the image shown in Figure 1 (p. 1343) .
- How to incorporate and use an external class library [323] in addition to the standard Java class library.

[317] http://cnx.org/content/m45117/latest/
[318] http://docs.oracle.com/javase/7/docs/api/index.html
[319] http://cnx.org/content/m44148/latest/#Discussion_and_sample_code
[320] http://cnx.org/content/m44148/latest/
[321] http://cnx.org/content/m44252/latest/Prob01.java
[322] http://cnx.org/content/m44252/latest/Prob01.jpg
[323] http://cnx.org/content/m44148/latest/

- The general syntax [324] for a Java application *(not a Java applet).*
- How to write [325] , compile, and execute [326] a Java application that uses the Guzdial-Ericson Multimedia Class Library.
- How to use the standard print and println [327] methods of the **System** class to display text on the command-line screen.
- The effect of overridden [328] versions of the toString [329] method on the **print** and **println** methods
- Knowledge of the need for and use of import directives [330] .
- How to define [331] and instantiate an object [332] of a new class named **Prob01Runner** .
- Knowledge of the difference between local variables [333] and instance variables [334] .
- How to declare local reference variables [335] of a given type.
- How to save the **Prob01Runner** object's reference in a reference variable [336] of type **Prob01Runner** named **obj** .
- How to call [337] the **run** method, *(which controls the major behavior of the program)* , on the **Prob01Runner** object's reference.
- How to use private [338] , public [339] , protected [340] , and package-private [341] access modifiers.
- How to declare and use reference variables [342] versus primitive variables [343] .
- How to select the appropriate overloaded constructors [344] in order to i nstantiate objects [345] of Ericson's **World** , **Turtle** , and **Picture** classes.
- How to define a constructor [346] for a class.
- How to write accessor methods [347] in your new class definition that return references to **Turtle** and **World** objects.
- How and why to call accessor methods [348] on objects.
- How to instantiate anonymous objects [349] .
- How to replace [350] the default white **Picture** object encapsulated in a **World** object with a new and different **Picture** object encapsulating the contents of a local image file.
- How to call a method on a **Picture** object to write text [351] onto the **Picture** object.
- How to call a variety of methods [352] on objects of the **Turtle** class to cause the **Turtle** objects in your world to do things, such as move forward, change their colors, etc.

[324] http://cnx.org/content/m44150/latest/#Listing_9
[325] http://cnx.org/content/m44148/latest/#Listing_2
[326] http://cnx.org/content/m44148/latest/#Figure_4
[327] http://cnx.org/content/m44150/latest/#Listing_5
[328] http://cnx.org/content/m44190/latest/
[329] http://cnx.org/content/m44149/latest/
[330] http://cnx.org/content/m44149/latest/#Listing_1
[331] http://cnx.org/content/m44149/latest/#Listing_2
[332] http://cnx.org/content/m44149/latest/#Listing_1
[333] http://cnx.org/content/m44204/latest/
[334] http://cnx.org/content/m44150/latest/
[335] http://cnx.org/content/m44153/latest/
[336] http://cnx.org/content/m44150/latest/#Listing_2
[337] http://cnx.org/content/m44149/latest/#Listing_1
[338] http://cnx.org/content/m44149/
[339] http://cnx.org/content/m44153/latest/
[340] http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
[341] http://cnx.org/content/m44149/
[342] http://cnx.org/content/m44149/latest/
[343] http://cnx.org/content/m44149/latest/
[344] http://cnx.org/content/m44149/latest/#Figure_4
[345] http://cnx.org/content/m44149/latest/#Listing_2
[346] http://cnx.org/content/m44193/latest/
[347] http://cnx.org/content/m44149/latest/#Listing_4
[348] http://cnx.org/content/m44149/latest/
[349] http://cnx.org/content/m44206/latest/
[350] http://cnx.org/content/m44149/latest/#Listing_5
[351] http://cnx.org/content/m44149/latest/#Listing_6
[352] http://cnx.org/content/m44149/latest/#Listing_7

- How pictures are composed of images.
- How images are composed of pixels.
- How pixels are composed of red, green, and blue color values.
- How objects can be used to represent pictures, images, and pixels.
- How methods can be called on those objects to manipulate the red, green, and blue color values.

Listing 5.1:

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image file named Prob01.jpg [353] to produce the graphic output image shown in Figure 1 (p. 1343) below.

Click Prob01.java [354] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other three lines of text shown below on the command-line screen:

```
    Display your name here.
A 300 by 274 world with 2 turtles in it.
joe turtle at 256, 131 heading 45.0.
sue turtle at 50, 37 heading 0.0.
*******************************************************/
public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob01Runner obj = new Prob01Runner();
    obj.run();

    System.out.println(obj.getMars());
    System.out.println(obj.getJoe());
    System.out.println(obj.getSue());
  }//end main
}//end class Prob01nd program specifications.
```

[353] http://cnx.org/content/m44252/latest/Prob01.jpg
[354] http://cnx.org/content/m44252/latest/Prob01.java

Required output image for Prob01.

Figure 5.535: Required output image for Prob01.

5.5.1.1.8.2 Program 2

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob02.java [355] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items listed under Program 1 (p. 1340) above. Instead, I will concentrate on new concepts, knowledge, and skills not included in the above list.

- How to design and implement an algorithm that will transform Prob02.jpg [356] into the image shown in Figure 2 (p. 1345) .
- How to declare and initialize [357] instance variables in a single statement.

[355] http://cnx.org/content/m44252/latest/Prob02.java
[356] http://cnx.org/content/m44252/latest/Prob02.jpg
[357] http://cnx.org/content/m44203/latest/#Listing_2

- How to instantiate a new object [358] of Ericson's **Picture** class encapsulating the contents of a local image file.
- How to call methods [359] on the new **Picture** object for a variety of purposes.
- How to declare a reference variable [360] capable of storing a reference to a one-dimensional array object.

- How to populate a one-dimensional array object [361] of type **Pixel[]** with references to all of the **Pixel** objects encapsulated in a **Picture** object.
- How to use a l oop structure [362] to individually access the reference to each **Pixel** object in the array object.
- How to use a reference to a **Pixel** object to modify [363] the red, green, and blue color values belonging to the pixel that is represented by the **Pixel** object.
- How use a pair of **PictureExplorer** objects to design and implement an algorithm that will transform the original image [364] into the required output image shown in Figure 2 (p. 1345) .
- How to display [365] your modified **Picture** object in an object of Ericson's **PictureExplorer** class.

- How to confirm the validity of your algorithm by numerically comparing the color values in your output with the color values produced by compiling and running the program solution [366] given below.

Listing 5.2: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses the class definition shown below and Ericson's media library along with the image file named Prob02.jpg [367] to produce the graphic output image shown in Figure 2 (p. 1345) below.

Click Prob02.java [368] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename Prob02.jpg height 274 width 365
*********************************************************/
public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob02Runner obj = new Prob02Runner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Prob02
//End program specifications.
```

[358] http://cnx.org/content/m44203/latest/#Listing_2
[359] http://cnx.org/content/m44203/latest/#Listing_3
[360] http://cnx.org/content/m44203/latest/#Listing_4
[361] http://cnx.org/content/m44203/latest/#Listing_4
[362] http://cnx.org/content/m44203/latest/#Listing_4
[363] http://cnx.org/content/m44203/latest/#Listing_4
[364] http://cnx.org/content/m44252/latest/Prob02.jpg
[365] http://cnx.org/content/m44203/latest/#Listing_5
[366] http://cnx.org/content/m44252/latest/Prob02.java
[367] http://cnx.org/content/m44252/latest/Prob02.jpg
[368] http://cnx.org/content/m44252/latest/Prob02.java

Required output image for Prob02.

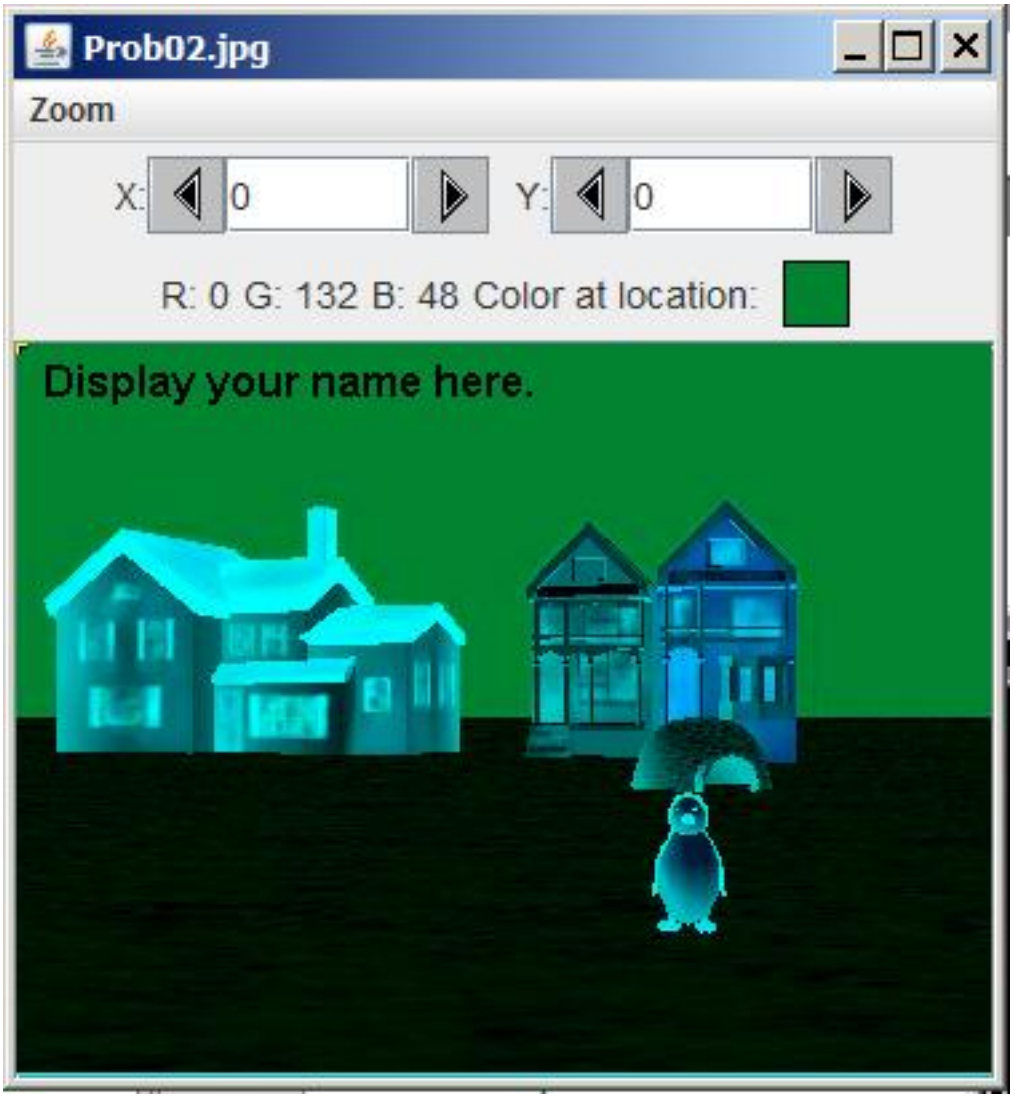


Figure 5.536: Required output image for Prob02.

5.5.1.1.8.3 Program 3

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this programand/or understand the given solution in Prob03.java [369] . Some of these items are general in nature and some are specific to the use

[369] http://cnx.org/content/m44252/latest/Prob03.java

of Ericson's multimedia library. I won't repeat the items listed above. Instead, I will concentrate on new concepts, knowledge, and skills not included in the above lists.

There are at least two alternative ways to write a program that will satisfy these requirements.

Both alternatives

- How to design and implement an algorithm that will transform Prob03.jpg [370] into the required output image shown in Figure 3 (p. 1347) .

Alternative 1

- How to create a new one-dimensional array object [371] of type **Pixel[]** and populate it with references to all of the **Pixel** objects encapsulated in a **Picture** object.
- How to use some complicated indexing arithmetic [372] in conjunction with the one-dimensional array mentioned above to apply the required algorithm.

Alternative 2

- How to use a nested loop structure [373] to achieve the same result.
- How to use a cast operator [374] .

Listing 5.3: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses the class definition shown below and Ericson's media library along with the image file named Prob03.jpg [375] to produce the graphic output image shown in Figure 3 (p. 1347) below.

Click Prob03.java [376] to download a Java source file containing the solution to this program.

In addition to the output, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename Prob03.jpg height 274 width 365

*******************************************************/
public class Prob03{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
    System.out.println(obj.getPicture());
  }//end main
}//end class Prob03
//End program specifications.
```

[370] http://cnx.org/content/m44252/latest/Prob03.jpg
[371] http://cnx.org/content/m44204/latest/#Listing_3
[372] http://cnx.org/content/m44204/latest/#Listing_5
[373] http://cnx.org/content/m44207/latest/#Listing_5
[374] http://cnx.org/content/m44168/
[375] http://cnx.org/content/m44252/latest/Prob03.jpg
[376] http://cnx.org/content/m44252/latest/Prob03.java

Required output image for Prob03.

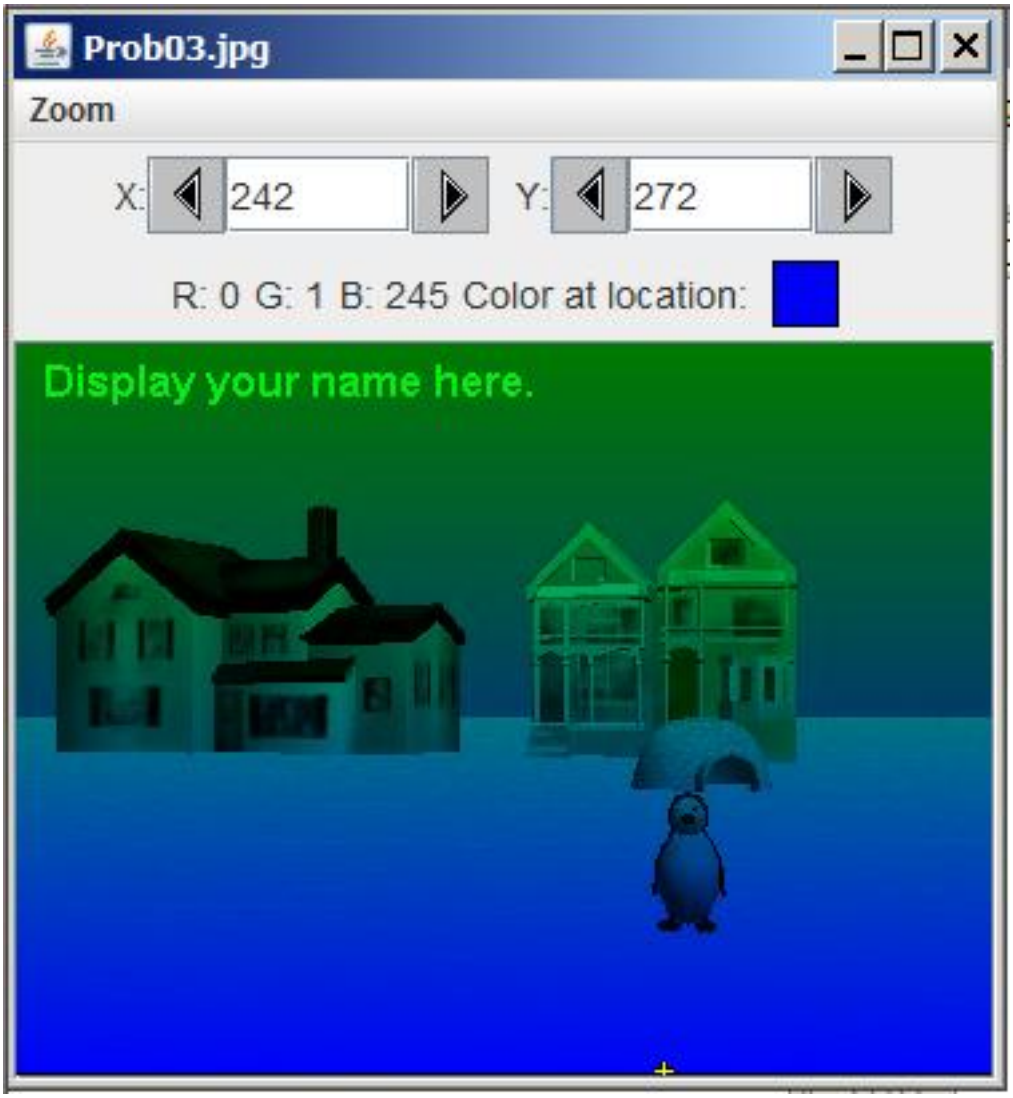


Figure 5.537: Required output image for Prob03.

5.5.1.1.8.4 Program 4

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this programand/or understand the given solution in Prob04.java [377] . Some of these items are general in nature and some are specific to the use

[377] http://cnx.org/content/m44252/latest/Prob04.java

of Ericson's multimedia library. I won't repeat the items listed above. Instead, I will concentrate on new concepts, knowledge, and skills not included in the above lists.

- How to design and implement an algorithm that will transform Prob04.jpg [378] into the image shown in Figure 4 (p. 1349) .
- Similarities and differences between classes and interfaces [379] .
- The different types under which you can store an object's reference when the class from which the object was instantiated extends a class and implements one or more interfaces.
- The kinds of new relationships that are created when a class implements one or more interfaces.
- The implications of inheriting one or more abstract methods.
- The significance of all interface methods being implicitly abstract.
- That **DigitalPicture** is an interface [380] and is not a class in Ericson's library.
- The relationship that exists between the **DigitalPicture** interface and the **Picture** class.
- The circumstances under which an accessor method can return a reference to an object as type **DigitalPicture** .
- The difference between displaying a **Picture** object with the **show** and **explore** methods.

Listing 5.4: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses the class definition shown below and Ericson's media library along with the image file named Prob04.jpg [381] to produce the graphic output image shown in Figure 4 (p. 1349) below.

Click Prob04.java [382] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename Prob04.jpg height 274 width 365
*******************************************************/
public class Prob04{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    DigitalPicture digitalPicture =
                                obj.getDigitalPicture();
    System.out.println(digitalPicture);
    digitalPicture.show();
  }//end main
}//end class Prob04
//End program specifications.
```

[378] http://cnx.org/content/m44252/latest/Prob04.jpg
[379] http://cnx.org/content/m44195/latest/
[380] http://cnx.org/content/m44195/latest/?collection=col11441/latest
[381] http://cnx.org/content/m44252/latest/Prob04.jpg
[382] http://cnx.org/content/m44252/latest/Prob04.java

Required output image for Prob04.

Figure 5.538: Required output image for Prob04.

5.5.1.1.8.5 Program 5

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob05.java [383] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items listed above. Instead, I will concentrate on new concepts, knowledge, and skills not included in the above lists.

- How to design and implement an algorithm that will transform Prob05a.jpg [384] and Prob05b.jpg [385] into the images shown in Figure 5 (p. 1351) and Figure 6 (p. 1352) .
- What it means for the method named **getDigitalPictures** to return a reference to an object as type DigitalPicture[] [386] .

[383] http://cnx.org/content/m44252/latest/Prob05.java
[384] http://cnx.org/content/m44252/latest/Prob05a.jpg
[385] http://cnx.org/content/m44252/latest/Prob05b.jpg
[386] http://cnx.org/content/m44198/latest/#Listing_1

- What you can do with a reference of type **DigitalPicture[]** .
- The implications of the fact that calling the **show** method each of the elements of the array of type **DigitalPicture[]** produces a different output image.

Listing 5.5: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses the class definition shown below and Ericson's media library along with the image files named Prob05a.jpg [387] and Prob05b.jpg [388] to produce the pair of graphic output images shown in Figure 5 (p. 1351) and Figure 6 (p. 1352) below.

Note that unless you know how to position the output images on the screen, they will both end up in the upper-left corner of the screen with one image partially or completely covering the other image. Use your mouse to drag and separate the images.

Click Prob05.java [389] to download a Java source file containing the solution to this program.

In addition to the output images mentioned above, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename Prob05b.jpg height 309 width 412
Picture, filename Prob05a.jpg height 274 width 365
**********************************************************/
public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob05Runner obj = new Prob05Runner();
    DigitalPicture[] digitalPictures =
                                obj.getDigitalPictures();

    System.out.println(digitalPictures[0]);
    digitalPictures[0].show();

    System.out.println(digitalPictures[1]);
    digitalPictures[1].show();
  }//end main
}//end class Prob05
//End program specifications.
```

[387] http://cnx.org/content/m44252/latest/Prob05a.jpg
[388] http://cnx.org/content/m44252/latest/Prob05b.jpg
[389] http://cnx.org/content/m44252/latest/Prob05.java

First required output image for Prob05.

Figure 5.539: First required output image for Prob05.

Second required output image for Prob05.

Figure 5.540: Second required output image for Prob05.

5.5.1.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2321 Practice Group 1
- File: PracticeGroup01.htm
- Published: 08/03/12
- Revised: 02/10/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.5.2 Java OOP: ITSE 2321 Practice Group 2[390]

5.5.2.1 ITSE 2321 Object-Oriented Programming - Practice Group 2

5.5.2.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs in this group require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [391] .

5.5.2.1.2 Input Image Files

Links are provided within the individual program specifications for downloading any image files that may be required to write, compile, and test your programs.

5.5.2.1.3 Solution source code files

Links are provided within the individual program specifications for downloading source code files that contain the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [392] .

5.5.2.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

5.5.2.1.5 New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX.

[390] This content is available online at <http://cnx.org/content/m44254/1.6/>.
[391] http://cnx.org/content/m44148/latest/
[392] http://cnx.org/content/m44148/latest/

5.5.2.1.6 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click on the input image file link(s) and use the capabilities of your browser to download and save the image file(s).
2. Click on the Java solution source code link(s) and use the capabilities of your browser to download and save the source code file(s).
3. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
4. Write, compile, and execute a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

In addition to the hints listed above, I will precede the detailed specifications for each program with a discussion that contains hints about the concepts and skills that you will probably need to successfully write the program.

In order to write this or any other Java program of substance, you will need to know how to use [393] the Java Platform, Standard Edition API Specification [394] as well as the documentation for the Guzdial-Ericson Multimedia Class Library [395] .

You may find other useful hints in my online tutorials and slides for this course as well as in the YouTube video lectures for this course.

5.5.2.1.7 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [396] .

5.5.2.1.8 Program Specifications

5.5.2.1.8.1 Program 1

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob01.java [397] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items that were listed with the programs in Practice Group 1 [398] . Instead, I will concentrate on new concepts, knowledge, and skills not included in previous lists.

- How to design and implement an algorithm that will transform the original image [399] into the image shown in Figure 1 (p. 1357) .
- How to instantiate an object and call a method on that object in a single statement [400] .

[393] http://cnx.org/content/m45117/latest/
[394] http://docs.oracle.com/javase/7/docs/api/index.html
[395] http://cnx.org/content/m44148/latest/#Discussion_and_sample_code
[396] http://cnx.org/content/m44148/latest/
[397] http://cnx.org/content/m44254/latest/Prob01.java
[398] http://cnx.org/content/m44252/latest/
[399] http://cnx.org/content/m44254/latest/Prob01.jpg
[400] http://cnx.org/content/m44207/latest/#Listing_1

- How methods belonging to an object call other methods [401] belonging to the same object.
- How to call a method on an object to modify the object saving only [402] a reference to the modified object.
- The use of nested loops [403] to access and modify image pixels on the basis of the horizontal and vertical coordinates of the pixels.
- How to create a mirror image of a portion of an image about a vertical line [404] in the image.
- How to create a mirror image of a portion of an image about a horizontal [405] line in the image.

Listing 5.6: Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image file named Prob01.jpg [406] to produce the graphic output image shown in Figure 1 (p. 1357) below.

Click Prob01.java [407] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
    Display your name here.
Picture, filename Prob01.jpg height 240 width 320
*******************************************************/
public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Picture pic = new Prob01Runner().run();
    System.out.println(pic);
  }//end main method
}//end class Prob01
//End program specifications.
```

[401] http://cnx.org/content/m44207/latest/#Listing_3
[402] http://cnx.org/content/m44207/latest/#Listing_3
[403] http://cnx.org/content/m44207/latest/#Listing_5
[404] http://cnx.org/content/m44207/latest/#Listing_4
[405] http://cnx.org/content/m44207/latest/#Listing_7
[406] http://cnx.org/content/m44254/latest/Prob01.jpg
[407] http://cnx.org/content/m44254/latest/Prob01.java

Required output image for Prob01.

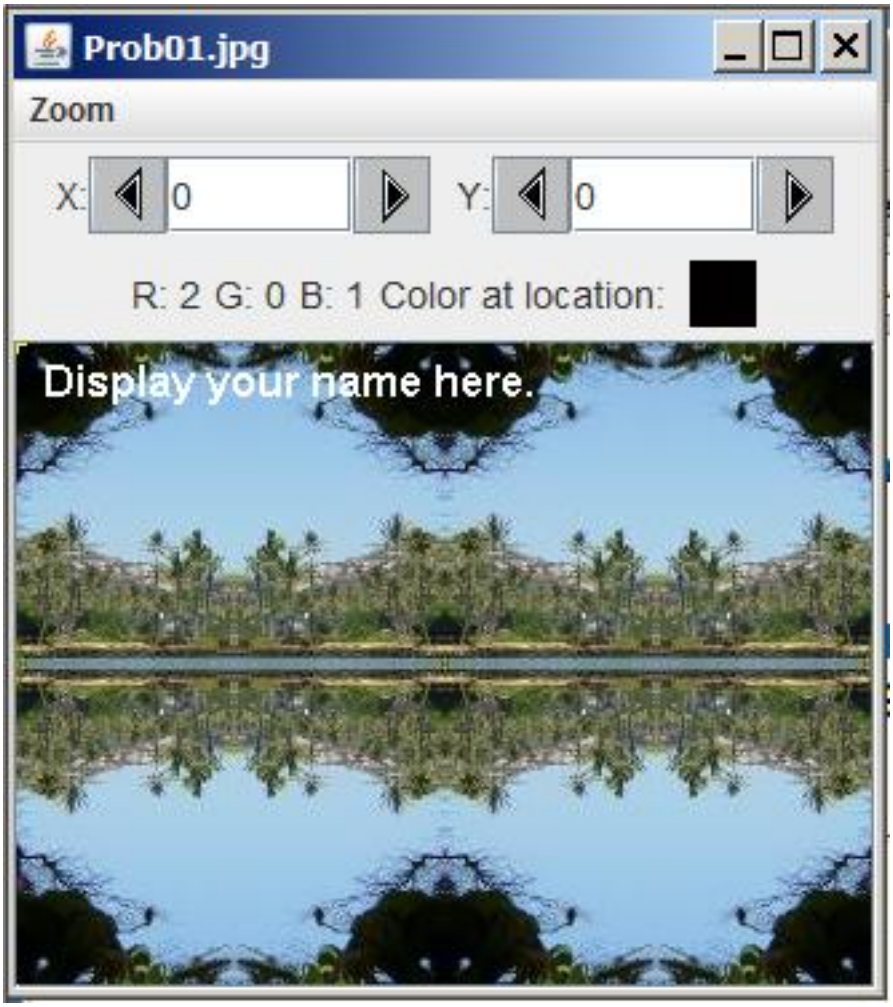


Figure 5.541: Required output image for Prob01.

5.5.2.1.8.2 Program 2

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob02.java [408] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items that were listed with the programs in Practice Group 1 [409] or earlier programs in this practice group. Instead, I will concentrate on new concepts, knowledge, and

[408] http://cnx.org/content/m44254/latest/Prob02.java
[409] http://cnx.org/content/m44252/latest/

skills not included in previous lists.

- How to design and implement an algorithm that will transform Prob02a.jpg [410] and Prob02b.jpg [411] into the image shown in Figure 2 (p. 1359) .
- How to flip [412] an image around its center line.
- How to crop [413] an image.
- How to copy [414] one image onto another image.

Listing 5.7: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses the class definition shown below and Ericson's media library along with the image files named Prob02a.jpg [415] and Prob02b.jpg [416] to produce the graphic output image shown in Figure 2 (p. 1359) below.

Click Prob02.java [417] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename Prob02a.jpg height 118 width 100
Picture, filename Prob02b.jpg height 240 width 320
Picture, filename None height 101 width 77
**********************************************************/
public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Picture[] pictures = new Prob02Runner().run();
    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob02
//End program specifications.
```

[410] http://cnx.org/content/m44254/latest/Prob02a.jpg
[411] http://cnx.org/content/m44254/latest/Prob02b.jpg
[412] http://cnx.org/content/m44238/latest/#Listing_4
[413] http://cnx.org/content/m44238/latest/#Listing_4
[414] http://cnx.org/content/m44238/latest/#Listing_7
[415] http://cnx.org/content/m44254/latest/Prob02a.jpg
[416] http://cnx.org/content/m44254/latest/Prob02b.jpg
[417] http://cnx.org/content/m44254/latest/Prob02.java

Required output image for Prob02.

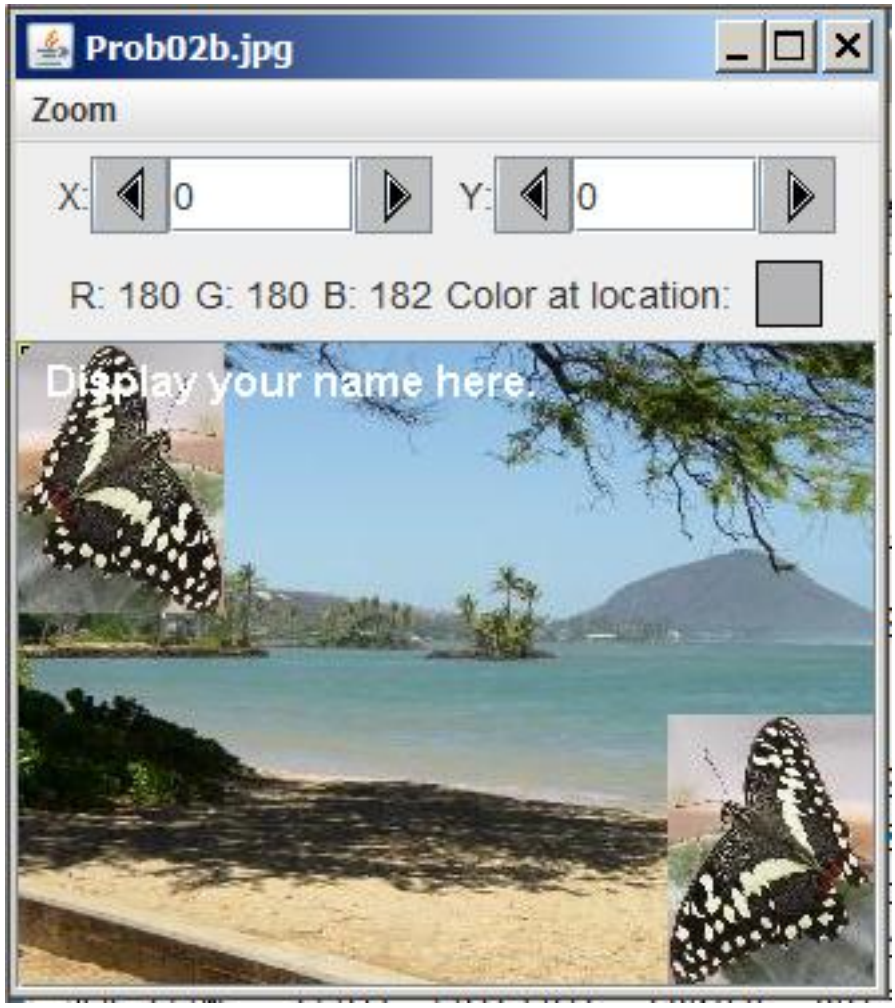


Figure 5.542: Required output image for Prob02.

5.5.2.1.8.3 Program 3

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob03.java [418] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items that were listed with the programs in Practice Group 1 [419] or earlier programs in this practice group. Instead, I will concentrate on new concepts, knowledge, and

[418] http://cnx.org/content/m44254/latest/Prob03.java
[419] http://cnx.org/content/m44252/latest/

skills not included in previous lists.

- How to design and implement an algorithm that will transform Prob03a.bmp [420] , Prob03b.bmp [421] , Prob03c.bmp [422] , and Prob03d.jpg [423] into the image shown in Figure 3 (p. 1361) .
- Green screen processing [424] of image data.
- Differences among bmp, jpg [425] , and png files insofar as green screen processing is concerned.
- Scaling [426] the size of images while maintaining the aspect ratio.

Listing 5.8: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses the class definition shown below and Ericson's media library along with the image files named Prob03a.bmp [427] , Prob03b.bmp [428] , Prob03c.bmp [429] , and Prob03d.jpg [430] to produce the graphic output image shown in Figure 3 (p. 1361) below.

Click Prob03.java [431] to download a Java source file containing the solution to this program.

In addition to the output, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename None height 256 width 344
*********************************************************/
public class Prob03{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob03Runner obj = new Prob03Runner();
    obj.run();
  }//end main
}//end class Prob03
//End program specifications.
```

[420] http://cnx.org/content/m44254/latest/Prob03a.bmp
[421] http://cnx.org/content/m44254/latest/Prob03b.bmp
[422] http://cnx.org/content/m44254/latest/Prob03c.bmp
[423] http://cnx.org/content/m44254/latest/Prob03d.jpg
[424] http://cnx.org/content/m44210/latest/
[425] http://cnx.org/content/m44210/latest/#Discussion_and_sample_code
[426] http://cnx.org/content/m44210/latest/#Listing_4
[427] http://cnx.org/content/m44254/latest/Prob03a.bmp
[428] http://cnx.org/content/m44254/latest/Prob03b.bmp
[429] http://cnx.org/content/m44254/latest/Prob03c.bmp
[430] http://cnx.org/content/m44254/latest/Prob03d.jpg
[431] http://cnx.org/content/m44254/latest/Prob03.java

Required output image for Prob03.

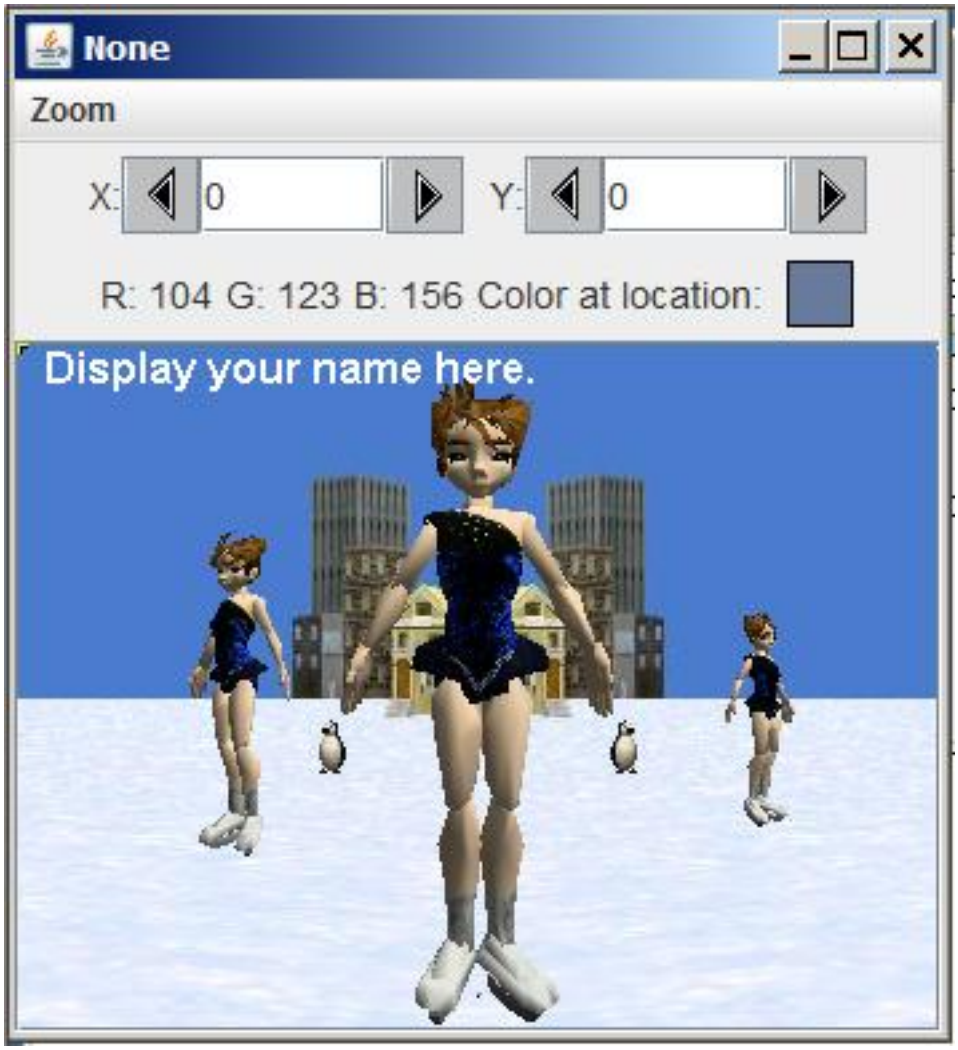


Figure 5.543: Required output image for Prob03.

5.5.2.1.8.4 Program 4

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob04.java [432] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items that were listed with the programs in Practice Group

[432] http://cnx.org/content/m44254/latest/Prob04.java

1 [433] or earlier programs in this practice group. Instead, I will concentrate on new concepts, knowledge, and skills not included in previous lists.

- How to design and implement an algorithm that will transform Prob04a.bmp [434] , Prob04b.bmp [435] , and Prob04c.jpg [436] into the image shown in Figure 4 (p. 1363) .
- How to use one image as a pattern [437] to perform modifications on another image.
- How to brighten [438] the colors of selected pixels in an image.
- How to tint [439] the colors of selected pixels in an image.

Listing 5.9: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses the class definition shown below and Ericson's media library along with the image files named Prob04a.bmp [440] , Prob04b.bmp [441] , and Prob04c.jpg [442] to produce the graphic output image shown in Figure 4 (p. 1363) below.

Click Prob04.java [443] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename None height 293 width 392
***********************************************************/
import java.awt.Color;
public class Prob04{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    obj.run();
  }//end main
}//end class Prob04
//End program specifications.
```

[433] http://cnx.org/content/m44252/latest/
[434] http://cnx.org/content/m44254/latest/Prob04a.bmp
[435] http://cnx.org/content/m44254/latest/Prob04b.bmp
[436] http://cnx.org/content/m44254/latest/Prob04c.jpg
[437] http://cnx.org/content/m44234/latest/#Listing_5
[438] http://cnx.org/content/m44234/latest/
[439] http://cnx.org/content/m44234/latest/#Listing_8
[440] http://cnx.org/content/m44254/latest/Prob04a.bmp
[441] http://cnx.org/content/m44254/latest/Prob04b.bmp
[442] http://cnx.org/content/m44254/latest/Prob04c.jpg
[443] http://cnx.org/content/m44254/latest/Prob04.java

Required output image for Prob04.

Figure 5.544: Required output image for Prob04.

5.5.2.1.8.5 Program 5

Discussion

The following is a non-exhaustive list of concepts that you need to understand along with knowledge and skills that you need to possess in order to successfully write this program and/or understand the given solution in Prob05.java [444] . Some of these items are general in nature and some are specific to the use of Ericson's multimedia library. I won't repeat the items that were listed with the programs in Practice Group 1 [445] or earlier programs in this practice group. Instead, I will concentrate on new concepts, knowledge, and skills not included in previous lists.

- How to design and implement an algorithm that will transform Prob05a.jpg [446] into the image shown in Figure 5 (p. 1365) .

[444] http://cnx.org/content/m44254/latest/Prob05.java
[445] http://cnx.org/content/m44252/latest/
[446] http://cnx.org/content/m44254/latest/Prob05.jpg

- How to extend a class to make it possible to override one or more methods [447] belonging to the class.

Listing 5.10: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses the class definition shown below and Ericson's media library along with the image file named Prob05a.jpg [448] to produce the graphic output image shown in Figure 5 (p. 1365) below.

Click Prob05.java [449] to download a Java source file containing the solution to this program.

In addition to the output images mentioned above, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Simple Picture, filename Prob05.jpg height 240 width 320

**********************************************************/
public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    MyPicture pic = new Prob05Runner().run();
    System.out.println(pic);
    pic.addMessage("String",10,50);
    pic.show();
  }//end main method
}//end class Prob05
//End program specifications.
```

[447] http://cnx.org/content/m44177/latest/
[448] http://cnx.org/content/m44254/latest/Prob05.jpg
[449] http://cnx.org/content/m44254/latest/Prob05.java

Required output image for Prob05.

Figure 5.545: Required output image for Prob05.

5.5.2.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2321 Practice Group 2
- File: PracticeGroup02.htm
- Published: 08/03/12
- Revised: 02/10/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

5.5.3 Java OOP: ITSE 2321 Practice Group 3[450]

5.5.3.1 ITSE 2321 Object-Oriented Programming - Practice Group 3

5.5.3.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs in this group require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [451] .

5.5.3.1.2 Input Image Files

Links are provided within the individual program specifications for downloading any image files that may be required to write, compile, and test your programs.

5.5.3.1.3 Solution source code files

Links are provided within the individual program specifications for downloading source code files that contain the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [452] .

5.5.3.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

5.5.3.1.5 New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX.

[450] This content is available online at <http://cnx.org/content/m44255/1.4/>.
[451] http://cnx.org/content/m44148/latest/
[452] http://cnx.org/content/m44148/latest/

5.5.3.1.6 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click on the input image file link(s) and use the capabilities of your browser to download and save the image file(s).
2. Click on the Java solution source code link(s) and use the capabilities of your browser to download and save the source code file(s).
3. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
4. Write, compile, and execute a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course as well as in the YouTube video lectures for this course.

5.5.3.1.7 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [453] .

5.5.3.1.8 Program Specifications

5.5.3.1.8.1 Program 1

Listing 5.11: Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image file named Prob01.jpg [454] to produce the graphic output image shown in Figure 1 (p. 1369) below.

Click Prob01.java [455] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
    Display your name here.
Picture, filename None height 201 width 201
*******************************************************/
public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob01Runner().run();
  }//end main method
}//end class Prob01//End program specifications.
```

[453] http://cnx.org/content/m44148/latest/
[454] http://cnx.org/content/m44255/latest/Prob01.jpg
[455] http://cnx.org/content/m44255/latest/Prob01.java

Required output image for Prob01.

Figure 5.546: Required output image for Prob01.

5.5.3.1.8.2 Program 2

Listing 5.12: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses the class definition shown below and Ericson's media library along with the image file named Prob02.jpg [456] to produce the graphic output image shown in Figure 2 (p. 1370) below.

Click Prob02.java [457] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename None height 404 width 425
*************************************************************/
public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
```

[456] http://cnx.org/content/m44255/latest/Prob02.jpg
[457] http://cnx.org/content/m44255/latest/Prob02.java

```
    new Prob02Runner().run();
  }//end main method
}//end class Prob02//End program specifications.
```

Required output image for Prob02.

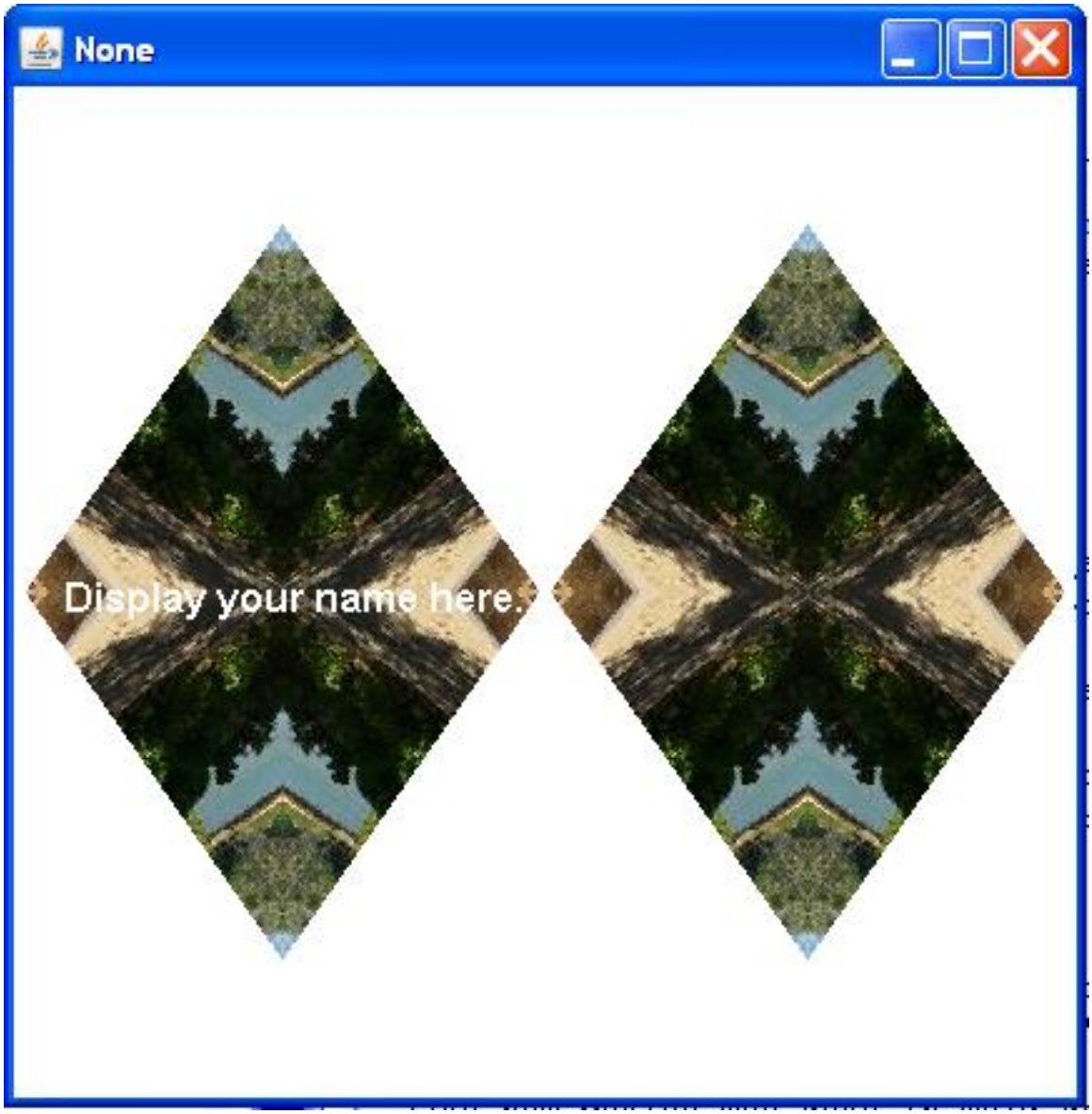


Figure 5.547: Required output image for Prob02.

5.5.3.1.8.3 Program 3

Listing 5.13: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses the class definition shown below and Ericson's media library to produce the graphic output image shown in Figure 3 (p. 1372) below.

Click Prob03.java [458] to download a Java source file containing the solution to this program.

In addition to the output, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename None height 300 width 300
*******************************************************/
public class Prob03{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03//End program specifications.
```

[458] http://cnx.org/content/m44255/latest/Prob03.java

Required output image for Prob03.

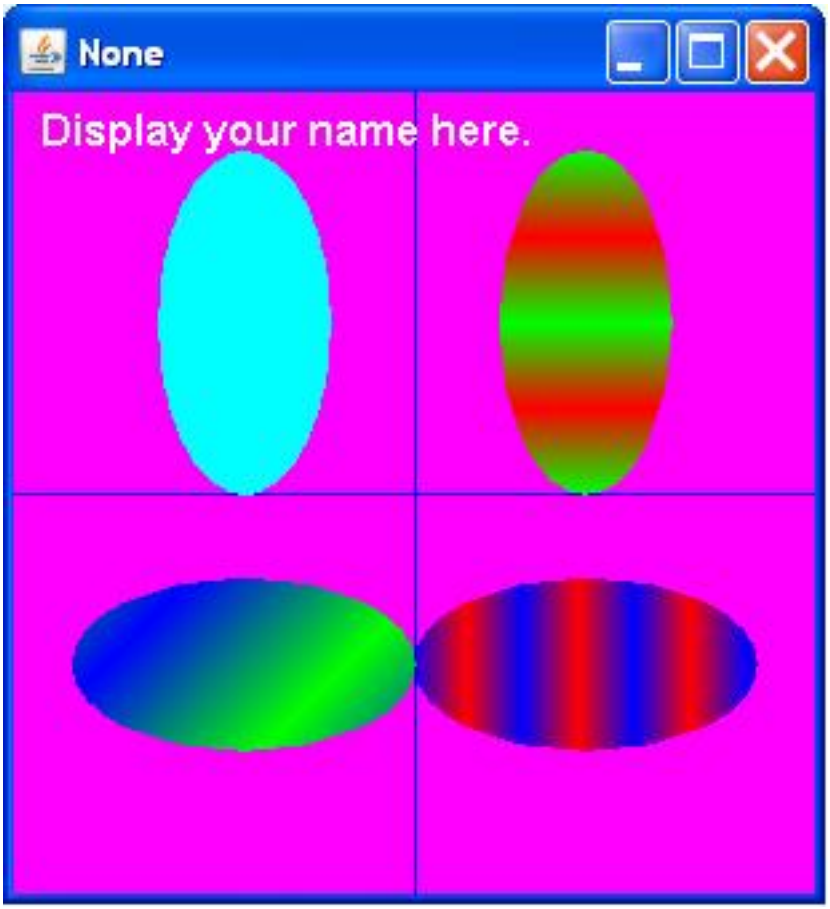


Figure 5.548: Required output image for Prob03.

5.5.3.1.8.4 Program 4

Listing 5.14: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses the class definition shown below and Ericson's media library along with the image file named Prob04a.jpg [459] to produce the graphic output image shown in Figure 4 (p. 1373) below.

Click Prob04.java [460] to download a Java source file containing the solution to this program.

In addition to the output image, your program must display your name and the other text shown below on the command-line screen:

[459] http://cnx.org/content/m44255/latest/Prob04a.jpg
[460] http://cnx.org/content/m44255/latest/Prob04.java

```
Display your name here.
Picture, filename None height 256 width 341
*******************************************************/
public class Prob04{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob04Runner().run();
  }//end main method
}//end class Prob04
//End program specifications.
```

Required output image for Prob04.

Figure 5.549: Required output image for Prob04.

5.5.3.1.8.5 Program 5

Listing 5.15: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses the class definition shown below and Ericson's media library along with the image files named Prob05a.jpg [461] and Prob05b.jpg [462] to produce the graphic output image shown in Figure 5 (p. 1375) below.

Click Prob05.java [463] to download a Java source file containing the solution to this program.

In addition to the output images mentioned above, your program must display your name and the other text shown below on the command-line screen:

```
Display your name here.
Picture, filename None height 252 width 330
*******************************************************/
public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob05Runner().run();
  }//end main method
}//end class Prob05//End program specifications.
```

[461] http://cnx.org/content/m44255/latest/Prob05a.jpg
[462] http://cnx.org/content/m44255/latest/Prob05b.jpg
[463] http://cnx.org/content/m44255/latest/Prob05.java

Required output image for Prob05.

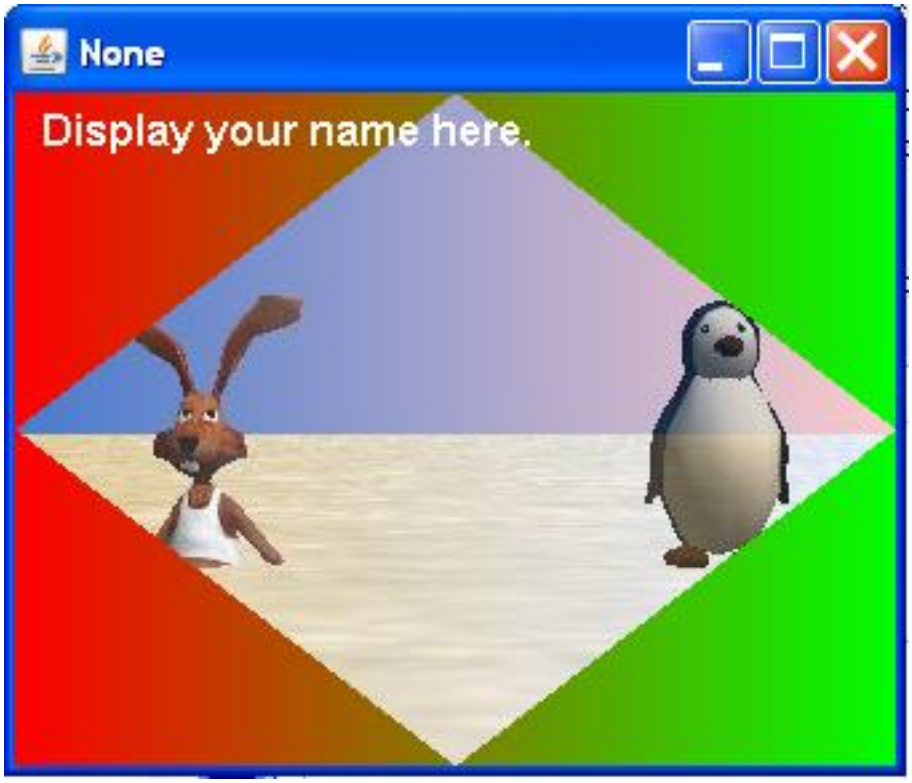


Figure 5.550: Required output image for Prob05.

5.5.3.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2321 Practice Group 3
- File: PracticeGroup03.htm
- Published: 08/03/12
- Revised: 02/10/13

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 6

ITSE2317 - Java Programming (Intermediate)

6.1 Jy0030: Java OOP: Preface to ITSE 2317[1]

6.1.1 Welcome

Welcome to the course material for ***ITSE2317 - Java Programming (Intermediate)*** , which I teach at Austin Community College [2] in Austin, TX.

Official information about the course

The college website for this course is: http://www.austincc.edu/baldwin/ [3]

As of November 2012, the description for this course reads:

"ITSE 2317 - Java Programming (Intermediate)

Introduction to JAVA programming with object-orientation. Emphasis on the fundamental syntax and semantics of JAVA for applications and web applets."

The prerequisite for the course is ITSE 2321 [4] or department approval.

Downloads

I encourage you to take advantage of all of the download options that cnx.org has to offer in order to customize this material for use in your organized courses or for personal self study.

And if you find the material useful, I would like to hear more about how you are using it.

6.1.2 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0030: Java OOP: Preface to ITSE 2317
- File: Jy0030.htm
- Published: November 29, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a

[1] This content is available online at <http://cnx.org/content/m45258/1.1/>.
[2] http://www.austincc.edu/
[3] http://www.austincc.edu/baldwin/
[4] http://cnx.org/content/m45222

pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.2 Essence of OOP

6.2.1 Java OOP: The AWT and Swing, A Preview[5]

6.2.1.1 Table of Contents

6.2.1.2 Preface

This module is part of a collection of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

(Editor's note: As you will see when you read this module, the original version was written about fourteen years ago. However, despite numerous improvements that have occurred in Java since then, most of what was true on this topic in 1998 is still true in 2012. A wise man once said, "If it ain't broke, don't fix it."

One of the changes that you will see in subsequent modules is that the name of the package that contains the ***Swing*** *classes has been changed to* ***javax.swing*** *.)*

6.2.1.3 Discussion

This module provides a very brief preview of some of what you can expect to find in subsequent modules regarding the Abstract Windows Toolkit (**AWT**) and the **Swing** component set.

The user interface of a modern computer program often involves techniques to activate many of the human senses. We use *icons, text boxes, images, sound, boardroom graphics, etc.*

We weren't too concerned with these aspects of programming in the prerequisite course *(ITSE 2321)* because there was a lot that you needed to learn to prepare yourself for understanding material of this sort. That is about to change.

Much of the actual programming that you will do with Java will involve those aspects of the interface that we commonly refer to as the **Graphical User Interface (GUI)** .

As of 5/10/98, there are two primary packages that are used for **GUI** programming under JDK 1.1.6:

1. java. **awt** .*

[5]This content is available online at <http://cnx.org/content/m44331/1.2/>.

2. com.sun.java. **swing** .*

There are, of course, numerous other packages that are used in support of these two packages.

The **AWT** material was made available to Java programmers early in the life of Java. This was the original material that was used to create graphical user interfaces. Major improvements to the **AWT** were introduced with the release of JDK 1.1.

The **Swing** components became available in released form for use with JDK 1.1 around the beginning of 1998. These components added significantly to the ability of the programmer to create GUIs, both in terms of functionality and cosmetics.

The capability and cosmetics of the **AWT** were very limited but **Swing** made GUI programming in Java competitive in the real world. A Java programmer no longer need apologize for the quality of the GUIs that she can create.

We expect that these two packages may become more integrated (causing changes in your **import** directives) with the release of JDK 1.2, *(probably sometime in 1998)* but hopefully the concepts involved won't be greatly different.

As of 3/5/97, there were more than fifty classes defined in package **java.awt** . We will discuss some of the more important **AWT** classes in subsequent modules.

As of 5/10/98, the *com.sun.java.swing* package contains more than 75 classes and about 20 interfaces. You might expect, therefore, that learning to use this material effectively won't be a trivial task.

It is very important to understand that **Swing** is an extension of, and not a replacement for the **AWT** . While it is true that there is some overlap *(for example a **Swing JButton** component might be viewed as an improved functional replacement for an **AWT Button** component, and once you begin using **Swing** buttons you may choose to never again use an **AWT** button)* , the basic functionality of **Swing** is built upon the functionality of the **AWT** .

Therefore, as students, we cannot simply skip over an understanding of the **AWT** and move on to **Swing** . The **AWT** is the foundation for **Swing** .

We must first understand the **AWT** and then understand how **Swing** extends and improves on the **AWT** . I will attempt to integrate an understanding of both the **AWT** and **Swing** in the remaining modules in this collection.

We will begin by introducing you to a few simple components of each type and use these components to teach you about such topics as event-driven programming, layout, graphics, etc. Then, time permitting, we will dig a little deeper into the more complex aspects of both the **AWT** and **Swing** components and other features.

What I won't do is show you a lot of pictures of various **AWT** and **Swing** components as is the case with many books and other tutorials *(although such pictures can be important for an appreciation of GUI programming)* . *(Have you noticed how many Java books use copies of the JavaSoft documentation as filler material to make the book appear to contain more information than it actually contains? At least half of many of the books currently in print is nothing more than a reproduction of the documentation that you can download for free from JavaSoft. Oh well, enough of that!)*

If you want to see some pictures of **AWT** and **Swing** components *(which would be only natural)* , you can create them yourself on your own computer screen.

For examples of the **AWT** components, simply look in the folders in the software that you downloaded from JavaSoft. When you install JDK 1.1.6, a folder named "demo" will be created that contains about two-dozen sample programs. Many of these sample programs have graphical user interfaces that make use of the **AWT** . Just run the programs to see examples of the use of the **AWT** .

When you download and install **Swing** 1.0.1, a folder named "examples" will be created. This folder contains about nine folders, each of which contains a demonstration application or applet that makes use of **Swing** . You can run these programs to see the examples on your computer screen.

A particularly interesting demonstration application is the one named **SwingSet** . One of the new components in **Swing** is a tabbed pane that looks much like a common cardboard file folder with a labeled tab on the top, bottom, left, or right. The **AWT** doesn't contain such a component.

This demonstration starts with about twenty such tabbed panes on the screen, each one of which demonstrates one aspect of the use of **Swing** . By clicking on each of the labeled tabs, you can select and exercise one aspect of **Swing** . In addition, there are five menus that contain selections, some of which impact the behavior of some aspect of the demonstration.

While you are there, pay attention to the fact that virtually all of the **Swing** components are also containers, so it is possible to cause other items *(such as images)* to be contained in components such as buttons and menus.

Take a look at the pane labeled *RadioButtons* and see how two different images of JavaSoft's little creature named Duke can be made to function as a radio button. In this case, the selected Duke is waving while the unselected Dukes aren't waving.

Duke shows up again under *ToggleButtons* where the button which has been toggled has Duke animated in a child's swing.

The *Checkboxes* pane uses light bulbs that either are or are not illuminated to illustrate selection of **Checkbox** items.

The examples on the *Slider* pane are truly impressive *(the **AWT** doesn't have a slider component, although it is possible to use a **ScrollBar** as a crude slider)* .

Take a look at the *ListBox* pane to see another example of using images inside of a component.

The *DebugGraphics* pane demonstrates how to run your program in slow motion so that you can see how the components are assembled for debugging purposes. Note that a **Slider** is used to control the speed of assembly of the components.

And of course, every where you turn in this demo, you will see *tool tips* that are not a part of the **AWT** . For a little comic relief, take a look at the *ToolTips* pane.

Don't forget to pull down the *Options* menu and select the "look and feel" of the different panes as you view them.

Actually, words are inadequate to describe what you are going to find when you install and run the **SwingSet** demonstration. To use a corny phrase made famous by an old TV commercial *(which many of you are probably too young to remember)* , "Try it, you'll like it."

6.2.1.4 What's next?

The next module in the collection will take a first look at *callbacks* .

6.2.1.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: The AWT and Swing, A Preview
- File: Java0073.htm
- Published: 1998
- Revised: August 20, 2012

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.2.2 Java OOP: Callbacks - I[6]

6.2.2.1 Table of Contents

6.2.2.2 Preface

This module is one in a series of three modules designed to teach you about callbacks in Object-Oriented Programming (OOP) using Java. The other two modules are titled Callbacks - II and Callbacks - III. If interested, you will find those modules at http://www.dickbaldwin.com/toc.htm [7] .

6.2.2.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.2.2.2.1.1 Figures

6.2.2.2.1.2 Listings

[6] This content is available online at <http://cnx.org/content/m44333/1.2/>.
[7] http://www.dickbaldwin.com/toc.htm

- Listing 6 (p. 1389) . Define the CallBack interface.
- Listing 7 (p. 1389) . Define the Teacher class.
- Listing 8 (p. 1389) . Define the method named register.
- Listing 9 (p. 1390) . The unRegister method.
- Listing 10 (p. 1391) . Define the callTheRoll method.
- Listing 11 (p. 1391) . Define the class named Dog.
- Listing 12 (p. 1392) . Define the class named Callback02.
- Listing 13 (p. 1395) . Complete listing of program named Callback02.

6.2.2.3 Preview

Many processes in the standard Java API make use of a mechanism that might be referred to as a *callback* mechanism. Basically, this is a mechanism where a method in one object asks a method in another object to *"call me back"* or *"notify me"* when an interesting event happens.

An interesting event

For example, an interesting event might be that the price of a specified stock goes above its previous high value, or the toaster finishes toasting the bread.

Multicasting

In fact, many different objects may ask one object to notify them when the interesting event happens. This is sometimes referred to as *multicasting* . *(The one-to-one case is often referred to as unicasting.)*

Going further, many different objects may ask one object to notify them when any interesting event in a family of interesting events happens, and to identify the specific event that actually happened along with the notification.

Many examples

For example, we see different forms of callback activity in conjunction with

- the *Delegation Event Model* used with GUIs in JDK 1.1,
- the *Observer/Observable* concept used in the *Model-View-Controller* paradigm,
- the concept of *Bound Properties* and *Constrained Properties* in *Java Beans* , etc.

You can find examples of all of these in the pages of my online tutorial lessons.

Callback implementation

Callback capabilities are often implemented in other languages by passing a function pointer to another function. The receiving function uses the passed function pointer to call another function when an interesting event happens. However, Java doesn't support function pointers. In this module, we will learn how to implement the callback mechanism using **interfaces** instead.

From the simple to the more complex

As usual, our approach will be to learn the material by reviewing programs that progress from very simple to more complex. As mentioned earlier, this topic consumes all of this module and two additional lessons on my website as well.

Meaningful scenarios

It is usually easier to understand abstract concepts if they are explained in terms of meaningful scenarios. In this case, our scenario will consist of a *teacher* and some *students* . In the beginning there will only be one student. Ultimately there will be many students and there will also be some animals in the classroom as well.

Registration

The students *(and the animals)* register themselves on the teachers roll book to be notified of interesting events. Initially the interesting event will simply be the teacher taking the roll. Ultimately the interesting event will be notification that it is *either* time for recess, *or* it is time for lunch.

Unicast and multicast scenarios

Initially, only one student receives notification of the one type of event. Ultimately, all of the students and all of the animals receive notification of both types of event *(recess or lunch)* but some of those who are notified choose to ignore the notification.

We will refer to the case where only one student is on the list as the *unicast* program. We will refer to the case where many students *(and possibly animals as well)* are on the list as the *multicast* program. This terminology was selected because it matches the terminology used in the JDK 1.1 documentation for the Delegation Event Model.

Without further discussion, let's look at some code.

6.2.2.4 Discussion and sample code

6.2.2.4.1 Unicast sample program

The purpose of this program is to develop a *callback* capability using *Interfaces* . This version of the program is designed to emphasize the structure of the process. Therefore an effort was made to avoid the requirement for any extra code so it doesn't do anything fancy.

A CallBack interface

This program defines a **CallBack** interface *(interface named* ***CallBack***) that can be used to establish a new type of object reference, and also to declare the interface to a method named **callBack** that will be contained in all objects of classes that implement the interface. This method will then be used to notify those objects whenever something interesting happens.

A Teacher class

The program defines a **Teacher** class that has the ability to

- create and maintain a list of *(only)* one object of the interface type *(multiple objects come later)* , and
- to notify that object that something interesting has happened by calling its **callBack** method.

(As mentioned earlier, the size of the list was constrained to only one object in order to emphasize callback structure and avoid getting bogged down in list processing. A subsequent version will implement list processing.)

A Student class

The program defines a class named **Student** that *implements* the **CallBack** interface. Objects of the **Student** class can be *registered* on the list maintained by an object of the **Teacher** class, and can be *notified* by the object of the **Teacher** class whenever something interesting happens. Notification takes the form of calling the **callBack** method on the object.

The method named callBack

The body of the **callBack** method can be designed to do anything, but in this case, to keep things simple, it just announces that it has been called.

The controlling class

Finally, the program defines a controlling class named **Callback01** that ties all the pieces together and exercises them.

The program was originally tested using JDK 1.1.3 under Win95 and more recently tested using JDK 1.7 under Windows Vista.

The output from the program is shown in the complete program listing in a later section.

Interesting unicast code fragments

Listing 1 (p. 1383) defines an interface named **CallBack** that creates a new type and declares a generic method named **callBack** that can be used to execute a callback on any object that is instantiated from a class that implements the interface.

Listing 6.1: Define the CallBack interface.

```
    interface CallBack{
  public void callBack();
}//end interface CallBack
```

A class that can register and notify objects of type CallBack

Next we need a class whose objects can maintain a list of references to objects of type **CallBack** *(objects whose class implements the* ***CallBack*** *interface)* .

We refer to the process of putting an object on the list is *registering* the object.

This class also needs to have the ability to notify all the objects on that list when something interesting happens. We will name this class **Teacher** in keeping with the scenario described earlier.

As mentioned earlier, to keep things simple, and emphasize the callback structure without getting bogged down in list processing, we will begin with a limitation of one object for the length of the list.

The unicast class named Teacher

The unicast **Teacher** class consists of one instance variable of type **CallBack** *(the interface type)* and two instance methods.

One of the methods named **register** places an object on the list. The other method named **callTheRoll** calls the **callBack** method on the object that is on the list.

Note that the object on the list is guaranteed to have a method named **callBack** because it implements the **CallBack** interface. Otherwise, it couldn't get on the list in the first place. This is because the **register** method requires the incoming object's reference to be of type **CallBack** .

The Teacher class is defined in Listing 2 (p. 1384) .

Listing 6.2: Define the Teacher class.

```
    class Teacher{
  CallBack obj; //list of objects of type CallBack
  //-----------------------------------------------------//

  //Method to add objects to the list.
  void register(CallBack obj){
    this.obj = obj;
  }//end register()
  //-----------------------------------------------------//

  //Method to notify all objects on the list
  void callTheRoll(){
    obj.callBack();
  }//end callTheRoll()
  //-----------------------------------------------------//
}//end class Teacher
```

A class that implements the CallBack interface

Next, we need a class that *implements* the **CallBack** interface. Objects of this class can be registered on the list maintained by an object of the **Teacher** class, and will be *notified* whenever that object calls the **callBack** method on the registered objects on the list. In keeping with the scenario described earlier, we will name this class **Student** .

By claiming to implement the **CallBack** interface, this class is required to provide a concrete definition for the method named **callBack** that is declared in the interface. Otherwise, the program won't compile. In this case, that definition is rather simple. The **callBack** method simply announces that it has been called.

The callback mechanism

As we saw above, an object of the **Teacher** class will call the **callBack** method on all objects on its list when the interesting event occurs. It is important to note that the callback mechanism is to call this method.

The **Student** class is defined in Listing 3 (p. 1385) .

Listing 6.3: Define the Student class.

```
  class Student implements CallBack{
String name;
//--------------------------------------------------//

Student(String name){//constructor
  this.name = name;  //save the name to identify the obj
}//end constructor
//--------------------------------------------------//

public void callBack(){
  System.out.println(name + " here");
}//end callBack()
}//end class Student
```

A controlling class named Callback01

Finally, we need a controlling class to tie all the pieces together and to exercise them. The **main** method in this class

- instantiates an object of the **Teacher** class named **missJones** ,
- instantiates an anonymous **Student** object named " **Joe** ",
- registers the object on the list maintained by **missJones** , and
- calls the **callTheRoll** method on **missJones** to cause the objects on the list to be notified (to cause their **callBack** methods to be called).

This is not too complicated once you break the process into its component parts.

The class named Callback01 is defined in Listing 4 (p. 1385) .

Listing 6.4: A controlling class named Callback01.

```
  class Callback01{
public static void main(String[] args){
  //Instantiate Teacher object
  Teacher missJones = new Teacher();
  //Instantiate and register a Student object with the
  // Teacher object
  missJones.register(new Student("Joe"));
  //Cause the Teacher object to do a callBack on the
  // Student object.
  missJones.callTheRoll();
}//end main()
}//end class Callback01
```

There you have it. This simple program contains the sum and substance of one approach to callbacks in Java.

It is critical to note that the objects registered on the list are of the interface type **CallBack** . This guarantees that there cannot be an object on the list that does not have an instance method named **callBack** .

Unicast Program Listing

A complete listing of the program is provided in Listing 5 (p. 1386) so that you can view the code fragments in context.

Listing 6.5: Complete listing of program named Callback01.

```
    /*File Callback01.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a callback
capability using Interfaces.  This version of the
program is designed to emphasize the structure of
the process, and therefore an effort was made to
avoid the requirement for any extra code to do
anything fancy.

Tested using JDK 1.1.3 under Win95.

The output from the program is:

Joe here.
**********************************************************/
//First we define an interface that will create a new type
// and declare a generic method that can be used to
// callback any object that is of a class that implements
// the interface.
interface CallBack{
  public void callBack();
}//end interface CallBack
//=========================================================//

//Next we need a class whose objects can maintain a
// registered list of objects of type CallBack (whose
// class implements the CallBack interface) and can
// notify all the objects on that list when something
// interesting happens.

//To keep things simple, and emphasize the structure of
// what we are doing, we will begin with a limitation
// of one object on the length of the list.

class Teacher{
  CallBack obj; //list of objects of type CallBack
  //-------------------------------------------------------//

  //Method to add objects to the list.
  void register(CallBack obj){
    this.obj = obj;
  }//end register()
  //-------------------------------------------------------//
```

```
  //Method to notify all objects on the list that
  // something interesting has happened.
  void callTheRoll(){
    //Call the callBack() method on the object.  The
    // object is guaranteed to have such a method because
    // it is of a class that implements the CallBack
    // interface.
    obj.callBack();
  }//end callTheRoll()
  //-------------------------------------------------//
}//end class Teacher
//=========================================================//

//Class that implements the CallBack interface.  Objects
// of this class can be registered on the list maintained
// by an object of the Teacher class, and will be notified
// whenever that object calls the callBack method on the
// registered objects on the list.
class Student implements CallBack{
  String name;
  //-------------------------------------------------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify the obj
  }//end constructor
  //-------------------------------------------------//

  //An object of the Teacher class will call this method
  // as the callback mechanism to notify an object of this
  // class that something interesting has happened.
  public void callBack(){
    System.out.println(name + " here");
  }//end overridden callBack()
}//end class Student
//=========================================================//

//Controlling class that ties all the pieces together and
// exercises them.
class Callback01{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();
    //Instantiate and register a Student object with the
    // Teacher object
    missJones.register(new Student("Joe"));
    //Cause the Teacher object to do a callBack on the
    // Student object.
    missJones.callTheRoll();
  }//end main()
}//end class Callback01
//=========================================================/
```

6.2.2.4.2 Multicast sample program

The multicast version of this program does not modify the basic callback mechanism developed in the previous program. It simply enhances that mechanism to make it possible to maintain *a list of objects* registered for callback and to notify all the objects on that list when an interesting event happens.

In case you started reading at this point, this is an enhanced version of the program named **Callback01** . You should familiarize yourself with that program before trying to understand this program.

A list of registered objects

This program has the capability to create and maintain a list of objects that register for callback whereas the program named **Callback01** could only remember a single object for callback.

Multiple classes implement CallBack interface

In addition, this program defines two different classes that implement the **CallBack** interface. Mixed objects of those two types are maintained on the list and notified at callback time. This is a subtle but very important point. It is not necessary that all the objects that are registered on a callback list be of the same class type, only that they all be of a class that implements the **CallBack** interface.

The CallBack interface

As before, this program defines a **CallBack** interface that establishes a new type of object, and also declares the interface to a method named **callBack** that is contained in all objects of classes that implement the interface. Because the **callBack** method is guaranteed to be contained in all of the objects on the list, it can be used to notify those objects whenever something interesting happens.

The Teacher class

The program defines a **Teacher** class that creates and maintains a list of objects of the **CallBack** interface type, and notifies those objects that something interesting has happened by calling the **callBack** method on each of the objects on the list.

The size of the list is limited only to the largest **Vector** object that can be accommodated by the system. *(See the Java documentation or my online tutorials for information about the **Vector** class.)*

The Student and Dog classes

The program defines a class named **Student** that implements the **CallBack** interface. The program also defines a class named **Dog** that implements the **CallBack** interface as well. *(Back in the description of the scenario, I promised you that **missJones** was going to have to deal with animals in the classroom. I'm glad I don't have that problem.)*

Registration and notification of Student and Dog objects

Objects of the **Student** and **Dog** classes can be *registered* on the list *(of **CallBack** objects)* maintained by an object of the **Teacher** class *(because they both implement the **CallBack** interface)* , and can be *notified* by the object of the **Teacher** class whenever something interesting happens.

Addition and removal from the list

Note that objects can be added to the list and then removed from the list. One object is first added and later removed for demonstration purposes.

The callback mechanism

As before, notification takes the form of calling the **callBack** method on each of the objects on the list.

Behavior of the callBack methods

The behavior of the **callBack** methods in the classes that implement the interface can be designed to do anything. In this case, to keep things simple, they just announce that they have been called. However, they make the announcement in slightly different ways.

Text display statements

This program contains display statements in the registration and notification methods for demonstration purposes only, and to allow us to track what is happening as the program runs.

The controlling class

Finally, the program defines a controlling class named **Callback02** that ties all the pieces together and exercises them.

The program was originally tested using JDK 1.1.3 under Win95 and more recently tested using JDK 1.7 under Windows Vista.

The output from the program is shown following a discussion of the controlling class at the end of the next section.

Interesting multicast code fragments

Listing 6 (p. 1389) defines an interface that creates a new type and declares a generic method that can be used to call back any object that is of a class that implements the interface. There is nothing new here.

Listing 6.6: Define the CallBack interface.

```
 interface CallBack{
  public void callBack();
}//end interface CallBack
```

A class that can register and notify objects of type CallBack

Next we need a class whose objects can maintain a registered list of objects of type **CallBack** *(objects whose class implements the* ***CallBack*** *interface)* and can *notify* all the objects on that list when something interesting happens. As before, we name this class **Teacher** .

The Teacher class

The **Teacher** class has grown to the point that we will break it into parts and discuss them separately.

There is quite a bit here that is new, due simply to the requirement for list processing. There is nothing new about the basic callback mechanism.

An object of type Vector

We start out by replacing the single instance variable of type **CallBack** by a reference to an object of type **Vector** . We will maintain our list in an object of type **Vector** .

Recall that a **Vector** object can only work with references to objects of type **Object** , so this will entail some down casting later.

(Editor's note: Sometime around JDK 1.5, a concept known as generics was released into Java, which eliminated the restriction to objects of type ***Object*** *mentioned in the previous paragraph. However, this code has not been updated to take advantage of that capability.)*

The constructor for our new **Teacher** class, which is shown in Listing 7 (p. 1389) , instantiates the **Vector** object.

Listing 6.7: Define the Teacher class.

```
 class Teacher{
Vector objList; //list of objects of type CallBack
//-----------------------------------------------//

Teacher(){//constructor
  objList = new Vector();
}//end constructor
```

The method named register

Next we need a method to add objects to the list. We will synchronize it to protect against the possibility of two or more objects on different threads trying to register at the same time.

Note that the references to the objects are received as type **CallBack** , which is the interface type, and stored as type **Object** , because the **Vector** class only accommodates references to objects of type **Object** . *(See the earlier editor's note.)* Again, this will lead to some down casting requirements later.

Listing 6.8: Define the method named register.

```
  synchronized void register(CallBack obj){
 this.objList.addElement(obj);
 System.out.println(obj + " added");
}//end register()
```

The unRegister method

To be general, we also need a method to remove objects from the list. Removal of an object from the list is a little more complicated than adding an object to the list due to the possibility of having two or more identical objects on the list. *(We could, and possibly should, guard against that possibility when constructing the list.)*

Figure 1 (p. 1390) contains a partial excerpt from the JDK 1.1.3 documentation, which describes the **removeElement** method of the **Vector** class that we are using to accomplish this *(three different methods are available to remove objects from a* ***Vector*** *).*

Documentation for the removeElement method.

```
public final synchronized boolean removeElement(Object obj)

This method removes the first occurrence of the argument from this
vector. Indices beyond that point are adjusted appropriately

Parameters: obj - the component to be removed.

Returns: true if the argument was a component of this vector;
         false otherwise.
```

Figure 6.1: Documentation for the removeElement method.

Registered object removal code

Given that explanation, the code for removal of an object from the list is straightforward. The **unRegister** method is shown in Listing 9 (p. 1390) .

Listing 6.9: The unRegister method.

```
  synchronized void unRegister(CallBack obj){
 if(this.objList.removeElement(obj))
   System.out.println(obj + " removed");
 else System.out.println(obj + " not in the list");
}//end register()
```

The callTheRoll method

Now we need a method to notify all of the objects on the list that something interesting has happened. We will name this method **callTheRoll** to adhere to our classroom scenario.

Maintain the integrity of the callback list

One of the potential problems with this type of callback mechanism is that when the callback method is called on an object, that method might take a while to finish.

(As an aside, when writing callback methods, if they do anything significant in terms of time, the code in the method should probably spawn another thread to do the actual work and return as quickly as possible.)

This leads to the possibility that additional objects might attempt to register during that time interval. To protect against this, we make a copy of the state of the list object as it existed at the point in time that the decision was made to do the callbacks, and then perform the callbacks using that copy. That way, the original list is free to be updated as needed during this interval.

So, we start out by creating a clone of the list. We also *synchronize* this process to prevent the list from being modified while we are creating the clone.

Following this, we use a **for** loop to access all the objects on the list, and call the **callBack** method on those objects. *(Actually, the list contains references to objects, and not the actual objects, so we are calling the method on the references.)*

As promised earlier, we have to downcast from **Object** to **CallBack** to gain access to the **callBack** method in the objects.

Listing 6.10: Define the callTheRoll method.

```
void callTheRoll(){
  Vector tempList;//save a temporary copy of list here

  synchronized(this){
    tempList = (Vector)objList.clone();
  }//end synchronized block

  for(int cnt = 0; cnt < tempList.size(); cnt++){
    ((CallBack)tempList.elementAt(cnt)).callBack();
  }//end for loop
}//end callTheRoll()
```

End of the class named Teacher

That ends the discussion of the class named **Teacher** and brings us to the class named **Student** that implements the **CallBack** interface. This class hasn't changed. As indicated earlier, this version of the program also has a class named **Dog** that implements the interface. These two classes are essentially the same.

Define the class named Dog

Because of their similarity, and because they are essentially the same as in the previous program, I will simply show the class named **Dog** with no further discussion.

Listing 6.11: Define the class named Dog.

```
  class Dog implements CallBack{
String name; //store name here for later ID
//-------------------------------------------------------//

Dog(String name){//constructor
  this.name = name; //save the name to identify the obj
}//end constructor
//-------------------------------------------------------//
```

```
  //An object of the Teacher class will call this method
  // as the callback mechanism to notify an object of this
  // class that something interesting has happened.

  public void callBack(){//announce callBack

    System.out.println("Woof, Woof " + name);
  }//end overridden callBack()
}//end class Dog
```

The controlling class

That brings us to the controlling class named **Callback02** that ties all the pieces together and exercises them. This class is shown in Listing 12 (p. 1392) .

Listing 6.12: Define the class named Callback02.

```
class Callback02{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

 //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");

    //Instantiate some Dog objects.
    Dog spot = new Dog("Spot");
    Dog fido = new Dog("Fido");
    Dog brownie = new Dog("Brownie");

    //Register some Student and Dog objects with the
    // Teacher object.
    System.out.println("Register Tom");
    missJones.register(tom);
    System.out.println("Register Spot");
    missJones.register(spot);
    System.out.println("Register Sue");
    missJones.register(sue);
    System.out.println("Register Fido");
    missJones.register(fido);
    System.out.println("Register Peg");
    missJones.register(peg);
    System.out.println("Register Bob");
    missJones.register(bob);
    System.out.println("Register Brownie");
    missJones.register(brownie);

    //Remove a Student object from the list.
```

```
    System.out.println("Remove Peg");
    missJones.unRegister(peg);

    //Try to remove an object that is not on the list.
    System.out.println("Try to remove Joe");
    missJones.unRegister(joe);

    System.out.println();//blank line

    //Cause the Teacher object to do a callBack on all
    // the objects on the list.
    missJones.callTheRoll();
  }//end main()
}//end class Callback02
```

Differences relative to Callback01

This program differs from the previous program primarily in terms of the volume of **Student** and **Dog** objects to be instantiated and registered on the **Teacher** object. There are also a lot of display statements to help us keep track of what is going on.

The ability to remove objects from the list is also illustrated.

Call the roll

Finally, the callback to the objects on the list is executed in Listing 12 (p. 1392) by calling the **callTheRoll** method on the **Teacher** object named **missJones** . The output from running this program is shown later.

Mixed object types

A subtle, but extremely important point is illustrated here. **Student** and **Dog** are different classes. Objects of both of those classes are registered on the single object of the **Teacher** class. The **Teacher** object doesn't care that they are different, so long as they are all instantiated from classes that implement the **CallBack** interface. The **register** method will only accept object references of type **CallBack** .

Program output

The output from running this program is shown in Figure 2 (p. 1394) . You can see the identification of each individual object as it is added to, or removed from the list.

Output from Callback02.

```
    Register Tom
Student@1cc73e added
Register Spot
Dog@1cc74e added
Register Sue
Student@1cc741 added
Register Fido
Dog@1cc751 added
Register Peg
Student@1cc744 added
Register Bob
Student@1cc747 added
Register Brownie
Dog@1cc754 added
Remove Peg
Student@1cc744 removed
Try to remove Joe
Student@1cc74a not in the list

Tom here
Woof, Woof Spot
Sue here
Woof, Woof Fido
Bob here
Woof, Woof Brownie
```

Figure 6.2: Output from Callback02.

Note that the attempt to remove Joe from the list was not successful because he was never registered in the first place.

Finally, you see the output produced by calling **callTheRoll** which in turn calls the **callBack** method on each of the objects on the list.

Note that Peg didn't appear in the roll call because she was first added and then removed from the list before the roll call was taken.

The sum and substance

So there you have it, the sum and substance of multicast callbacks in Java. Obviously improvements could be made. You can see a couple of them in the remaining two tutorial lessons on callbacks that are published on my website.

Multicast Program Listing

A complete listing of the multicast program named **Callback02** is provided in Listing 13 (p. 1395) .

Listing 6.13: Complete listing of program named Callback02.

```
    /*File Callback02.java Copyright 1997, R.G.Baldwin
The purpose of this program is to develop a callback
capability using Interfaces.

This is an enhanced version of the program named
Callback01. You should familiarize yourself with
the earlier program before getting into this program.

This version has the added capability to create and
maintain a list of objects that register for callback
whereas the program named Callback01 could only remember
a single object for callback.

Tested using JDK 1.1.3 under Win95.

The output from the program was:

Register Tom
Student@1cc73e added
Register Spot
Dog@1cc74e added
Register Sue
Student@1cc741 added
Register Fido
Dog@1cc751 added
Register Peg
Student@1cc744 added
Register Bob
Student@1cc747 added
Register Brownie
Dog@1cc754 added
Remove Peg
Student@1cc744 removed
Try to remove Joe
Student@1cc74a not in the list

Tom here
Woof, Woof Spot
Sue here
Woof, Woof Fido
Bob here
Woof, Woof Brownie

Note that Peg didn't appear in the callBack list because
she was first added to, and later removed from the list.
**********************************************************/
import java.util.*;

//First we define an interface that will create a new type
```

```
// and declare a generic method that can be used to
// callback any object that is of a class that implements
// the interface.
interface CallBack{
  public void callBack();
}//end interface CallBack
//=======================================================//

//Next we need a class whose objects can maintain a
// registered list of objects of type CallBack (whose
// class implements the CallBack interface) and can
// notify all the objects on that list when something
// interesting happens.

class Teacher{
  Vector objList; //list of objects of type CallBack
  //-------------------------------------------------------//

  Teacher(){//constructor
    //Instantiate a Vector object to contain the list
    // of registered objects.
    objList = new Vector();
  }//end constructor
  //-------------------------------------------------------//

  //Method to add objects to the list.  Synchronize to
  // protect against two or more objects on different
  // threads trying to register at the same time.  Note
  // that the objects are received as type CallBack which
  // is the interface type, and stored as type Object,
  // because the Vector class only accommodates objects of
  // type Object.
  synchronized void register(CallBack obj){
    this.objList.addElement(obj);
    System.out.println(obj + " added");
  }//end register()
  //-------------------------------------------------------//

  //Method to remove objects from the list.
  synchronized void unRegister(CallBack obj){
    if(this.objList.removeElement(obj))
      //true when successfully found and removed
      System.out.println(obj + " removed");
    else//false on failure to find and remove
      System.out.println(obj + " not in the list");
  }//end register()
  //-------------------------------------------------------//

  //Method to notify all objects on the list that
  // something interesting has happened.
  void callTheRoll(){
```

```
    Vector tempList;//save a temporary copy of list here

    //Make a copy of the list to avoid the possibility of
    // the list changing while objects are being notified.
    // Synchronize to protect against list changing while
    // making the copy.
    synchronized(this){
      tempList = (Vector)objList.clone();
    }//end synchronized block

    //Call the callBack() method on each object on
    // the list.  The object are guaranteed to have such
    // a method, even if they are of different types,
    // because they are all of a class that implements
    // the CallBack interface. If not, they could not
    // have been registered on the list in the first
    // place.  Note the requirement to downcast to
    // type CallBack.
    for(int cnt = 0; cnt < tempList.size(); cnt++){
      ((CallBack)tempList.elementAt(cnt)).callBack();
    }//end for loop
  }//end callTheRoll()
  //-----------------------------------------------------//
}//end class Teacher
//=======================================================//

//Class that implements the CallBack interface.  Objects
// of this class can be registered on the list maintained
// by an object of the Teacher class, and will be notified
// whenever that object calls the callBack method on the
// registered objects on the list.  This program will not
// compile if this class fails to implement the CallBack
// interface

class Student implements CallBack{
  String name; //store the object name here for later ID
  //-----------------------------------------------------//

  Student(String name){//constructor
    this.name = name;  //save the name to identify the obj
  }//end constructor
  //-----------------------------------------------------//

  //An object of the Teacher class will call this method
  // as the callback mechanism to notify an object of this
  // class that something interesting has happened.

  public void callBack(){//announce callBack
    System.out.println(name + " here");
  }//end overridden callBack()
}//end class Student
```

```
//=========================================================//

//Another Class that implements the CallBack interface.
// Objects of this class can also be registered on the list
// maintained by an object of the Teacher class, and will
// also be notified whenever that object calls the
// callBack() method on the registered objects on the
// list. This program will not compile if this class
// fails to implement the CallBack interface.

class Dog implements CallBack{
  String name; //store name here for later ID
  //-------------------------------------------------------//

  Dog(String name){//constructor
    this.name = name; //save the name to identify the obj
  }//end constructor
  //-------------------------------------------------------//

  //An object of the Teacher class will call this method
  // as the callback mechanism to notify an object of this
  // class that something interesting has happened.

  public void callBack(){//announce callBack
    System.out.println("Woof, Woof " + name);
  }//end overridden callBack()
}//end class Dog
//=========================================================//

//Controlling class that ties all the pieces together and
// exercises them.
class Callback02{
  public static void main(String[] args){
    //Instantiate Teacher object
    Teacher missJones = new Teacher();

    //Instantiate some Student objects
    Student tom = new Student("Tom");
    Student sue = new Student("Sue");
    Student peg = new Student("Peg");
    Student bob = new Student("Bob");
    Student joe = new Student("Joe");

    //Instantiate some Dog objects.
    Dog spot = new Dog("Spot");
    Dog fido = new Dog("Fido");
    Dog brownie = new Dog("Brownie");

    //Register some Student and Dog objects with the
    // Teacher object.
    System.out.println("Register Tom");
```

```
    missJones.register(tom);
    System.out.println("Register Spot");
    missJones.register(spot);
    System.out.println("Register Sue");
    missJones.register(sue);
    System.out.println("Register Fido");
    missJones.register(fido);
    System.out.println("Register Peg");
    missJones.register(peg);
    System.out.println("Register Bob");
    missJones.register(bob);
    System.out.println("Register Brownie");
    missJones.register(brownie);

    //Remove a Student object from the list.
    System.out.println("Remove Peg");
    missJones.unRegister(peg);

    //Try to remove an object that is not on the list.
    System.out.println("Try to remove Joe");
    missJones.unRegister(joe);

    System.out.println();//blank line

    //Cause the Teacher object to do a callBack on all
    // the objects on the list.
    missJones.callTheRoll();
  }//end main()
}//end class Callback02
//=============================================================//
```

6.2.2.5 Run the program

I encourage you to copy the code from Listing 5 [8] and Listing 13 (p. 1395) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.2.2.6 Summary

In this module, you learned the fundamentals of callbacks using interfaces in Java.

6.2.2.7 What's next?

In the next module, you will learn about something that goes by the name Delegation Event Model along with a few other names as well.

6.2.2.8 Miscellaneous

This section contains a variety of miscellaneous information.

[8] http://cnx.org/content/m44333/latest/../../Part%201/Java3102/Java3102.htm#Listing_5

NOTE: **Housekeeping material**

- Module name: Java OOP: Callbacks - I
- File: Java0077.htm
- Published: 1998
- Revised: August 20, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.2.3 Java OOP: Event Handling in JDK 1.1, A First Look, Delegation Event Model[9]

This is a placeholder for the CNXML version of this module, which is incomplete.
You will find a copy of the original HTML version at
http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java080/Java080.htm
[10]

-end-

6.2.4 Java OOP: Swing and the Delegation Event Model[11]

This is a placeholder for the CNXML version of this module, which is incomplete.
You will find a copy of the original HTML version at
http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java081/Java081.htm
[12]

-end-

6.2.5 Java OOP: Member Classes[13]

This is a placeholder for the CNXML version of this module, which is incomplete.
You will find a copy of the original HTML version at
http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1636/Java1636.htm
[14]

-end-

[9] This content is available online at <http://cnx.org/content/m44340/1.1/>.
[10] http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java080/Java080.htm
[11] This content is available online at <http://cnx.org/content/m44336/1.1/>.
[12] http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java081/Java081.htm
[13] This content is available online at <http://cnx.org/content/m44347/1.1/>.
[14] http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1636/Java1636.htm

6.2.6 Java OOP: Local Classes[15]

This is a placeholder for the CNXML version of this module, which is incomplete.

You will find a copy of the original HTML version at
http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1638/Java1638.htm
[16]

-end-

6.2.7 Java OOP: Anonymous Classes[17]

This is a placeholder for the CNXML version of this module, which is incomplete.

You will find a copy of the original HTML version at
http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1640/Java1640.htm
[18]

-end-

6.3 Multimedia

6.3.1 Part 1

6.3.1.1 Java OOP: Modifying the World and SimpleTurtle Classes[19]

6.3.1.1.1 Table of Contents

6.3.1.1.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [20] .

[15] This content is available online at <http://cnx.org/content/m44346/1.1/>.
[16] http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1638/Java1638.htm
[17] This content is available online at <http://cnx.org/content/m44342/1.1/>.
[18] http://www.austincc.edu/baldwin/ITSE2317LectureNotesAndSlides/LectureNotes/Java1640/Java1640.htm
[19] This content is available online at <http://cnx.org/content/m44330/1.1/>.
[20] http://cnx.org/content/m44148/latest/

6.3.1.1.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.1.1.2.1.1 Figures

- Figure 1 (p. 1403) . The required graphic output image.
- Figure 2 (p. 1404) . Required output on the command line screen.

6.3.1.1.2.1.2 Listings

- Listing 1 (p. 1405) . Modification to load a jpg file by default.
- Listing 2 (p. 1405) . Original array containing turtle colors.
- Listing 3 (p. 1406) . Modified array of turtle colors.
- Listing 4 (p. 1406) . Change the initial heading.
- Listing 5 (p. 1407) . Source code for driver class named Prob01..
- Listing 6 (p. 1408) . Source code for the modified World class..
- Listing 7 (p. 1413) . Source code for the modified SimpleTurtle class..

6.3.1.1.3 Preview

Program specifications

Write a program named **Prob01** that uses the class definition shown in Listing 5 (p. 1407) and Ericson's media library along with the image file named **Prob01.jpg** to produce the graphic output image shown in Figure 1 (p. 1403) .

The required graphic output image.

Figure 6.3: The required graphic output image.

No new classes

You may **not** define any new classes to cause your program to behave as required, and you may not modify the class definition for the class named **Prob01** given in Listing 5 (p. 1407) . You must copy and modify *(if necessary)* the following media classes to cause your program to produce the required output:

- World.java
- Turtle.java
- SimpleTurtle.java

Files in your folder

Your folder must contain only the following class files, source-code files, and image files:

- Prob01.class
- Prob01.java
- Prob01.jpg
- SimpleTurtle.class
- SimpleTurtle.java
- Turtle.class
- Turtle.java
- World.class

- World.java

Output text

In addition to the output image described above, your program must produce the output text on the command- line screen shown in Figure 2 (p. 1404) .

Required output on the command line screen.

```
    Dick Baldwin
Picture, filename Prob01.jpg height 274 width 365
Dick Baldwin
Dick Baldwin
Dick Baldwin
Dick Baldwin
```

Figure 6.4: Required output on the command line screen.

You must substitute your name for mine wherever my name appears in the image and on the command-line screen.

An analysis

As is often the case, the real challenge with this problem is to decide what needs to be done to satisfy the specifications.

Required modifications

By comparing the default behavior of the **World** and **SimpleTurtle** classes with the requirements of this program, it can be determined that the following modifications to the **World** and **SimpleTurtle** classes are required to meet the specifications. *(Modification of the **Turtle** class is not required)* :

- Modify the **World** class to load a picture named **Prob01.jpg** as the default background for the world in place of the all-white **Picture** object.
- Modify the **World** class to display the student's name near the top of the image.
- Modify the **World** class to display the student's name and information about the picture on the command-line screen.
- Modify the **SimpleTurtle** class to change the initial heading for new turtle objects to northeast instead of north.
- Modify the **SimpleTurtle** class to change the order in which colors are assigned to new turtles as they are instantiated.

6.3.1.1.4 Discussion and sample code

6.3.1.1.4.1 Modifications to the World class

Ericson's **World** class was modified to cause it to load a jpg file by default instead of displaying a blank picture by default. It was also modified to cause it to display text on the background image and to display text on the command line screen. These changes are reflected in Figure 1 (p. 1403) and Figure 2 (p. 1404) .

A complete listing of the modified **World** class is shown in Listing 6 (p. 1408) .

Modifying the code

The code used to accomplish the modifications described above is shown in Listing 1 (p. 1405) .

Listing 6.14: Modification to load a jpg file by default.

```
//create the background picture

//picture = new Picture(width,height);

picture = new Picture("Prob01.jpg");
picture.addMessage("Dick Baldwin",10,20);
System.out.println(picture);
```

Note that one original statement was disabled and replaced by three new statements.

In addition, several other **println** statements were added at strategic locations within the **World** and **SimpleTurtle** classes *(not shown here)* to cause the student's name to appear multiple times in the text output shown in Figure 2 (p. 1404) .

Meeting the requirements

These modifications to the **World** and **SimpleTurtle** classes met the following requirements established earlier under Analysis [21] .

- Modify the **World** class to load a picture named **Prob01.jpg** as the default background for the world in place of the all-white **Picture** object.
- Modify the **World** class to display the student's name near the top of the image.
- Modify the **World** class to display the student's name and information about the picture on the command-line screen.

6.3.1.1.4.2 Modifications to the SimpleTurtle class

The **SimpleTurtle** class was modified to change the order in which colors are assigned to new turtle objects and to change the initial heading of the turtle from north to northeast.

A complete listing of the modified **SimpleTurtle** class is shown in Listing 7 (p. 1413) near the end of the module.

Change the order of color assignment

Listing 2 (p. 1405) declares and initializes an array of color data that is used in the original version of the **SimpleTurtle** class to assign colors to the turtles on a cyclical basis as they are instantiated.

Listing 6.15: Original array containing turtle colors.

```
    /** array of colors to use for the turtles */
  private static Color[] colorArray =
    { Color.green,
      Color.cyan,
      new Color(204,0,204),
      Color.gray};
```

[21] http://cnx.org/content/m44330/latest/Lecture01.htm#Analysis

Listing 3 (p. 1406) declares and initializes a modified version of the array of color data that is used to assign colors to the turtles as they are instantiated.

Listing 6.16: Modified array of turtle colors.

```
    /** array of colors to use for the turtles */
private static Color[] colorArray =
  { Color.cyan,
    new Color(204,0,204),
    Color.green,
    Color.gray};
```

Determining which color to use

The code that assigns colors to the turtles as they are instantiated keeps track of the number of turtle objects that have been instantiated.

An index is computed as the

turtle count modulus the length of the array .

The colors are extracted from the array on a cyclical basis as more and more turtle objects are instantiated.

Each time the number of turtles is evenly divisible by the length of the array, the index used to access colors from the array starts over at zero.

Meeting the requirements

This modification to the **SimpleTurtle** class accomplished the following requirement established earlier under Analysis [22] .

- Modify the **SimpleTurtle** class to change the order in which colors are assigned to new turtles as they are instantiated.

Change the initial heading

Listing 4 (p. 1406) modifies the initialization value for a variable named **heading** , which is used to establish the direction that the turtle is facing.

Listing 6.17: Change the initial heading.

```
    /** heading angle */
//THIS IS A MODIFICATION
//private double heading = 0;//default faces north
private double heading = 45;// default faces northeast
```

The default direction in the original version of the class is due north or zero degrees. The modified default direction is northeast or 45 degrees.

Meeting the requirements

This modification to the **SimpleTurtle** class accomplished the following requirement established earlier under Analysis [23] .

- Modify the **SimpleTurtle** class to change the initial heading for new turtle objects to northeast instead of north.

That completes the required modifications that were established under Analysis [24] .

[22] http://cnx.org/content/m44330/latest/Lecture01.htm#Analysis
[23] http://cnx.org/content/m44330/latest/Lecture01.htm#Analysis
[24] http://cnx.org/content/m44330/latest/Lecture01.htm#Analysis

6.3.1.1.5 Run the program

I encourage you to copy the code from Listing 5 (p. 1407) , Listing 6 (p. 1408) , and Listing 7 (p. 1413) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Click Prob01.jpg [25] to download the required input image file.

6.3.1.1.6 Summary

You learned how to make simple modification to the **World** and **SimpleTurtle** classes that modify how a program that uses Ericson's library behaves.

6.3.1.1.7 What's next?

This module dealt with modifications to the **World** and **SimpleTurtle** Classes. The next module will deal with modifications to the **Turtle** and **SimpleTurtle** Classes.

6.3.1.1.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Modifying the World and SimpleTurtle Classes
- File: Java3102.htm
- Revised: 08/18/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.1.1.9 Complete program listings

Complete listings of the classes discussed in this module are shown in Listing 5 (p. 1407) , Listing 6 (p. 1408) , and Listing 7 (p. 1413) below.

Listing 6.18: Source code for driver class named Prob01.

[25] http://cnx.org/content/m44330/latest/Prob01.jpg

```
    public class Prob01{
  public static void main(String[] args){
    World mars = new World(200,250);
    Turtle joe = new Turtle(mars);
    joe.forward();
    Turtle bill = new Turtle(mars);
    bill.moveTo(50,125);
    Turtle sue = new Turtle(mars);
    sue.moveTo(150,125);
    Turtle tom = new Turtle(mars);
    tom.moveTo(100,225);
  }//end main method
}//end class Prob01
```

Listing 6.19: Source code for the modified World class.

```
    import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;


/*Note, this version of the World class was modified to
 * cause it to load a jpg file by default instead of
 * displaying a blank picture by default. 12/23/08
 */

/**
 * Class to represent a 2d world that can hold turtles and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements ModelDisplay
{
  ////////////////// fields /////////////////////

  /** should automatically repaint when model changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.WHITE;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;
```

```
/** the list of turtles in the world */
private List<Turtle> turtleList = new ArrayList<Turtle>();

/** the JFrame to show this world in */
private JFrame frame = new JFrame("World");

/** background picture */
private Picture picture = null;

////////////////// the constructors ///////////////

/**
 * Constructor that takes no arguments
 */
public World()
{
  // set up the world and make it visible
  initWorld(true);
}

/**
 * Constructor that takes a boolean to
 * say if this world should be visible
 * or not
 * @param visibleFlag if true will be visible
 * else if false will not be visible
 */
public World(boolean visibleFlag)
{
  initWorld(visibleFlag);
}

/**
 * Constructor that takes a width and height for this
 * world
 * @param w the width for the world
 * @param h the height for the world
 */
public World(int w, int h)
{
  width = w;
  height = h;

  System.out.println("Dick Baldwin");
  // set up the world and make it visible
  initWorld(true);
}

///////////////// methods /////////////////////////
```

```
  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new Dimension(width,height));

    // create the background picture
    //THIS IS A MODIFICATION
    //picture = new Picture(width,height);
    picture = new Picture("Prob01.jpg");
    picture.addMessage("Dick Baldwin",10,20);
    System.out.println(picture);

    // add this panel to the frame
    frame.getContentPane().add(this);

    // pack the frame
    frame.pack();

    // show this world
    frame.setVisible(visibleFlag);
  }

  /**
   * Method to get the graphics context for drawing on
   * @return the graphics context of the background picture
   */
  public Graphics getGraphics() { return picture.getGraphics(); }

  /**
   * Method to clear the background picture
   */
  public void clearBackground() { picture = new Picture(width,height); }

  /**
   * Method to get the background picture
   * @return the background picture
   */
  public Picture getPicture() { return picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) { picture = pict; }

  /**
```

```
 * Method to paint this component
 * @param g the graphics context
 */
public synchronized void paintComponent(Graphics g)
{
  Turtle turtle = null;

  // draw the background image
  g.drawImage(picture.getImage(),0,0,null);

  // loop drawing each turtle on the background image
  Iterator iterator = turtleList.iterator();
  while (iterator.hasNext())
  {
    turtle = (Turtle) iterator.next();
    turtle.paintComponent(g);
  }
}

/**
 * Metod to get the last turtle in this world
 * @return the last turtle added to this world
 */
public Turtle getLastTurtle()
{
  return (Turtle) turtleList.get(turtleList.size() - 1);
}


/**
 * Method to add a model to this model displayer
 * @param model the model object to add
 */
public void addModel(Object model)
{
  turtleList.add((Turtle) model);
  if (autoRepaint)
     repaint();
}

/**
 * Method to check if this world contains the passed
 * turtle
 * @return true if there else false
 */
public boolean containsTurtle(Turtle turtle)
{
  return (turtleList.contains(turtle));
}

/**
```

```
   * Method to remove the passed object from the world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }

  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) { autoRepaint = value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }
```

```
  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should show or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this world
   * in the form of a string
   * @return a string of information about this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " + getHeight() +
      " world with " + turtleList.size() + " turtles in it.";
  }

} // end of World class
```

Listing 6.20: Source code for the modified SimpleTurtle class.

```
    import javax.swing.*;
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.Observer;
import java.util.Random;

 /*Note: This class was modified to change the order of the
 * colors used for new turtle objects as well as the
 * initial heading for the turtle. 12/23/08
 */

/**
```

```
 *Class that represents a Logo-style turtle.  The turtle
 * starts off facing north.
 * A turtle can have a name, has a starting x and y position,
 * has a heading, has a width, has a height, has a visible
 * flag, has a body color, can have a shell color, and has a pen.
 * The turtle will not go beyond the model display or picture
 * boundaries.
 *
 * You can display this turtle in either a picture or in
 * a class that implements ModelDisplay.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimpleTurtle
{
  ///////////////// fields ////////////////////////

  /** count of the number of turtles created */
  private static int numTurtles = 0;

  /** array of colors to use for the turtles */
  //THIS IS A MODIFICATION
  //THE ORDER OF THE COLORS IN THE ARRAY HAS BEEN MODIFIED
  private static Color[] colorArray = { Color.cyan, new Color(204,0,204),Color.green, Color.gray};

  /** who to notify about changes to this turtle */
  private ModelDisplay modelDisplay = null;

  /** picture to draw this turtle on */
  private Picture picture = null;

  /** width of turtle in pixels */
  private int width = 15;

  /** height of turtle in pixels */
  private int height = 18;

  /** current location in x (center) */
  private int xPos = 0;

  /** current location in y (center) */
  private int yPos = 0;

  /** heading angle */
  //THIS IS A MODIFICATION
  //private double heading = 0;// default is facing north
  private double heading = 45;// default is facing northeast

  /** pen to use for this turtle */
  private Pen pen = new Pen();
```

```
/** color to draw the body in */
private Color bodyColor = null;

/** color to draw the shell in */
private Color shellColor = null;

/** color of information string */
private Color infoColor = Color.black;

/** flag to say if this turtle is visible */
private boolean visible = true;

/** flag to say if should show turtle info */
private boolean showInfo = false;

/** the name of this turtle */
private String name = "No name";

////////////////// constructors ////////////////////

/**
 * Constructor that takes the x and y position for the
 * turtle
 * @param x the x pos
 * @param y the y pos
 */
public SimpleTurtle(int x, int y)
{
  xPos = x;
  yPos = y;
  bodyColor = colorArray[numTurtles % colorArray.length];
  setPenColor(bodyColor);
  numTurtles++;
}

/**
 * Constructor that takes the x and y position and the
 * model displayer
 * @param x the x pos
 * @param y the y pos
 * @param display the model display
 */
public SimpleTurtle(int x, int y, ModelDisplay display)
{
  this(x,y); // invoke constructor that takes x and y
  modelDisplay = display;
  display.addModel(this);
}

/**
```

```
   * Constructor that takes a model display and adds
   * a turtle in the middle of it
   * @param display the model display
   */
  public SimpleTurtle(ModelDisplay display)
  {
    // invoke constructor that takes x and y
    this((int) (display.getWidth() / 2),
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);
    System.out.println("Dick Baldwin");
  }

  /**
   * Constructor that takes the x and y position and the
   * picture to draw on
   * @param x the x pos
   * @param y the y pos
   * @param picture the picture to draw on
   */
  public SimpleTurtle(int x, int y, Picture picture)
  {
    this(x,y); // invoke constructor that takes x and y
    this.picture = picture;
    this.visible = false; // default is not to see the turtle
  }

  /**
   * Constructor that takes the
   * picture to draw on and will appear in the middle
   * @param picture the picture to draw on
   */
  public SimpleTurtle(Picture picture)
  {
    // invoke constructor that takes x and y
    this((int) (picture.getWidth() / 2),
         (int) (picture.getHeight() / 2));
    this.picture = picture;
    this.visible = false; // default is not to see the turtle
  }

  ////////////////////// methods ////////////////////////

  /**
   * Get the distance from the passed x and y location
   * @param x the x location
   * @param y the y location
   */
  public double getDistance(int x, int y)
  {
```

```
  int xDiff = x - xPos;
  int yDiff = y - yPos;
  return (Math.sqrt((xDiff * xDiff) + (yDiff * yDiff)));
}

/**
 * Method to turn to face another simple turtle
 */
public void turnToFace(SimpleTurtle turtle)
{
  turnToFace(turtle.xPos,turtle.yPos);
}

 /**
 * Method to turn towards the given x and y
 * @param x the x to turn towards
 * @param y the y to turn towards
 */
public void turnToFace(int x, int y)
{
  double dx = x - this.xPos;
  double dy = y - this.yPos;
  double arcTan = 0.0;
  double angle = 0.0;

  // avoid a divide by 0
  if (dx == 0)
  {
    // if below the current turtle
    if (dy > 0)
      heading = 180;

    // if above the current turtle
    else if (dy < 0)
      heading = 0;
  }
  // dx isn't 0 so can divide by it
  else
  {
    arcTan = Math.toDegrees(Math.atan(dy / dx));
    if (dx < 0)
      heading = arcTan - 90;
    else
      heading = arcTan + 90;
  }

  // notify the display that we need to repaint
  updateDisplay();
}

/**
```

```
   * Method to get the picture for this simple turtle
   * @return the picture for this turtle (may be null)
   */
  public Picture getPicture() { return this.picture; }

  /**
   * Method to set the picture for this simple turtle
   * @param pict the picture to use
   */
  public void setPicture(Picture pict) { this.picture = pict; }

  /**
   * Method to get the model display for this simple turtle
   * @return the model display if there is one else null
   */
  public ModelDisplay getModelDisplay() { return this.modelDisplay; }

  /**
   * Method to set the model display for this simple turtle
   * @param theModelDisplay the model display to use
   */
  public void setModelDisplay(ModelDisplay theModelDisplay)
  { this.modelDisplay = theModelDisplay; }

  /**
   * Method to get value of show info
   * @return true if should show info, else false
   */
  public boolean getShowInfo() { return this.showInfo; }

  /**
   * Method to show the turtle information string
   * @param value the value to set showInfo to
   */
  public void setShowInfo(boolean value) { this.showInfo = value; }

  /**
   * Method to get the shell color
   * @return the shell color
   */
  public Color getShellColor()
  {
    Color color = null;
    if (this.shellColor == null && this.bodyColor != null)
      color = bodyColor.darker();
    else color = this.shellColor;
    return color;
  }

  /**
   * Method to set the shell color
```

```
 * @param color the color to use
 */
public void setShellColor(Color color) {  this.shellColor = color; }

/**
 * Method to get the body color
 * @return the body color
 */
public Color getBodyColor() { return this.bodyColor; }

/**
 * Method to set the body color which
 * will also set the pen color
 * @param color the color to use
 */
public void setBodyColor(Color color)
{
  this.bodyColor = color;
  setPenColor(this.bodyColor);
}

/**
 * Method to set the color of the turtle.
 * This will set the body color
 * @param color the color to use
 */
public void setColor(Color color) { this.setBodyColor(color); }

/**
 * Method to get the information color
 * @return the color of the information string
 */
public Color getInfoColor() { return this.infoColor; }

/**
 * Method to set the information color
 * @param color the new color to use
 */
public void setInfoColor(Color color) { this.infoColor = color; }

/**
 * Method to return the width of this object
 * @return the width in pixels
 */
public int getWidth() { return this.width; }

/**
 * Method to return the height of this object
 * @return the height in pixels
 */
public int getHeight() { return this.height; }
```

```
/**
 * Method to set the width of this object
 * @param theWidth in width in pixels
 */
public void setWidth(int theWidth) { this.width = theWidth; }

/**
 * Method to set the height of this object
 * @param theHeight the height in pixels
 */
public void setHeight(int theHeight) { this.height = theHeight; }

/**
 * Method to get the current x position
 * @return the x position (in pixels)
 */
public int getXPos() { return this.xPos; }

/**
 * Method to get the current y position
 * @return the y position (in pixels)
 */
public int getYPos() { return this.yPos; }

/**
 * Method to get the pen
 * @return the pen
 */
public Pen getPen() { return this.pen; }

/**
 * Method to set the pen
 * @param thePen the new pen to use
 */
public void setPen(Pen thePen) { this.pen = thePen; }

/**
 * Method to check if the pen is down
 * @return true if down else false
 */
public boolean isPenDown() { return this.pen.isPenDown(); }

/**
 * Method to set the pen down boolean variable
 * @param value the value to set it to
 */
public void setPenDown(boolean value) { this.pen.setPenDown(value); }

/**
 * Method to lift the pen up
```

```
 */
public void penUp() { this.pen.setPenDown(false);}

/**
 * Method to set the pen down
 */
public void penDown() { this.pen.setPenDown(true);}

/**
 * Method to get the pen color
 * @return the pen color
 */
public Color getPenColor() { return this.pen.getColor(); }

/**
 * Method to set the pen color
 * @param color the color for the pen ink
 */
public void setPenColor(Color color) { this.pen.setColor(color); }

/**
 * Method to set the pen width
 * @param width the width to use in pixels
 */
public void setPenWidth(int width) { this.pen.setWidth(width); }

/**
 * Method to get the pen width
 * @return the width of the pen in pixels
 */
public int getPenWidth() { return this.pen.getWidth(); }

/**
 * Method to clear the path (history of
 * where the turtle has been)
 */
public void clearPath()
{
  this.pen.clearPath();
}

/**
 * Method to get the current heading
 * @return the heading in degrees
 */
public double getHeading() { return this.heading; }

/**
 * Method to set the heading
 * @param heading the new heading to use
 */
```

```
  public void setHeading(double heading)
  {
    this.heading = heading;
  }

  /**
   * Method to get the name of the turtle
   * @return the name of this turtle
   */
  public String getName() { return this.name; }

  /**
   * Method to set the name of the turtle
   * @param theName the new name to use
   */
  public void setName(String theName)
  {
    this.name = theName;
  }

  /**
   * Method to get the value of the visible flag
   * @return true if visible else false
   */
  public boolean isVisible() { return this.visible;}

  /**
   * Method to hide the turtle (stop showing it)
   * This doesn't affect the pen status
   */
  public void hide() { this.setVisible(false); }

  /**
   * Method to show the turtle (doesn't affect
   * the pen status
   */
  public void show() { this.setVisible(true); }

  /**
   * Method to set the visible flag
   * @param value the value to set it to
   */
  public void setVisible(boolean value)
  {
    // if the turtle wasn't visible and now is
    if (visible == false && value == true)
    {
      // update the display
      this.updateDisplay();
    }
```

```
  // set the visibile flag to the passed value
  this.visible = value;
}

/**
 * Method to update the display of this turtle and
 * also check that the turtle is in the bounds
 */
public synchronized void updateDisplay()
{
  // check that x and y are at least 0
  if (xPos < 0)
    xPos = 0;
  if (yPos < 0)
    yPos = 0;

  // if picture
  if (picture != null)
  {
    if (xPos >= picture.getWidth())
      xPos = picture.getWidth() - 1;
    if (yPos >= picture.getHeight())
      yPos = picture.getHeight() - 1;
    Graphics g = picture.getGraphics();
    paintComponent(g);
  }
  else if (modelDisplay != null)
  {
    if (xPos >= modelDisplay.getWidth())
      xPos = modelDisplay.getWidth() - 1;
    if (yPos >= modelDisplay.getHeight())
      yPos = modelDisplay.getHeight() - 1;
    modelDisplay.modelChanged();
  }
}

/**
 * Method to move the turtle foward 100 pixels
 */
public void forward() { forward(100); }

/**
 * Method to move the turtle forward the given number of pixels
 * @param pixels the number of pixels to walk forward in the heading direction
 */
public void forward(int pixels)
{
  int oldX = xPos;
  int oldY = yPos;

  // change the current position
```

```
    xPos = oldX + (int) (pixels * Math.sin(Math.toRadians(heading)));
    yPos = oldY + (int) (pixels * -Math.cos(Math.toRadians(heading)));

    // add a move from the old position to the new position to the pen
    pen.addMove(oldX,oldY,xPos,yPos);

    // update the display to show the new line
    updateDisplay();
  }

  /**
   * Method to go backward by 100 pixels
   */
  public void backward()
  {
    backward(100);
  }

  /**
   * Method to go backward a given number of pixels
   * @param pixels the number of pixels to walk backward
   */
  public void backward(int pixels)
  {
    forward(-pixels);
  }

  /**
   * Method to move to turtle to the given x and y location
   * @param x the x value to move to
   * @param y the y value to move to
   */
  public void moveTo(int x, int y)
  {
    this.pen.addMove(xPos,yPos,x,y);
    this.xPos = x;
    this.yPos = y;
    this.updateDisplay();
  }

  /**
   * Method to turn left
   */
  public void turnLeft()
  {
   this.turn(-90);
  }

  /**
   * Method to turn right
   */
```

```
public void turnRight()
{
  this.turn(90);
}

/**
 * Method to turn the turtle the passed degrees
 * use negative to turn left and pos to turn right
 * @param degrees the amount to turn in degrees
 */
public void turn(int degrees)
{
  this.heading = (heading + degrees) % 360;
  this.updateDisplay();
}

/**
 * Method to draw a passed picture at the current turtle
 * location and rotation in a picture or model display
 * @param dropPicture the picture to drop
 */
public synchronized void drop(Picture dropPicture)
{
  Graphics2D g2 = null;

  // only do this if drawing on a picture
  if (picture != null)
    g2 = (Graphics2D) picture.getGraphics();
  else if (modelDisplay != null)
    g2 = (Graphics2D) modelDisplay.getGraphics();

  // if g2 isn't null
  if (g2 != null)
  {

    // save the current tranform
    AffineTransform oldTransform = g2.getTransform();

    // rotate to turtle heading and translate to xPos and yPos
    g2.rotate(Math.toRadians(heading),xPos,yPos);

    // draw the passed picture
    g2.drawImage(dropPicture.getImage(),xPos,yPos,null);

    // reset the tranformation matrix
    g2.setTransform(oldTransform);

    //  draw the pen
    pen.paintComponent(g2);
  }
}
```

```
/**
 * Method to paint the turtle
 * @param g the graphics context to paint on
 */
public synchronized void paintComponent(Graphics g)
{
  // cast to 2d object
  Graphics2D g2 = (Graphics2D) g;

  // if the turtle is visible
  if (visible)
  {
    // save the current tranform
    AffineTransform oldTransform = g2.getTransform();

    // rotate the turtle and translate to xPos and yPos
    g2.rotate(Math.toRadians(heading),xPos,yPos);

    // determine the half width and height of the shell
    int halfWidth = (int) (width/2); // of shell
    int halfHeight = (int) (height/2); // of shell
    int quarterWidth = (int) (width/4); // of shell
    int thirdHeight = (int) (height/3); // of shell
    int thirdWidth = (int) (width/3); // of shell

    // draw the body parts (head)
    g2.setColor(bodyColor);
    g2.fillOval(xPos - quarterWidth,
                yPos - halfHeight - (int) (height/3),
                halfWidth, thirdHeight);
    g2.fillOval(xPos - (2 * thirdWidth),
                yPos - thirdHeight,
                thirdWidth,thirdHeight);
    g2.fillOval(xPos - (int) (1.6 * thirdWidth),
                yPos + thirdHeight,
                thirdWidth,thirdHeight);
    g2.fillOval(xPos + (int) (1.3 * thirdWidth),
                yPos - thirdHeight,
                thirdWidth,thirdHeight);
    g2.fillOval(xPos + (int) (0.9 * thirdWidth),
                yPos + thirdHeight,
                thirdWidth,thirdHeight);


    // draw the shell
    g2.setColor(getShellColor());
    g2.fillOval(xPos - halfWidth,
                yPos - halfHeight, width, height);

    // draw the info string if the flag is true
```

```
      if (showInfo)
        drawInfoString(g2);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);
    }

    //  draw the pen
    pen.paintComponent(g);
  }

  /**
   * Method to draw the information string
   * @param g the graphics context
   */
  public synchronized void drawInfoString(Graphics g)
  {
    g.setColor(infoColor);
    g.drawString(this.toString(),xPos + (int) (width/2),yPos);
  }

  /**
   * Method to return a string with informaiton
   * about this turtle
   * @return a string with information about this object
   */
  public String toString()
  {
    return this.name + " turtle at " + this.xPos + ", " +
      this.yPos + " heading " + this.heading + ".";
  }

} // end of class
```

zz

zz

zz

zz

NOTE: **zz-Title of note:** zz-Body of note

zz

TemplateForATable
PutTableDataHere

Table 6.1

zz

zz

-end-

6.3.1.2 Java OOP: Modifications to the Turtle and SimpleTurtle Classes[26]

6.3.1.2.1 Table of Contents

6.3.1.2.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [27] .

6.3.1.2.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.1.2.2.1.1 Figures

6.3.1.2.2.1.2 Listings

[26] This content is available online at <http://cnx.org/content/m44348/1.1/>.
[27] http://cnx.org/content/m44148/latest/

6.3.1.2.3 Preview

Program specifications

Write a program named **Prob02** that uses the class definition shown in Listing 4 (p. 1433) and Ericson's media library to produce the graphic output image shown in Figure 1 (p. 1429) .

Required screen output.

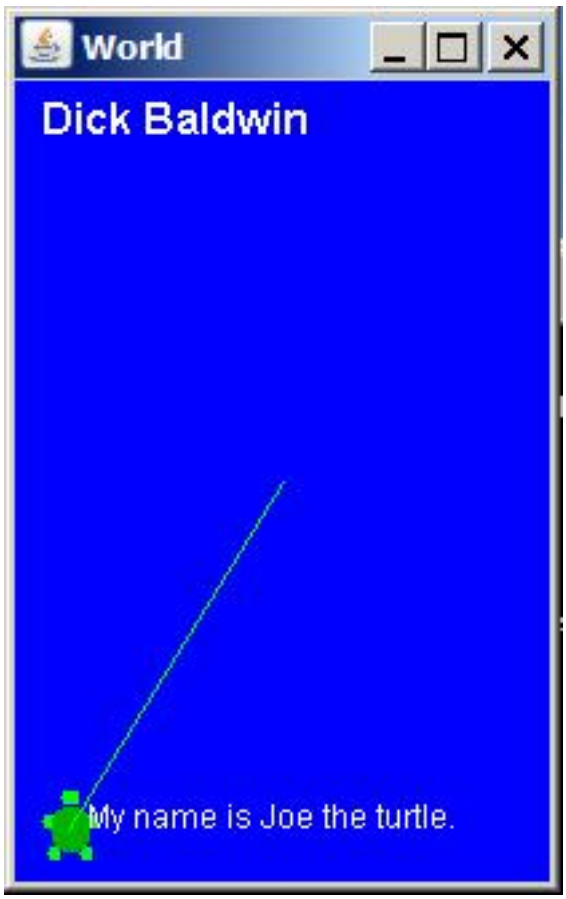


Figure 6.5: Required screen output.

No new classes allowed

You may not define any new classes to cause your program to behave as required, and you may not modify the class definition for the class named **Prob02** given in Listing 4 (p. 1433) .

Files in your folder

You must copy and modify *(as necessary)* the media classes named **Turtle.java** and **SimpleTurtle.java** from Ericson's library to cause your program to produce the required output.

Your folder must contain only the following class files and source-code files:

- Prob02.class

- Prob02.java
- SimpleTurtle.class
- SimpleTurtle.java
- Turtle.class
- Turtle.java

Output text

In addition to the output image described above, your program must produce the text output shown in Figure 2 (p. 1430) on the command- line screen

---

Required text output.

```
    Dick Baldwin
My name is Joe the turtle.
```

Figure 6.6: Required text output.

Required modifications

By comparing the default behavior of the **Turtle** and **SimpleTurtle** classes with the requirements of this program, it can be determined that the following modifications to the **Turtle** and **SimpleTurtle** classes are required to meet the specifications.

1. Modify the **Turtle** class to cause the student's name to be displayed on the command line.
2. Modify the **Turtle** and **SimpleTurtle** classes to accept and save a **String** parameter in addition to the **World** parameter when the Turtle object is constructed.
3. Modify the **SimpleTurtle** class to cause the default background of the world to be BLUE.
4. Modify the **SimpleTurtle** class to cause the student's name to be displayed near the top of the **World** image.
5. Modify the **toString** method in the **SimpleTurtle** class to cause it to return the value of the **String** parameter whenever the **toString** method is called. This causes the **drawInfoString** method to display the string in place of its normal behavior. It also causes the last statement in Listing 4 (p. 1433) to display the turtle's name.

6.3.1.2.4 Discussion and sample code

6.3.1.2.4.1 Modifications to the Turtle class

Ericson's **Turtle** class was modified according to the first two items listed above under required modifications [28] .

A complete listing of the modified **Turtle** class is provided in Listing 5 (p. 1433) near the end of the module.

Modification to the Turtle constructor

[28] http://cnx.org/content/m44348/latest/Lecture02.htm#Required_modifications

The **Turtle** class has several overloaded constructors. One of the constructors was modified to accept a **String** parameter in addition to the **World** parameter and pass the new parameter along to the superclass constructor. The code is shown in Listing 1 (p. 1431) .

Listing 6.21: Modified Turtle constructor.

```
  public Turtle (ModelDisplay modelDisplay,
                 String turtleName){
  // let the parent constructor handle it
  super(modelDisplay,turtleName);
  System.out.println("Dick Baldwin");
}
```

A **println** statement was also added to the modified constructor to cause it to display the student's name on the command line screen when the **Turtle** object is constructed as shown in Figure 2.

6.3.1.2.4.2 Modifications to the SimpleTurtle class

A complete listing of the modified **SimpleTurtle** class is shown in Listing 6 (p. 1435) near the end of the module.

The superclass of the Turtle class

The **SimpleTurtle** class is the superclass of the **Turtle** class. Therefore, the **SimpleTurtle** class must be modified to accept the **String** parameter passed to the superclass in Listing 1 (p. 1431) . This was accomplished by modifying one of the constructors of the **SimpleTurtle** class as shown in Listing 2 (p. 1431) .

Listing 6.22: Modified SimpleTurtle constructor.

```
  String turtleName = null;

public SimpleTurtle(ModelDisplay display,
                    String turtleName){

  // call constructor that takes x and y
  this((int) (display.getWidth() / 2),
       (int) (display.getHeight() / 2));
  modelDisplay = display;
  display.addModel(this);

  //THIS IS THE MODIFICATION
  this.turtleName = turtleName;
  Picture picture = ((World)(display)).getPicture();
  picture.setAllPixelsToAColor(Color.BLUE);
  picture.addMessage("Dick Baldwin",10,20);
}
```

The modification is shown in the last four statements in Listing 2 (p. 1431) . This modification satisfies items 2, 3, and 4 listed earlier under required modifications [29] .

Modified toString method

[29] http://cnx.org/content/m44348/latest/Lecture02.htm#Required_modifications

Listing 3 (p. 1432) shows the modified **toString** method that satisfies item 5 listed above under required modifications [30] .

Listing 6.23: Modified toString method.

```
  public String toString(){
//return this.name + " turtle at " + this.xPos + ", " +
//      this.yPos + " heading " + this.heading + ".";
return "My name is " + turtleName + " the turtle.";
}//end toString
```

The original code was preserved as comments in Listing 3 (p. 1432) , and the new modified code is shown below those comments.

6.3.1.2.5 Run the program

I encourage you to copy the code from Listing 4 (p. 1433) , Listing 5 (p. 1433) , and Listing 6 (p. 1435) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.1.2.6 Summary

You learned how to:

1. Modify the **Turtle** class to cause the student's name to be displayed on the command line.
2. Modify the **Turtle** and **SimpleTurtle** classes to accept and save a **String** parameter in addition to the **World** parameter when the **Turtle** object is constructed.
3. Modify the **SimpleTurtle** class to cause the default background of the world to be BLUE.
4. Modify the **SimpleTurtle** class to cause the student's name to be displayed near the top of the **World** image.
5. Modify the **toString** method in the **SimpleTurtle** class to cause it to return the value of the **String** parameter whenever the **toString** method is called. This causes the **drawInfoString** method to display the string in place of its normal behavior. It also causes the last statement in Listing 4 (p. 1433) to display the turtle's name.

6.3.1.2.7 What's next?

In the next module, you will learn how to incorporate GUI components into a **World** object. In particular, you will learn how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

6.3.1.2.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Modifications to the Turtle and SimpleTurtle Classes
- File: Java3104.htm
- Revised: 08/19/12

[30] http://cnx.org/content/m44348/latest/Lecture02.htm#Required_modifications

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.1.2.9 Complete program listing

Complete listings of the programs discussed in this module are shown below.

Listing 6.24: Source code for the class named Prob02.

```
    import java.awt.Color;

public class Prob02{
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars,"Joe");
    joe.moveTo(20,280);
    joe.setInfoColor(Color.WHITE);
    joe.setShowInfo(true);
    System.out.println(joe);
  }//end main method
}//end class Prob02
```

Listing 6.25: Modified Turtle class.

```
    /*12/23/0812/23/08 This class and the class named
SimpleTurtle were modified to:

Accept and save a String parameter in addition to the
World parameter when the Turtle object is constructed.

Modify the toString method to cause it to return the
value of the String parameter whenever the toString
method is called. This causes the drawInfoString method
to display the string in place of its normal behavior.

Cause the default background of the world to be BLUE.

Cause the student's name to be displayed near the top of
```

```
the World image.

Cause the student's name as well as the turtle's name to
be displayed on the command line.
*/

/**
 * Class that represents a turtle which is similar to a Logo turtle.
 * This class inherts from SimpleTurtle and is for students
 * to add methods to.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Turtle extends SimpleTurtle
{
  ////////////////// constructors ///////////////////////

  /** Constructor that takes the x and y and a picture to
   * draw on
   * @param x the starting x position
   * @param y the starting y position
   * @param picture the picture to draw on
   */
  public Turtle (int x, int y, Picture picture)
  {
    // let the parent constructor handle it
    super(x,y,picture);
  }

  /** Constructor that takes the x and y and a model
   * display to draw it on
   * @param x the starting x position
   * @param y the starting y position
   * @param modelDisplayer the thing that displays the model
   */
  public Turtle (int x, int y,
                 ModelDisplay modelDisplayer)
  {
    // let the parent constructor handle it
    super(x,y,modelDisplayer);
  }

  //THIS IS A MODIFICATION
  //The following constructor was modified to accept and
  // save a String parameter and pass it to the superclass
  // constructor.
  /** Constructor that takes the model display
   * @param modelDisplay the thing that displays the model
   */
  public Turtle (ModelDisplay modelDisplay,
```

```
                String turtleName){
    // let the parent constructor handle it
    super(modelDisplay,turtleName);
    System.out.println("Dick Baldwin");
  }

  /**
   * Constructor that takes a picture to draw on
   * @param p the picture to draw on
   */
  public Turtle (Picture p)
  {
    // let the parent constructor handle it
    super(p);
  }

  ////////////////// methods ///////////////////////


} // this } is the end of class Turtle, put all new methods before this
```

Listing 6.26: Modified SimpleTurtle class.

```
    import javax.swing.*;
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.Observer;
import java.util.Random;

/*12/23/08 This class and the class named Turtle were
modified to:
Accept and save a String parameter in addition to the
World parameter when the Turtle object is constructed.

Modify the toString method to cause it to return the
value of the String parameter whenever the toString
method is called. This causes the drawInfoString method
to display the string in place of its normal behavior.

Cause the default background of the world to be BLUE.

Cause the student's name to be displayed near the top of
the World image.

Cause the student's name as well as the turtle's name to
be displayed on the command line.
 */

/**
 * Class that represents a Logo-style turtle.  The turtle
```

```
 * starts off facing north.
 * A turtle can have a name, has a starting x and y position,
 * has a heading, has a width, has a height, has a visible
 * flag, has a body color, can have a shell color, and has a pen.
 * The turtle will not go beyond the model display or picture
 * boundaries.
 *
 * You can display this turtle in either a picture or in
 * a class that implements ModelDisplay.
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimpleTurtle
{
  ///////////////// fields ////////////////////////

  /** count of the number of turtles created */
  private static int numTurtles = 0;

  /** array of colors to use for the turtles */
  private static Color[] colorArray = { Color.green, Color.cyan, new Color(204,0,204), Color.gray};

  /** who to notify about changes to this turtle */
  private ModelDisplay modelDisplay = null;

  /** picture to draw this turtle on */
  private Picture picture = null;

  /** width of turtle in pixels */
  private int width = 15;

  /** height of turtle in pixels */
  private int height = 18;

  /** current location in x (center) */
  private int xPos = 0;

  /** current location in y (center) */
  private int yPos = 0;

  /** heading angle */
  private double heading = 0;  // default is facing north

  /** pen to use for this turtle */
  private Pen pen = new Pen();

  /** color to draw the body in */
  private Color bodyColor = null;

  /** color to draw the shell in */
```

```
private Color shellColor = null;

/** color of information string */
private Color infoColor = Color.black;

/** flag to say if this turtle is visible */
private boolean visible = true;

/** flag to say if should show turtle info */
private boolean showInfo = false;

/** the name of this turtle */
private String name = "No name";

////////////////// constructors /////////////////

/**
 * Constructor that takes the x and y position for the
 * turtle
 * @param x the x pos
 * @param y the y pos
 */
public SimpleTurtle(int x, int y)
{
  xPos = x;
  yPos = y;
  bodyColor = colorArray[numTurtles % colorArray.length];
  setPenColor(bodyColor);
  numTurtles++;
}

/**
 * Constructor that takes the x and y position and the
 * model displayer
 * @param x the x pos
 * @param y the y pos
 * @param display the model display
 */
public SimpleTurtle(int x, int y, ModelDisplay display)
{
  this(x,y); // call constructor that takes x and y
  modelDisplay = display;
  display.addModel(this);
}
//THIS IS A MODIFICATION
//The following constructor was modified to accept and
// save a String parameter.
String turtleName = null;
/**
 * Constructor that takes a model display and adds
 * a turtle in the middle of it
```

```
   * @param display the model display
   */
  public SimpleTurtle(ModelDisplay display,
                      String turtleName){

    // call constructor that takes x and y
    this((int) (display.getWidth() / 2),
         (int) (display.getHeight() / 2));
    modelDisplay = display;
    display.addModel(this);
    this.turtleName = turtleName;
    Picture picture = ((World)(display)).getPicture();
    //THIS IS A MODIFICATION
    picture.setAllPixelsToAColor(Color.BLUE);
    picture.addMessage("Dick Baldwin",10,20);
  }

  /**
   * Constructor that takes the x and y position and the
   * picture to draw on
   * @param x the x pos
   * @param y the y pos
   * @param picture the picture to draw on
   */
  public SimpleTurtle(int x, int y, Picture picture)
  {
    this(x,y); // call constructor that takes x and y
    this.picture = picture;
    this.visible = false; // default is not to see the turtle
  }

  /**
   * Constructor that takes the
   * picture to draw on and will appear in the middle
   * @param picture the picture to draw on
   */
  public SimpleTurtle(Picture picture)
  {
    // call constructor that takes x and y
    this((int) (picture.getWidth() / 2),
         (int) (picture.getHeight() / 2));
    this.picture = picture;
    this.visible = false; // default is not to see the turtle
  }

  ////////////////////// methods ///////////////////////

  /**
   * Get the distance from the passed x and y location
   * @param x the x location
   * @param y the y location
```

```
 */
public double getDistance(int x, int y)
{
  int xDiff = x - xPos;
  int yDiff = y - yPos;
  return (Math.sqrt((xDiff * xDiff) + (yDiff * yDiff)));
}

/**
 * Method to turn to face another simple turtle
 */
public void turnToFace(SimpleTurtle turtle)
{
  turnToFace(turtle.xPos,turtle.yPos);
}

 /**
 * Method to turn towards the given x and y
 * @param x the x to turn towards
 * @param y the y to turn towards
 */
public void turnToFace(int x, int y)
{
  double dx = x - this.xPos;
  double dy = y - this.yPos;
  double arcTan = 0.0;
  double angle = 0.0;

  // avoid a divide by 0
  if (dx == 0)
  {
    // if below the current turtle
    if (dy > 0)
      heading = 180;

    // if above the current turtle
    else if (dy < 0)
      heading = 0;
  }
  // dx isn't 0 so can divide by it
  else
  {
    arcTan = Math.toDegrees(Math.atan(dy / dx));
    if (dx < 0)
      heading = arcTan - 90;
    else
      heading = arcTan + 90;
  }

  // notify the display that we need to repaint
  updateDisplay();
```

```
}

/**
 * Method to get the picture for this simple turtle
 * @return the picture for this turtle (may be null)
 */
public Picture getPicture() { return this.picture; }

/**
 * Method to set the picture for this simple turtle
 * @param pict the picture to use
 */
public void setPicture(Picture pict) { this.picture = pict; }

/**
 * Method to get the model display for this simple turtle
 * @return the model display if there is one else null
 */
public ModelDisplay getModelDisplay() { return this.modelDisplay; }

/**
 * Method to set the model display for this simple turtle
 * @param theModelDisplay the model display to use
 */
public void setModelDisplay(ModelDisplay theModelDisplay)
{ this.modelDisplay = theModelDisplay; }

/**
 * Method to get value of show info
 * @return true if should show info, else false
 */
public boolean getShowInfo() { return this.showInfo; }

/**
 * Method to show the turtle information string
 * @param value the value to set showInfo to
 */
public void setShowInfo(boolean value) { this.showInfo = value; }

/**
 * Method to get the shell color
 * @return the shell color
 */
public Color getShellColor()
{
  Color color = null;
  if (this.shellColor == null && this.bodyColor != null)
    color = bodyColor.darker();
  else color = this.shellColor;
  return color;
}
```

```
/**
 * Method to set the shell color
 * @param color the color to use
 */
public void setShellColor(Color color) {  this.shellColor = color; }

/**
 * Method to get the body color
 * @return the body color
 */
public Color getBodyColor() { return this.bodyColor; }

/**
 * Method to set the body color which
 * will also set the pen color
 * @param color the color to use
 */
public void setBodyColor(Color color)
{
  this.bodyColor = color;
  setPenColor(this.bodyColor);
}

/**
 * Method to set the color of the turtle.
 * This will set the body color
 * @param color the color to use
 */
public void setColor(Color color) { this.setBodyColor(color); }

/**
 * Method to get the information color
 * @return the color of the information string
 */
public Color getInfoColor() { return this.infoColor; }

/**
 * Method to set the information color
 * @param color the new color to use
 */
public void setInfoColor(Color color) { this.infoColor = color; }

/**
 * Method to return the width of this object
 * @return the width in pixels
 */
public int getWidth() { return this.width; }

/**
 * Method to return the height of this object
```

```
   * @return the height in pixels
   */
  public int getHeight() { return this.height; }

  /**
   * Method to set the width of this object
   * @param theWidth in width in pixels
   */
  public void setWidth(int theWidth) { this.width = theWidth; }

  /**
   * Method to set the height of this object
   * @param theHeight the height in pixels
   */
  public void setHeight(int theHeight) { this.height = theHeight; }

  /**
   * Method to get the current x position
   * @return the x position (in pixels)
   */
  public int getXPos() { return this.xPos; }

  /**
   * Method to get the current y position
   * @return the y position (in pixels)
   */
  public int getYPos() { return this.yPos; }

  /**
   * Method to get the pen
   * @return the pen
   */
  public Pen getPen() { return this.pen; }

  /**
   * Method to set the pen
   * @param thePen the new pen to use
   */
  public void setPen(Pen thePen) { this.pen = thePen; }

  /**
   * Method to check if the pen is down
   * @return true if down else false
   */
  public boolean isPenDown() { return this.pen.isPenDown(); }

  /**
   * Method to set the pen down boolean variable
   * @param value the value to set it to
   */
  public void setPenDown(boolean value) { this.pen.setPenDown(value); }
```

```
/**
 * Method to lift the pen up
 */
public void penUp() { this.pen.setPenDown(false);}

/**
 * Method to set the pen down
 */
public void penDown() { this.pen.setPenDown(true);}

/**
 * Method to get the pen color
 * @return the pen color
 */
public Color getPenColor() { return this.pen.getColor(); }

/**
 * Method to set the pen color
 * @param color the color for the pen ink
 */
public void setPenColor(Color color) { this.pen.setColor(color); }

/**
 * Method to set the pen width
 * @param width the width to use in pixels
 */
public void setPenWidth(int width) { this.pen.setWidth(width); }

/**
 * Method to get the pen width
 * @return the width of the pen in pixels
 */
public int getPenWidth() { return this.pen.getWidth(); }

/**
 * Method to clear the path (history of
 * where the turtle has been)
 */
public void clearPath()
{
  this.pen.clearPath();
}

/**
 * Method to get the current heading
 * @return the heading in degrees
 */
public double getHeading() { return this.heading; }

/**
```

```
   * Method to set the heading
   * @param heading the new heading to use
   */
  public void setHeading(double heading)
  {
    this.heading = heading;
  }

  /**
   * Method to get the name of the turtle
   * @return the name of this turtle
   */
  public String getName() { return this.name; }

  /**
   * Method to set the name of the turtle
   * @param theName the new name to use
   */
  public void setName(String theName)
  {
    this.name = theName;
  }

  /**
   * Method to get the value of the visible flag
   * @return true if visible else false
   */
  public boolean isVisible() { return this.visible;}

  /**
   * Method to hide the turtle (stop showing it)
   * This doesn't affect the pen status
   */
  public void hide() { this.setVisible(false); }

  /**
   * Method to show the turtle (doesn't affect
   * the pen status
   */
  public void show() { this.setVisible(true); }

  /**
   * Method to set the visible flag
   * @param value the value to set it to
   */
  public void setVisible(boolean value)
  {
    // if the turtle wasn't visible and now is
    if (visible == false && value == true)
    {
      // update the display
```

```
    this.updateDisplay();
  }

  // set the visibile flag to the passed value
  this.visible = value;
}

/**
 * Method to update the display of this turtle and
 * also check that the turtle is in the bounds
 */
public synchronized void updateDisplay()
{
  // check that x and y are at least 0
  if (xPos < 0)
    xPos = 0;
  if (yPos < 0)
    yPos = 0;

  // if picture
  if (picture != null)
  {
    if (xPos >= picture.getWidth())
      xPos = picture.getWidth() - 1;
    if (yPos >= picture.getHeight())
      yPos = picture.getHeight() - 1;
    Graphics g = picture.getGraphics();
    paintComponent(g);
  }
  else if (modelDisplay != null)
  {
    if (xPos >= modelDisplay.getWidth())
      xPos = modelDisplay.getWidth() - 1;
    if (yPos >= modelDisplay.getHeight())
      yPos = modelDisplay.getHeight() - 1;
    modelDisplay.modelChanged();
  }
}

/**
 * Method to move the turtle foward 100 pixels
 */
public void forward() { forward(100); }

/**
 * Method to move the turtle forward the given number of pixels
 * @param pixels the number of pixels to walk forward in the heading direction
 */
public void forward(int pixels)
{
  int oldX = xPos;
```

```
    int oldY = yPos;

    // change the current position
    xPos = oldX + (int) (pixels * Math.sin(Math.toRadians(heading)));
    yPos = oldY + (int) (pixels * -Math.cos(Math.toRadians(heading)));

    // add a move from the old position to the new position to the pen
    pen.addMove(oldX,oldY,xPos,yPos);

    // update the display to show the new line
    updateDisplay();
  }

  /**
   * Method to go backward by 100 pixels
   */
  public void backward()
  {
    backward(100);
  }

  /**
   * Method to go backward a given number of pixels
   * @param pixels the number of pixels to walk backward
   */
  public void backward(int pixels)
  {
    forward(-pixels);
  }

  /**
   * Method to move to turtle to the given x and y location
   * @param x the x value to move to
   * @param y the y value to move to
   */
  public void moveTo(int x, int y)
  {
    this.pen.addMove(xPos,yPos,x,y);
    this.xPos = x;
    this.yPos = y;
    this.updateDisplay();
  }

  /**
   * Method to turn left
   */
  public void turnLeft()
  {
   this.turn(-90);
  }
```

```
/**
 * Method to turn right
 */
public void turnRight()
{
  this.turn(90);
}

/**
 * Method to turn the turtle the passed degrees
 * use negative to turn left and pos to turn right
 * @param degrees the amount to turn in degrees
 */
public void turn(int degrees)
{
  this.heading = (heading + degrees) % 360;
  this.updateDisplay();
}

/**
 * Method to draw a passed picture at the current turtle
 * location and rotation in a picture or model display
 * @param dropPicture the picture to drop
 */
public synchronized void drop(Picture dropPicture)
{
  Graphics2D g2 = null;

  // only do this if drawing on a picture
  if (picture != null)
    g2 = (Graphics2D) picture.getGraphics();
  else if (modelDisplay != null)
    g2 = (Graphics2D) modelDisplay.getGraphics();

  // if g2 isn't null
  if (g2 != null)
  {

    // save the current tranform
    AffineTransform oldTransform = g2.getTransform();

    // rotate to turtle heading and translate to xPos and yPos
    g2.rotate(Math.toRadians(heading),xPos,yPos);

    // draw the passed picture
    g2.drawImage(dropPicture.getImage(),xPos,yPos,null);

    // reset the tranformation matrix
    g2.setTransform(oldTransform);

    //  draw the pen
```

```
      pen.paintComponent(g2);
    }
  }

  /**
   * Method to paint the turtle
   * @param g the graphics context to paint on
   */
  public synchronized void paintComponent(Graphics g)
  {
    // cast to 2d object
    Graphics2D g2 = (Graphics2D) g;

    // if the turtle is visible
    if (visible)
    {
      // save the current tranform
      AffineTransform oldTransform = g2.getTransform();

      // rotate the turtle and translate to xPos and yPos
      g2.rotate(Math.toRadians(heading),xPos,yPos);

      // determine the half width and height of the shell
      int halfWidth = (int) (width/2); // of shell
      int halfHeight = (int) (height/2); // of shell
      int quarterWidth = (int) (width/4); // of shell
      int thirdHeight = (int) (height/3); // of shell
      int thirdWidth = (int) (width/3); // of shell

      // draw the body parts (head)
      g2.setColor(bodyColor);
      g2.fillOval(xPos - quarterWidth,
                  yPos - halfHeight - (int) (height/3),
                  halfWidth, thirdHeight);
      g2.fillOval(xPos - (2 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos - (int) (1.6 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (1.3 * thirdWidth),
                  yPos - thirdHeight,
                  thirdWidth,thirdHeight);
      g2.fillOval(xPos + (int) (0.9 * thirdWidth),
                  yPos + thirdHeight,
                  thirdWidth,thirdHeight);


      // draw the shell
      g2.setColor(getShellColor());
      g2.fillOval(xPos - halfWidth,
```

```
                yPos - halfHeight, width, height);

      // draw the info string if the flag is true
      if (showInfo)
        drawInfoString(g2);

      // reset the tranformation matrix
      g2.setTransform(oldTransform);
    }

    //  draw the pen
    pen.paintComponent(g);
  }

  /**
   * Method to draw the information string
   * @param g the graphics context
   */
  public synchronized void drawInfoString(Graphics g)
  {
    g.setColor(infoColor);
    g.drawString(this.toString(),xPos + (int) (width/2),yPos);
  }
  //This toString method was modified.
  /**
   * Method to return a string with informaiton
   * about this turtle
   * @return a string with information about this object
   */
  //THIS IS A MODIFICATION
  //MODIFIED toString METHOD
  public String toString()
  {
//    return this.name + " turtle at " + this.xPos + ", " +
//      this.yPos + " heading " + this.heading + ".";
    return "My name is " + turtleName + " the turtle.";
  }

} // end of class
```

-end-

6.3.1.3 Java OOP: Incorporating GUI Components into a World Object[31]

6.3.1.3.1 Table of Contents

[31] This content is available online at <http://cnx.org/content/m44350/1.1/>.

6.3.1.3.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [32] .

6.3.1.3.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.1.3.2.1.1 Figures

6.3.1.3.2.1.2 Listings

6.3.1.3.3 Preview

In this module, you will learn how to incorporate GUI components into a **World** object. In particular, you will learn how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

Program specifications

Write a program named **Prob03** that uses the **Prob03** class definition shown in Listing 2 (p. 1454) and Ericson's media library to produce the graphic output images shown in Figure 1 (p. 1451) and Figure 2 (p. 1452) .

[32] http://cnx.org/content/m44148/latest/

The image shown in Figure 1 (p. 1451) is the image that appears on the screen when the program starts running.

Initial screen output.

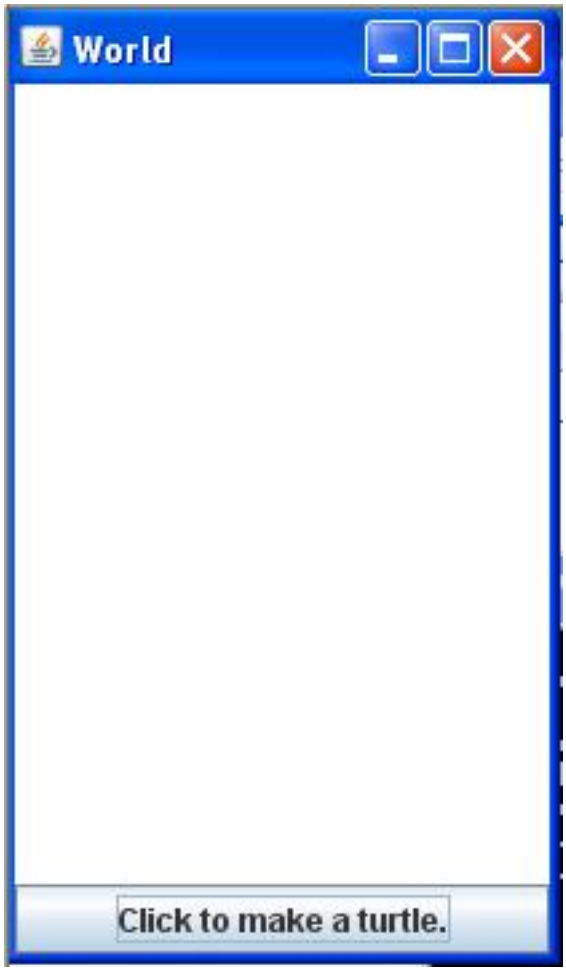


Figure 6.7: Initial screen output.

Click the button

The image shown in Figure 2 (p. 1452) is what you should see when you click the button at the bottom of the world.

Screen output after clicking the button.

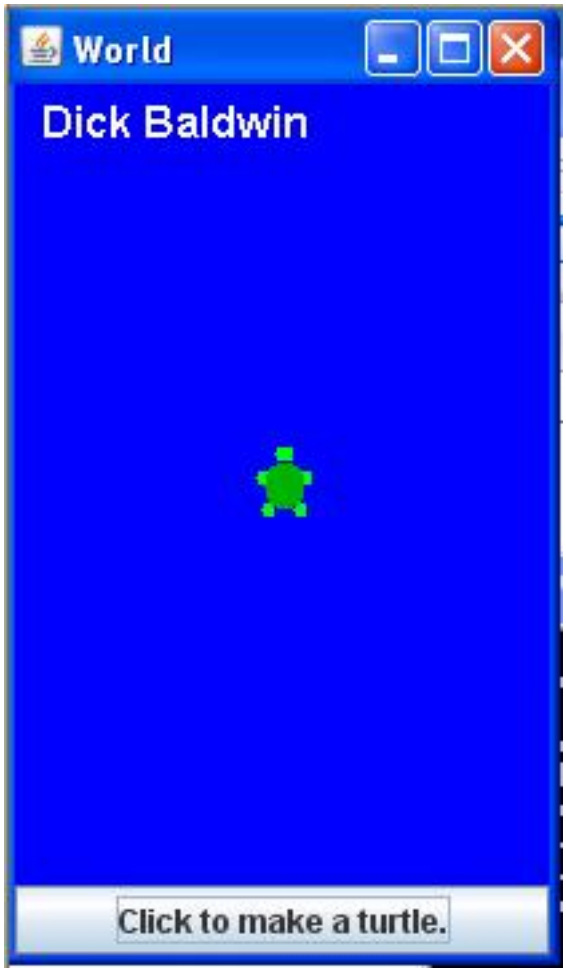


Figure 6.8: Screen output after clicking the button.

Modify Ericson's World class

You must copy and modify *(as necessary)* the media class named **World** to cause your program to produce the required output with the required behavior.

Add a JButton to the World

This program adds a **JButton** object to the SOUTH location of the **World** object as shown in Figure 1 (p. 1451) and Figure 2 (p. 1452) and registers an action listener on the button to control the behavior of the program.

Program behavior

The program initially displays an empty white world as shown in Figure 1 (p. 1451) . When the user clicks the button, the world's background color changes to blue, a turtle appears in the center of the **World** , and the student's name appears near the top of the world.

Output text

In addition to the output images described above, your program must produce the text output shown in Figure 3 (p. 1453) on the command- line screen

---

Required text output.

```
    Dick Baldwin
Dick Baldwin
Picture, filename None height 300 width 200
```

Figure 6.9: Required text output.

Analysis

A **World** object is actually a specialized use of a standard Java **JFrame** object. However, by default, the frame is not available to users of the **World** class. Therefore, in order to satisfy the requirements of this program, the **World** class must be modified to provide access to the frame.

Add a getFrame method

This program adds a method named **getFrame** to the **World** class. The **getFrame** method returns a reference to the **JFrame** object that is used to display the world. This makes it possible to treat **World** objects in much the same way that other **JFrame** objects are treated.

Add a button and pack the frame

The program uses the **JFrame** object's reference to add a **JButton** object to the SOUTH location of the **JFrame** . After adding the button, the program calls the **pack** method on the frame to cause the size of the frame to be automatically adjusted to accommodate both the **Picture** object that constitutes the background and the **JButton** object.

6.3.1.3.4 Discussion and sample code

6.3.1.3.4.1 Modification to the World class

A complete listing of the modified **World** class is provided in Listing 10 (p. 1459) near the end of the module

Getting access to the frame

An object of the **World** class contains a private instance variable named **frame** that contains a reference to the **JFrame** object. Because it is private, however, it is not available to users of the **World** class. The *getter* method shown in Listing 1 (p. 1453) was added to the **World** class to provide access to the **JFrame** .

Listing 6.27: The new getFrame method.

```
  public JFrame getFrame(){
 System.out.println("Dick Baldwin");
 return frame;
}//end getFrame
```

The new method also displays the student's name when the method is called, producing part of the text output in Figure 3 (p. 1453) .

No other change to is required

This is the only change to Ericson's library that is required to write this program. Everything else in the program makes use of existing library classes with no modifications.

6.3.1.3.4.2 The Prob03Runner class

Will explain in fragments

I will explain this program in fragments. A complete listing is shown in Listing 9 (p. 1458) near the end of the module.

The driver class

The driver class for this program is named **Prob03** . The definition of the driver class is shown in its entirety in Listing 2 (p. 1454) .

Listing 6.28: Prob03 class definition.

```
    import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03
```

The driver class simply instantiates a new object of a class named **Prob03Runner** and calls a method named **run** on that object.

Beginning of the Prob03Runner class

The beginning of the **Prob03Runner** class and the constructor for the class is shown in Listing 3 (p. 1454) .

Listing 6.29: Beginning of the Prob03Runner class.

```
    class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Dick Baldwin");
  }//end constructor
```

As you can see, the constructor simply displays the student's name, providing some of the text output shown in Figure 3 (p. 1454) .

Beginning of the run method

The beginning of the **run** method is shown in Listing 4 (p. 1454) . This is where things start to get interesting.

Listing 6.30: Beginning of the run method.

```
  public void run(){
//This reference must be final because it is
// referenced from within an anonymous class
// definition.
final World world = new World(200,300);

//Get a reference to the JFrame object that is used
// to display the World.
JFrame frame = world.getFrame();

//Instantiate a new JButton object and add it to the
// SOUTH location in the JFrame object.
JButton button = new JButton(
                           "Click to make a turtle.");
frame.getContentPane().add(button,BorderLayout.SOUTH);


frame.pack();//VERY IMPORTANT
```

A new World object

Listing 4 (p. 1454) begins by instantiating a new object of the **World** class with a size of 200x300 pixels. The reference to the object is saved in a **final** variable named **world** .

A final reference variable

As the comment indicates, the variable must be **final** because it is referenced from within an anonymous class definition. I won't take the time to explain that here. I will simply refer you to my website where I have published several tutorial modules on anonymous classes.

The size of the world...

The purpose of specifying the size of the world when it is instantiated is to implicitly specify the size of the **Picture** object that forms the background for the world.

The size of the picture actually matches the specified dimensions. Therefore the actual size of the world is a little larger than the specified dimensions due to the borders that surround the picture.

Get a reference to the frame

After the **World** object is instantiated, the new **getFrame** method is called on the world's reference in order to get and save a reference to the frame.

A new JButton object

Then a new **JButton** object is instantiated. The reference to the **JButton** object is saved in the variable named **button** .

Add the button to the frame and pack it

Then the button is added to the SOUTH location in the frame and the **pack** method is called on the frame. Calling the **pack** method causes the size of the frame, *(and hence the size of the world)* to be adjusted so as to accommodate the picture in the CENTER of the frame and the button at the bottom *(SOUTH)* of the frame.

The final size of the world

After the button is added and the world is packed, the overall height of the world is quite a bit larger than the original dimensions. I measured it and found it to be about 209x361 pixels including borders.

The expansion in height is necessary to make room for the button. However, as you can see in Figure 3 (p. 1453) , the size of the picture remains at 200x300 pixels.

Register an action listener on the button

I elected to use an *anonymous class* to register an action listener on the button. The purpose of the listener is to produce the desired behavior when the button is clicked.

Note, however, that there are other ways to register an action listener on the button and the student is not required to use an anonymous class for that purpose.

Beginning of the anonymous class

The definition of the anonymous listener class and the instantiation of the listener object begins in Listing 5 (p. 1456) .

Listing 6.31: Register an action listener on the button.

```
    button.addActionListener(new ActionListener()
  {//Begin the class definition

    public void actionPerformed(ActionEvent e){
      Picture picture = world.getPicture();
      System.out.println(picture);
```

Unfamiliar with anonymous classes?

If you are unfamiliar with anonymous classes and action listeners, I will simply refer you to my website where I have published several tutorial modules on the topic. I have also published modules on the topic in this collection.

In a nutshell...

In a nutshell, however, the method named **actionPerformed** , which begins in Listing 5 (p. 1456) , will be executed each time the user clicks the button in Figure 1 (p. 1451) .

Behavior of the actionPerformed method

The code in Listing 5 (p. 1456) gets and saves a reference to the **Picture** object that forms the background in the world object. Then it passes a copy of that reference to the **println** method, producing the third line of output text shown in Figure 3 (p. 1453) .

Set the background picture to blue

Listing 6 (p. 1456) calls the method named **setAllPixelsToAColor** on the **Picture** object passing the color BLUE as a parameter.

Listing 6.32: Set the picture background to blue.

```
      picture.setAllPixelsToAColor(Color.BLUE);
```

As you might expect, this causes the background of the world to turn from white to blue as shown in Figure 2 (p. 1452) .

Display the student's name on the picture

Listing 7 (p. 1456) calls the **addMessage** method on the picture to add the student's name near the upper-left corner of the world. *(See Figure 2 (p. 1452) .)*

Listing 6.33: Display the student's name on the picture.

```
      picture.addMessage("Dick Baldwin",10,20);
```

Finally, Listing 8 (p. 1456) instantiates a new **Turtle** object in the default color, with the default heading *(north)* , located in the default position, which is the center of the picture that constitutes the background image for the world.

Listing 6.34: Add a turtle to the world.

```
      Turtle turtle = new Turtle(world);
    }//end actionPerformed

  }//end class definition
```

```
    );//end addActionListener

  }//end run
  //-----------------------------------------------------//
}//end class Prob03Runner
```

Multiple clicks

If you click the button more than once, you will instantiate a new **Turtle** object and produce a line of output text on the command line screen with each click. The turtles will all be in the same location but they will cycle through four different color schemes.

Note that adding the turtle to the world causes the world to be repainted, eliminating the requirement to purposely repaint the world.

End the run method

Finally, Listing 8 (p. 1456) signals the end of the **run** method, causing the **run** method to terminate and return control to the **main** method in Listing 2 (p. 1454) . The **main** method terminates causing the program to terminate.

6.3.1.3.5 Run the program

I encourage you to copy the code from Listing 9 (p. 1458) and Listing 10 (p. 1459) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.1.3.6 Summary

In this module, you learned how to incorporate GUI components into a **World** object. In particular, you learned how to add a **JButton** object to a **World** object and register an action listener on the button to control the behavior of the program.

6.3.1.3.7 What's next?

In the next module, you will learn how to modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object. Then you will place a turtle object in a world and perform a series of maneuvers causing the turtle to draw a square spiral.

6.3.1.3.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Incorporating GUI Components into a World Object
- File: Java3106.htm
- Revised: 08/19/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.1.3.9 Complete program listing

Complete listings of the programs discussed in this module are provided below.

Listing 6.35: Source code for the program named Prob03.

```
    import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

public class Prob03{
  public static void main(String[] args){
    new Prob03Runner().run();
  }//end main method
}//end class Prob03


//=======================================================//
class Prob03Runner{
  public Prob03Runner(){
    System.out.println("Dick Baldwin");
  }//end constructor
  //-------------------------------------------------------//
  public void run(){
    //This reference must be final because it is
    // referenced from within an anonymous class
    // definition.
    final World world = new World(200,300);

    //Get a reference to the JFrame object that is used
    // to display the World.
    JFrame frame = world.getFrame();

    //Instantiate a new JButton object and add it to the
    // SOUTH location in the JFrame object.
    JButton button = new JButton(
                               "Click to make a turtle.");
    frame.getContentPane().add(button,BorderLayout.SOUTH);


    frame.pack();//VERY IMPORTANT
```

```
    //Use an anonymous class to register an action
    // listener on the button. Note that the student is
    // not required to use an anonymous class.
    button.addActionListener(new ActionListener()
      {//Begin the class definition
        public void actionPerformed(ActionEvent e){
          Picture picture = world.getPicture();
          System.out.println(picture);
          //Set picture background to blue.
          picture.setAllPixelsToAColor(Color.BLUE);
          //Display the student's name on the picture.
          picture.addMessage(
                        "Dick Baldwin",10,20);
          //Add a turtle to the world. This causes the
          // world to be repainted.
          Turtle turtle = new Turtle(world);
        }//end actionPerformed
      }//end class definition
    );//end addActionListener

  }//end run
  //-----------------------------------------------//
}//end class Prob03Runner
```

Listing 6.36: The modified World class.

```
    import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;

/*12/23/08 Modified the World class. Added a method named
 *getFrame that returns a reference to the JFrame object
 *in which the turtles are displayed.
 */

/**
 * Class to represent a 2d world that can hold turtles and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements ModelDisplay
{
  ////////////////// fields ////////////////////////

  /** should automatically repaint when model changed */
```

```
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.white;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;

  /** the list of turtles in the world */
  private List<Turtle> turtleList = new ArrayList<Turtle>();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors ///////////////

  /**
   * Constructor that takes no arguments
   */
  public World()
  {
    // set up the world and make it visible
    initWorld(true);
  }

  /**
   * Constructor that takes a boolean to
   * say if this world should be visible
   * or not
   * @param visibleFlag if true will be visible
   * else if false will not be visible
   */
  public World(boolean visibleFlag)
  {
    initWorld(visibleFlag);
  }

  /**
   * Constructor that takes a width and height for this
   * world
   * @param w the width for the world
   * @param h the height for the world
   */
  public World(int w, int h)
  {
```

```
  width = w;
  height = h;

  // set up the world and make it visible
  initWorld(true);
}

///////////////// methods /////////////////////////
/**
 *Method to return a reference to the JFrame.
 */
public JFrame getFrame(){
  System.out.println("Dick Baldwin");
  return frame;
}//end getFrame

/**
 * Method to initialize the world
 * @param visibleFlag the flag to make the world
 * visible or not
 */
private void initWorld(boolean visibleFlag)
{
  // set the preferred size
  this.setPreferredSize(new Dimension(width,height));

  // create the background picture
  picture = new Picture(width,height);

  // add this panel to the frame
  frame.getContentPane().add(this);

  // pack the frame
  frame.pack();

  // show this world
  frame.setVisible(visibleFlag);
}

/**
 * Method to get the graphics context for drawing on
 * @return the graphics context of the background picture
 */
public Graphics getGraphics() { return picture.getGraphics(); }

/**
 * Method to clear the background picture
 */
public void clearBackground() { picture = new Picture(width,height); }

/**
```

```
   * Method to get the background picture
   * @return the background picture
   */
  public Picture getPicture() { return picture; }

  /**
   * Method to set the background picture
   * @param pict the background picture to use
   */
  public void setPicture(Picture pict) { picture = pict; }

  /**
   * Method to paint this component
   * @param g the graphics context
   */
  public synchronized void paintComponent(Graphics g)
  {
    Turtle turtle = null;

    // draw the background image
    g.drawImage(picture.getImage(),0,0,null);

    // loop drawing each turtle on the background image
    Iterator iterator = turtleList.iterator();
    while (iterator.hasNext())
    {
      turtle = (Turtle) iterator.next();
      turtle.paintComponent(g);
    }
  }

  /**
   * Metod to get the last turtle in this world
   * @return the last turtle added to this world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle) turtleList.get(turtleList.size() - 1);
  }


  /**
   * Method to add a model to this model displayer
   * @param model the model object to add
   */
  public void addModel(Object model)
  {
    turtleList.add((Turtle) model);
    if (autoRepaint)
       repaint();
  }
```

```
  /**
   * Method to check if this world contains the passed
   * turtle
   * @return true if there else false
   */
  public boolean containsTurtle(Turtle turtle)
  {
    return (turtleList.contains(turtle));
  }

  /**
   * Method to remove the passed object from the world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }

  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) { autoRepaint = value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
```

```
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }

  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should show or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}

  /**
   * Method to get an iterator on the list of turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this world
   * in the form of a string
   * @return a string of information about this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " + getHeight() +
      " world with " + turtleList.size() + " turtles in it.";
  }

} // end of World class
```

-end-

6.3.1.4 Java OOP: Background Color, Text Color, Mouse Clicks, etc.[33]

6.3.1.4.1 Table of Contents

6.3.1.4.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [34] .

6.3.1.4.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

####### 6.3.1.4.2.1.1 Figures

####### 6.3.1.4.2.1.2 Listings

[33] This content is available online at <http://cnx.org/content/m44351/1.1/>.
[34] http://cnx.org/content/m44148/latest/

6.3.1.4.3 Preview

The program that I will explain in this module requires you to modify both the **World** class and the **SimplePicture** class from Ericson's media library.

Just as you did in an earlier module, you will modify the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object.

You will modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object.

Program specifications

Write a program named **Prob04** that uses the class definition for the **Prob04** class shown in Listing 9 (p. 1475) along with Ericson's media library to produce the graphic output images shown in Figure 1 (p. 1466) , Figure 2 (p. 1467) , and Figure 3 (p. 1468) .

Output image at startup

Figure 1 (p. 1466) shows the output image when you first start the program.

Output image at startup.

Figure 6.10: Output image at startup.

Output image after ten mouse clicks

This program adds a **JButton** object to the SOUTH location of the **World** object as shown in Figure 1 (p. 1466) . Figure 2 (p. 1467) shows the output image after you click button ten times.

Output image after ten mouse clicks.

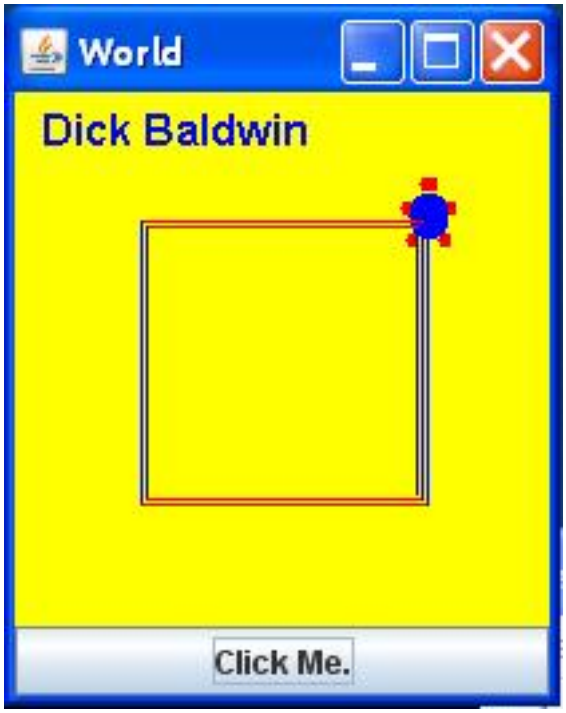


Figure 6.11: Output image after ten mouse clicks.

Output image after eleven mouse clicks

Figure 3 (p. 1468) shows the output image after you click button eleven times.

Output image after eleven mouse clicks.

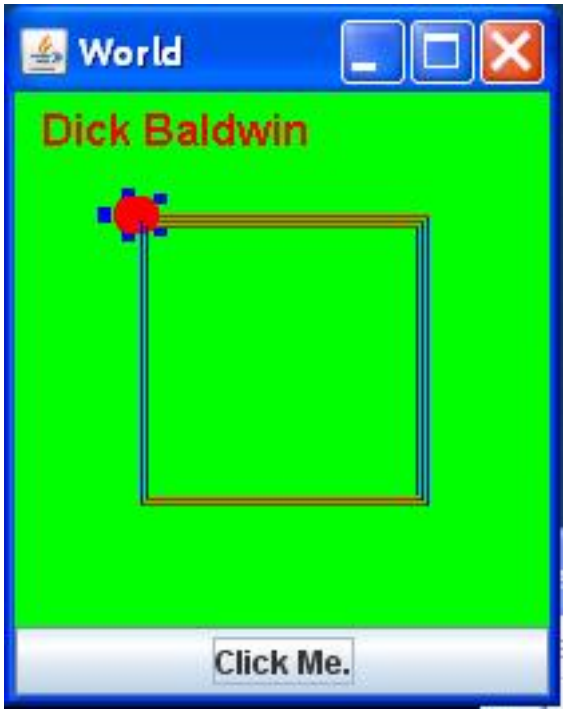


Figure 6.12: Output image after eleven mouse clicks.

Operational description

The program initially displays an empty white world with a button at the bottom as shown in Figure 1 (p. 1466) .

When the user clicks the button:

- The world's background color changes to green.
- A turtle appears near the bottom right corner of the World.
- The student's name appears near the top left corner of the world in red.
- The turtle has a blue body and a red shell.

Click the button again

When you click the button the second time:

- The background color changes to yellow.
- The student's name changes to blue.
- The turtle changes to a red body with a blue shell.
- The turtle turns 90 degrees left and moves forward 100 pixels plus the value of a click counter.
- The turtle leaves a blue trail.

Click the button another time

On the next click:

- The colors revert to the color scheme with the yellow background.
- The turtle turns 90 degrees left and moves forward 100 pixels plus the value of the click counter leaving a red trail.

Click the button repeatedly

This cycle repeats on each click with the turtle's trail drawing a square spiral of increasing size with red lines on the top and bottom of the spiral and blue lines on the right and left of the spiral.

Output text

In addition to the output images described above, your program must produce some output text on the command- line screen

6.3.1.4.4 Discussion and sample code

The driver class

The driver class named **Prob04** is shown at the beginning of Listing 9 (p. 1475) near the end of the module. The **main** method simply instantiates a new object of the class named **Prob04Runner** , which I will explain later. The event driven behavior of the program is controlled by a listener object that is registered on the button in the constructor of the **Prob04Runner** class.

Modification of the World class

This program adds a method named **getFrame** to the **World** class. The method returns a reference to the **JFrame** object that is used to display the world.

The program uses that reference to add a **JButton** object to the SOUTH location of the **World** . I explained a modification very similar to this in an earlier module, so I won't repeat that explanation here.

A complete listing of the modified **World** class is provided in Listing 10 (p. 1476) near the end of the module.

Modification of the SimplePicture class

This program modifies the **addMessage** method of the **SimplePicture** class to cause it to use a color variable named **messageColor** to set the color of the text. The modification also declares and initializes the private instance variable named **messageColor** .

The **SimplePicture** class was also modified to include a **setMessageColor** method that can be used to set the color value stored in the variable named **messageColor** .

The new setMessageColor method

The new method named **setMessageColor** that was added to the **SimplePicture** class is shown in Listing 1 (p. 1469) .

Listing 6.37: The new setMessageColor method.

```
  public void setMessageColor(Color color){
 System.out.println("Dick Baldwin");
 messageColor = color;
}//end setMessageColor
```

Save the color value

The **setMessageColor** method saves the incoming color parameter in a private instance variable named **messageColor** that was added to the class.

The default value of the variable is **Color.WHITE** , thereby preserving the default behavior of the **addMessage** method.

You can view the new variable named **messageColor** in Listing 11 (p. 1482) near the end of the module.

Display the student's name

The new **setMessageColor** method also causes my name to be displayed each time the method is called. This is of no operational value, but is useful during the testing stage of the modified class. This is part of the code that produces text output on the command line screen.

The modified addMessage method

The modified version of the **addMessage** method is shown in Listing 2 (p. 1470) .

Listing 6.38: The modified addMessage method.

```
  public void addMessage(
                  String message, int xPos, int yPos){
// get a graphics context to use to draw on the
// buffered image
Graphics2D graphics2d = bufferedImage.createGraphics();

// set the color to white
//graphics2d.setPaint(Color.white);

//modified by Baldwin on 12/23/08
graphics2d.setPaint(messageColor);

// set the font to Helvetica bold style and size 16
graphics2d.setFont(new Font("Helvetica",Font.BOLD,16));

// draw the message
graphics2d.drawString(message,xPos,yPos);

}//end addMessage
```

The original statement shown in Listing 2 (p. 1470) was disabled and replaced by the statement shown following the modification comment. This causes the text to be displayed on the image using the color stored in the new private instance variable named **messageColor** .

The Prob04Runner class

I will explain this program in fragments. A complete listing is shown in Listing 9 (p. 1475) near the end of the module.

The driver class

The driver class for this program is named **Prob04** . As I mentioned earlier, you can view the class definition in its entirety near the beginning of Listing 9 (p. 1475) .

Beginning of the Prob04Runner class

The **Prob04Runner** class begins in Listing 3 (p. 1470) .

Listing 6.39: Beginning of the Prob04Runner class.

```
  class Prob04Runner{
Turtle turtle = null;
Picture picture = null;
int counter = 0;
World world = new World(200,200);
JButton button = new JButton("Click Me.");

public Prob04Runner(){
  System.out.println("Dick Baldwin");
  System.out.println(world.getPicture());

  //Get a reference to the JFrame object that is used
  // to display the World.
```

```
JFrame frame = world.getFrame();

//Add the JButton object to the
// SOUTH location in the JFrame object.
frame.getContentPane().add(button,BorderLayout.SOUTH);

frame.pack();
```

Very familiar code

You should already be familiar with all of the code in Listing 3 (p. 1470) . When the code in Listing 3 (p. 1470) has finished executing, the image shown in Figure 1 (p. 1466) should have appeared on the screen.

Waiting for an event

At this point, the program is essentially idle waiting for the user to either click the button at the bottom of Figure 1 (p. 1466) , or click the red X-button in the upper-right corner of Figure 1 (p. 1466) . *(More on the red X-button later.)*

Beginning of the anonymous listener class

This program uses an anonymous inner class to register an action listener on the button. The anonymous class begins in Listing 4 (p. 1471) .

Listing 6.40: Beginning of the anonymous listener class.

```
  button.addActionListener(new ActionListener()
{//Begin the anonymous class definition

  public void actionPerformed(ActionEvent e){
    picture = world.getPicture();

    //Set picture background to green.
    picture.setAllPixelsToAColor(Color.GREEN);
    picture.setMessageColor(Color.RED);


    //Display the student's name on the picture.
    picture.addMessage("Dick Baldwin",10,20);
```

The event handler method named actionPerformed

Once the listener object is instantiated from the anonymous class and registered on the button, the method named **actionPerformed** , which begins in Listing 4 (p. 1471) , will be executed each time the button is clicked.

Get a reference to the background picture

The **actionPerformed** method begins by getting and saving a reference to the **Picture** object that provides the background image for the **World** object. By default, all of the pixels in this image are white, as shown in Figure 1 (p. 1466) .

Set the background color to green

Then the method calls Ericson's standard method named **setAllPixelsToAColor** method to set the color of all of the background pixels to green.

Display student's name in red

Following that, the method calls the new **setMessageColor** method to set the text color to red, and calls the modified **addMessage** method to display my name in red near the upper-left corner of the image. Figure 3 (p. 1468) shows an example of a green background and red text.

Add a turtle to the world

Listing 5 (p. 1472) tests to determine if the variable named **turtle** that was declared in Listing 3 (p. 1470) still contains null. If so, that means that this is the first time that the button has been clicked and the **Turtle** object has not yet been added to the world.

Listing 6.41: Add a turtle to the world.

```
  if(turtle == null){
turtle = new Turtle(150,150,world);
turtle.setHeading(90);
turtle.setShellColor(Color.RED);
turtle.setBodyColor(Color.BLUE);
```

Add a Turtle object and set its colors

Listing 5 (p. 1472) instantiates a new **Turtle** object and adds it to the world near the lower-right corner facing due east *(90 degrees)* .

The shell color is set to red and the body color *(feet and head)* is set to blue. An example of the turtle with this color scheme is shown in Figure 3 (p. 1468) .

Not the first click

If the conditional clause in Listing 5 (p. 1472) returns false, that means that this is not the first time the button has been clicked and the else clause, which begins in Listing 6 (p. 1472) will be executed.

Listing 6.42: Not the first click.

```
  }else{
turtle.turnLeft();
turtle.forward(100+counter);
```

Rotate and move

The **else** clause begins by causing the turtle to rotate to the left by a default angle of 90 degrees. Then the turtle moves forward by a distance equal to 100 plus the value of a counter that is incremented by one each time the button is clicked.

For example, on the second click of the button, the turtle moves toward the north drawing a blue line along the way. The default width of the line is one pixel and the default color of the line is the same as the shell color.

Process odd or even clicks

The behavior of the **actionPerformed** method at this point depends on whether the incremented value of the **counter** variable is even or odd. The code to accomplish this is shown in Listing 7 (p. 1472) .

Listing 6.43: Process odd or even clicks.

```
  if(counter++ %2 != 0){
  picture.setAllPixelsToAColor(Color.GREEN);
  picture.setMessageColor(Color.RED);

  picture.addMessage(
            "Dick Baldwin",10,20);
  turtle.setShellColor(Color.RED);
  turtle.setBodyColor(Color.BLUE);
}else{
  picture.setAllPixelsToAColor(Color.YELLOW);
  picture.setMessageColor(Color.BLUE);
```

```
            picture.addMessage(
                       "Dick Baldwin",10,20);
            turtle.setShellColor(Color.BLUE);
            turtle.setBodyColor(Color.RED);
          }//end else

          picture.addMessage(
                       "Dick Baldwin",10,20);
        }//end else

      }//end actionPerformed
    }//end class definition
  );//end addActionListener

}//end constructor
//-----------------------------------------------------//

}//end class Prob04Runner
```

If the counter value is odd...

If the value of the counter (*before it is incremented – note the post-increment operator)* is odd, the color scheme for the background, the message, and the turtle is set to that shown in Figure 3 (p. 1468) with the green background.

If the counter value is even...

Otherwise, if the counter value is even, the color scheme is set to that shown in Figure 2 (p. 1467) with the yellow background.

The end of several sections of code

Listing 7 (p. 1472) also completes the **else** clause that began in Listing 6 (p. 1472) .

In addition, Listing 7 (p. 1472) signals the end of the **actionPerformed** method, the end of the anonymous class definition, the end of the constructor, and the end of the class named **Prob04Runner** .

Waiting for an event

As mentioned earlier, once the constructor finishes execution, the program becomes idle waiting for the user to either click the button at the bottom of Figure 1 (p. 1466) , or click the red X-button in the upper-right corner of Figure 1 (p. 1466) .

Does not terminate as expected

Normally a user would expect the program to terminate and return control to the operating system when the user clicks the red X-button in the upper-right corner of the last remaining window. However, this program does not do that. Instead, clicking this button simply hides the window and control is not returned to the operating system.

A programming oversight

This was a programming oversight on my part, which can be corrected by adding the second statement in Listing 8 (p. 1473) to the definition of the **Prob04Runner** class immediately following the first statement shown in Listing 8 (p. 1473) .

Listing 6.44: Cause the program to terminate properly.

```
    JFrame frame = world.getFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

To understand why this is necessary to cause the program to terminate, I recommend that you visit the standard Sun javadocs and examine the description of the method named **setDefaultCloseOperation** in

the **JFrame** class.

6.3.1.4.5 Run the program

I encourage you to copy the code from Listing 9 (p. 1475) , Listing 10 (p. 1476) , and Listing 11 (p. 1482) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.1.4.6 Summary

Just as you did in an earlier module, you modified the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object.

You learned how to modify the **SimplePicture** class to make it possible to control the color of the text that is placed on the image in a **Picture** object.

Then you placed a turtle object in a world and performed a series of maneuvers causing the turtle to draw a square spiral.

6.3.1.4.7 What's next?

In the next module, you will learn how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

6.3.1.4.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Background Color, Text Color, Mouse Clicks, etc.
- File: Java3108.htm
- Revised: 08/20/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.1.4.9 Complete program listings

Complete listings of the programs discussed in this module are shown below.

Listing 6.45: Source code for the program named Prob04.

```
    /*File Prob04 Copyright 2008 R.G.Baldwin
**********************************************************/
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import java.awt.Toolkit;

public class Prob04{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob04Runner();
  }//end main method
}//end class Prob04
//========================================================//

class Prob04Runner{
  Turtle turtle = null;
  Picture picture = null;
  int counter = 0;
  World world = new World(200,200);
  JButton button = new JButton("Click Me.");

  public Prob04Runner(){
    System.out.println("Dick Baldwin");
    System.out.println(world.getPicture());

    //Get a reference to the JFrame object that is used
    // to display the World.
    JFrame frame = world.getFrame();

    //Add the JButton object to the
    // SOUTH location in the JFrame object.
    frame.getContentPane().add(button,BorderLayout.SOUTH);

    frame.pack();

    //Use an anonymous class to register an action
    // listener on the button. Note that the student is
    // not required to use an anonymous class.
    button.addActionListener(new ActionListener()
      {//Begin the class definition
        public void actionPerformed(ActionEvent e){
          picture = world.getPicture();
```

```
          //Set picture background to green.
          picture.setAllPixelsToAColor(Color.GREEN);
          picture.setMessageColor(Color.RED);

          //Display the student's name on the picture.
          picture.addMessage(
                        "Dick Baldwin",10,20);
          //Add a turtle to the world. This causes the
          // world to be repainted.
          if(turtle == null){
            turtle = new Turtle(150,150,world);
            turtle.setHeading(90);
            turtle.setShellColor(Color.RED);
            turtle.setBodyColor(Color.BLUE);
          }else{
            turtle.turnLeft();
            turtle.forward(100+counter);
            if(counter++ %2 != 0){
              picture.setAllPixelsToAColor(Color.GREEN);
              picture.setMessageColor(Color.RED);

              picture.addMessage(
                        "Dick Baldwin",10,20);
              turtle.setShellColor(Color.RED);
              turtle.setBodyColor(Color.BLUE);
            }else{
              picture.setAllPixelsToAColor(Color.YELLOW);
              picture.setMessageColor(Color.BLUE);


              picture.addMessage(
                        "Dick Baldwin",10,20);
              turtle.setShellColor(Color.BLUE);
              turtle.setBodyColor(Color.RED);
            }//end else
            picture.addMessage(
                        "Dick Baldwin",10,20);
          }//end else
        }//end actionPerformed
      }//end class definition
    );//end addActionListener

  }//end constructor
  //-----------------------------------------------------//

}//end class Prob04Runner
```

Listing 6.46: Modified World class.

```
import javax.swing.*;
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observer;
import java.awt.*;

/*12/23/08 Modified the World class. Added a method named
 *getFrame that returns a reference to the JFrame object
 *in which the turtles are displayed.
 */

/**
 * Class to represent a 2d world that can hold turtles and
 * display them
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class World extends JComponent implements ModelDisplay
{
  ////////////////// fields /////////////////////

  /** should automatically repaint when model changed */
  private boolean autoRepaint = true;

  /** the background color for the world */
  private Color background = Color.white;

  /** the width of the world */
  private int width = 640;

  /** the height of the world */
  private int height = 480;

  /** the list of turtles in the world */
  private List<Turtle> turtleList = new ArrayList<Turtle>();

  /** the JFrame to show this world in */
  private JFrame frame = new JFrame("World");

  /** background picture */
  private Picture picture = null;

  ////////////////// the constructors ///////////////

  /**
   * Constructor that takes no arguments
   */
  public World()
  {
    // set up the world and make it visible
```

```
    initWorld(true);
  }

  /**
   * Constructor that takes a boolean to
   * say if this world should be visible
   * or not
   * @param visibleFlag if true will be visible
   * else if false will not be visible
   */
  public World(boolean visibleFlag)
  {
    initWorld(visibleFlag);
  }

  /**
   * Constructor that takes a width and height for this
   * world
   * @param w the width for the world
   * @param h the height for the world
   */
  public World(int w, int h)
  {
    width = w;
    height = h;

    // set up the world and make it visible
    initWorld(true);
  }

  ////////////////// methods //////////////////////////
  /**
   *Method to return a reference to the JFrame.
   */
  public JFrame getFrame(){
    System.out.println("Dick Baldwin");
    return frame;
  }//end getFrame

  /**
   * Method to initialize the world
   * @param visibleFlag the flag to make the world
   * visible or not
   */
  private void initWorld(boolean visibleFlag)
  {
    // set the preferred size
    this.setPreferredSize(new Dimension(width,height));

    // create the background picture
    picture = new Picture(width,height);
```

```
  // add this panel to the frame
  frame.getContentPane().add(this);

  // pack the frame
  frame.pack();

  // show this world
  frame.setVisible(visibleFlag);
}

/**
 * Method to get the graphics context for drawing on
 * @return the graphics context of the background picture
 */
public Graphics getGraphics() { return picture.getGraphics(); }

/**
 * Method to clear the background picture
 */
public void clearBackground() { picture = new Picture(width,height); }

/**
 * Method to get the background picture
 * @return the background picture
 */
public Picture getPicture() { return picture; }

/**
 * Method to set the background picture
 * @param pict the background picture to use
 */
public void setPicture(Picture pict) { picture = pict; }

/**
 * Method to paint this component
 * @param g the graphics context
 */
public synchronized void paintComponent(Graphics g)
{
  Turtle turtle = null;

  // draw the background image
  g.drawImage(picture.getImage(),0,0,null);

  // loop drawing each turtle on the background image
  Iterator iterator = turtleList.iterator();
  while (iterator.hasNext())
  {
    turtle = (Turtle) iterator.next();
    turtle.paintComponent(g);
```

```
    }
  }

  /**
   * Metod to get the last turtle in this world
   * @return the last turtle added to this world
   */
  public Turtle getLastTurtle()
  {
    return (Turtle) turtleList.get(turtleList.size() - 1);
  }


  /**
   * Method to add a model to this model displayer
   * @param model the model object to add
   */
  public void addModel(Object model)
  {
    turtleList.add((Turtle) model);
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to check if this world contains the passed
   * turtle
   * @return true if there else false
   */
  public boolean containsTurtle(Turtle turtle)
  {
    return (turtleList.contains(turtle));
  }

  /**
   * Method to remove the passed object from the world
   * @param model the model object to remove
   */
  public void remove(Object model)
  {
    turtleList.remove(model);
  }

  /**
   * Method to get the width in pixels
   * @return the width in pixels
   */
  public int getWidth() { return width; }

  /**
   * Method to get the height in pixels
```

```
   * @return the height in pixels
   */
  public int getHeight() { return height; }

  /**
   * Method that allows the model to notify the display
   */
  public void modelChanged()
  {
    if (autoRepaint)
       repaint();
  }

  /**
   * Method to set the automatically repaint flag
   * @param value if true will auto repaint
   */
  public void setAutoRepaint(boolean value) { autoRepaint = value; }

  /**
   * Method to hide the frame
   */
//  public void hide()
//  {
//    frame.setVisible(false);
//  }

  /**
   * Method to show the frame
   */
//  public void show()
//  {
//    frame.setVisible(true);
//  }

  /**
   * Method to set the visibility of the world
   * @param value a boolean value to say if should show or hide
   */
  public void setVisible(boolean value)
  {
    frame.setVisible(value);
  }

  /**
   * Method to get the list of turtles in the world
   * @return a list of turtles in the world
   */
  public List getTurtleList()
  { return turtleList;}
```

```
  /**
   * Method to get an iterator on the list of turtles
   * @return an iterator for the list of turtles
   */
  public Iterator getTurtleIterator()
  { return turtleList.iterator();}

  /**
   * Method that returns information about this world
   * in the form of a string
   * @return a string of information about this world
   */
  public String toString()
  {
    return "A " + getWidth() + " by " + getHeight() +
      " world with " + turtleList.size() + " turtles in it.";
  }

} // end of World class
```

Listing 6.47: Modified SimplePicture class.

```
    import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.swing.ImageIcon;
import java.awt.*;
import java.io.*;
import java.awt.geom.*;

/*
12/23/08 Modified the addMessage method to cause it to
use a color variable to set the color of the message.
Also provided a setMessageColor method to set the color
and a variable named messageColor to contain the color.
/*


/**
 * A class that represents a simple picture.  A simple picture may have
 * an associated file name and a title.  A simple picture has pixels,
 * width, and height.  A simple picture uses a BufferedImage to
 * hold the pixels.  You can show a simple picture in a
 * PictureFrame (a JFrame).
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimplePicture implements DigitalPicture
{

  ////////////////////// Fields ////////////////////////
```

```
 /**
  * the color of the message
  */
  private Color messageColor = Color.WHITE;


 /**
  * the file name associated with the simple picture
  */
 private String fileName;

 /**
  * the title of the simple picture
  */
 private String title;

 /**
  * buffered image to hold pixels for the simple picture
  */
 private BufferedImage bufferedImage;

 /**
  * frame used to display the simple picture
  */
 private PictureFrame pictureFrame;

 /**
  * extension for this file (jpg or bmp)
  */
 private String extension;


////////////////////// Constructors ////////////////////////

/**
 * A Constructor that takes no arguments.  All fields will be null.
 * A no-argument constructor must be given in order for a class to
 * be able to be subclassed.  By default all subclasses will implicitly
 * call this in their parent's no argument constructor unless a
 * different call to super() is explicitly made as the first line
 * of code in a constructor.
 */
public SimplePicture()
{this(200,100);}

/**
 * A Constructor that takes a file name and uses the file to create
 * a picture
 * @param fileName the file name to use in creating the picture
 */
public SimplePicture(String fileName)
```

```
{

  // load the picture into the buffered image
  load(fileName);

}

/**
 * A constructor that takes the width and height desired for a picture and
 * creates a buffered image of that size.  This constructor doesn't
 * show the picture.
 * @param width the desired width
 * @param height the desired height
 */
public  SimplePicture(int width, int height)
{
  bufferedImage = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
  title = "None";
  fileName = "None";
  extension = "jpg";
  setAllPixelsToAColor(Color.white);
}

/**
 * A constructor that takes the width and height desired for a picture and
 * creates a buffered image of that size.  It also takes the
 * color to use for the background of the picture.
 * @param width the desired width
 * @param height the desired height
 * @param theColor the background color for the picture
 */
public  SimplePicture(int width, int height, Color theColor)
{
  this(width,height);
  setAllPixelsToAColor(theColor);
}

/**
 * A Constructor that takes a picture to copy information from
 * @param copyPicture the picture to copy from
 */
public SimplePicture(SimplePicture copyPicture)
{
  if (copyPicture.fileName != null)
  {
     this.fileName = new String(copyPicture.fileName);
     this.extension = copyPicture.extension;
  }
  if (copyPicture.title != null)
     this.title = new String(copyPicture.title);
  if (copyPicture.bufferedImage != null)
```

```
  {
    this.bufferedImage = new BufferedImage(copyPicture.getWidth(),
                                           copyPicture.getHeight(), BufferedImage.TYPE_INT_RGB);
    this.copyPicture(copyPicture);
  }
}

/**
 * A constructor that takes a buffered image
 * @param image the buffered image
 */
public SimplePicture(BufferedImage image)
{
  this.bufferedImage = image;
  title = "None";
  fileName = "None";
  extension = "jpg";
}

////////////////////////// Methods ///////////////////////////////////
/**
 * Method to set the color used for a message.
 */
 public void setMessageColor(Color color){
   System.out.println("Dick Baldwin");
   messageColor = color;
 }//end setMessageColor


/**
 * Method to get the extension for this picture
 * @return the extendsion (jpg or bmp)
 */
public String getExtension() { return extension; }


/**
 * Method that will copy all of the passed source picture into
 * the current picture object
 * @param sourcePicture  the picture object to copy
 */
public void copyPicture(SimplePicture sourcePicture)
{
  Pixel sourcePixel = null;
  Pixel targetPixel = null;

  // loop through the columns
  for (int sourceX = 0, targetX = 0;
       sourceX < sourcePicture.getWidth() &&
       targetX < this.getWidth();
       sourceX++, targetX++)
```

```
  {
    // loop through the rows
    for (int sourceY = 0, targetY = 0;
         sourceY < sourcePicture.getHeight() &&
         targetY < this.getHeight();
         sourceY++, targetY++)
    {
      sourcePixel = sourcePicture.getPixel(sourceX,sourceY);
      targetPixel = this.getPixel(targetX,targetY);
      targetPixel.setColor(sourcePixel.getColor());
    }
  }

}

/**
 * Method to set the color in the picture to the passed color
 * @param color the color to set to
 */
public void setAllPixelsToAColor(Color color)
{
  // loop through all x
  for (int x = 0; x < this.getWidth(); x++)
  {
    // loop through all y
    for (int y = 0; y < this.getHeight(); y++)
    {
      getPixel(x,y).setColor(color);
    }
  }
}

/**
 * Method to get the buffered image
 * @return the buffered image
 */
public BufferedImage getBufferedImage()
{
   return bufferedImage;
}

/**
 * Method to get a graphics object for this picture to use to draw on
 * @return a graphics object to use for drawing
 */
public Graphics getGraphics()
{
  return bufferedImage.getGraphics();
}

/**
```

```
 * Method to get a Graphics2D object for this picture which can
 * be used to do 2D drawing on the picture
 */
public Graphics2D createGraphics()
{
  return bufferedImage.createGraphics();
}

/**
 * Method to get the file name associated with the picture
 * @return  the file name associated with the picture
 */
public String getFileName() { return fileName; }

/**
 * Method to set the file name
 * @param name the full pathname of the file
 */
public void setFileName(String name)
{
  fileName = name;
}

/**
 * Method to get the title of the picture
 * @return the title of the picture
 */
public String getTitle()
{ return title; }

/**
 * Method to set the title for the picture
 * @param title the title to use for the picture
 */
public void setTitle(String title)
{
  this.title = title;
  if (pictureFrame != null)
      pictureFrame.setTitle(title);
}

/**
 * Method to get the width of the picture in pixels
 * @return the width of the picture in pixels
 */
public int getWidth() { return bufferedImage.getWidth(); }

/**
 * Method to get the height of the picture in pixels
 * @return  the height of the picture in pixels
 */
```

```
public int getHeight() { return bufferedImage.getHeight(); }

/**
 * Method to get the picture frame for the picture
 * @return the picture frame associated with this picture
 * (it may be null)
 */
public PictureFrame getPictureFrame() { return pictureFrame; }

/**
 * Method to set the picture frame for this picture
 * @param pictureFrame the picture frame to use
 */
public void setPictureFrame(PictureFrame pictureFrame)
{
  // set this picture objects' picture frame to the passed one
  this.pictureFrame = pictureFrame;
}

/**
 * Method to get an image from the picture
 * @return  the buffered image since it is an image
 */
public Image getImage()
{
  return bufferedImage;
}

/**
 * Method to return the pixel value as an int for the given x and y location
 * @param x the x coordinate of the pixel
 * @param y the y coordinate of the pixel
 * @return the pixel value as an integer (alpha, red, green, blue)
 */
public int getBasicPixel(int x, int y)
{
   return bufferedImage.getRGB(x,y);
}

/**
 * Method to set the value of a pixel in the picture from an int
 * @param x the x coordinate of the pixel
 * @param y the y coordinate of the pixel
 * @param rgb the new rgb value of the pixel (alpha, red, green, blue)
 */
public void setBasicPixel(int x, int y, int rgb)
{
  bufferedImage.setRGB(x,y,rgb);
}

/**
```

```
 * Method to get a pixel object for the given x and y location
 * @param x  the x location of the pixel in the picture
 * @param y  the y location of the pixel in the picture
 * @return a Pixel object for this location
 */
public Pixel getPixel(int x, int y)
{
  // create the pixel object for this picture and the given x and y location
  Pixel pixel = new Pixel(this,x,y);
  return pixel;
}

/**
 * Method to get a one-dimensional array of Pixels for this simple picture
 * @return a one-dimensional array of Pixel objects starting with y=0
 * to y=height-1 and x=0 to x=width-1.
 */
public Pixel[] getPixels()
{
  int width = getWidth();
  int height = getHeight();
  Pixel[] pixelArray = new Pixel[width * height];

  // loop through height rows from top to bottom
  for (int row = 0; row < height; row++)
    for (int col = 0; col < width; col++)
      pixelArray[row * width + col] = new Pixel(this,col,row);

  return pixelArray;
}



/**
 * Method to load the buffered image with the passed image
 * @param image  the image to use
 */
public void load(Image image)
{
  // get a graphics context to use to draw on the buffered image
  Graphics2D graphics2d = bufferedImage.createGraphics();

  // draw the image on the buffered image starting at 0,0
  graphics2d.drawImage(image,0,0,null);

  // show the new image
  show();
}

/**
 * Method to show the picture in a picture frame
```

```
 */
public void show()
{
   // if there is a current picture frame then use it
  if (pictureFrame != null)
    pictureFrame.updateImageAndShowIt();

  // else create a new picture frame with this picture
  else
    pictureFrame = new PictureFrame(this);
}

/**
 * Method to hide the picture
 */
public void hide()
{
  if (pictureFrame != null)
    pictureFrame.setVisible(false);
}

/**
 * Method to make this picture visible or not
 * @param flag true if you want it visible else false
 */
public void setVisible(boolean flag)
{
  if (flag)
    this.show();
  else
    this.hide();
}

/**
 * Method to open a picture explorer on a copy of this simple picture
 */
public void explore()
{
  // create a copy of the current picture and explore it
  new PictureExplorer(new SimplePicture(this));
}

/**
 * Method to force the picture to redraw itself.  This is very
 * useful after you have changed the pixels in a picture.
 */
public void repaint()
{
  // if there is a picture frame tell it to repaint
  if (pictureFrame != null)
    pictureFrame.repaint();
```

```
  // else create a new picture frame
  else
    pictureFrame = new PictureFrame(this);
}

/**
 * Method to load the picture from the passed file name
 * @param fileName the file name to use to load the picture from
 */
public void loadOrFail(String fileName) throws IOException
{
   // set the current picture's file name
  this.fileName = fileName;

  // set the extension
  int posDot = fileName.indexOf('.');
  if (posDot >= 0)
    this.extension = fileName.substring(posDot + 1);

  // if the current title is null use the file name
  if (title == null)
    title = fileName;

  File file = new File(this.fileName);

  if (!file.canRead())
  {
    // try adding the media path
    file = new File(FileChooser.getMediaPath(this.fileName));
    if (!file.canRead())
    {
      throw new IOException(this.fileName +
                            " could not be opened. Check that you specified the path");
    }
  }

  bufferedImage = ImageIO.read(file);
}


/**
 * Method to write the contents of the picture to a file with
 * the passed name without throwing errors
 * @param fileName the name of the file to write the picture to
 * @return true if success else false
 */
public boolean load(String fileName)
{
    try {
        this.loadOrFail(fileName);
```

```
        return true;

    } catch (Exception ex) {
        System.out.println("There was an error trying to open " + fileName);
        bufferedImage = new BufferedImage(600,200,
                                          BufferedImage.TYPE_INT_RGB);
        addMessage("Couldn't load " + fileName,5,100);
        return false;
    }

}


/**
 * Method to load the picture from the passed file name
 * this just calls load(fileName) and is for name compatibility
 * @param fileName the file name to use to load the picture from
 * @return true if success else false
 */
public boolean loadImage(String fileName)
{
    return load(fileName);
}

/**
 * Method to draw a message as a string on the buffered image
 * @param message the message to draw on the buffered image
 * @param xPos  the leftmost point of the string in x
 * @param yPos  the bottom of the string in y
 */
public void addMessage(String message, int xPos, int yPos)
{
  // get a graphics context to use to draw on the buffered image
  Graphics2D graphics2d = bufferedImage.createGraphics();

  // set the color to white
//   graphics2d.setPaint(Color.white);
  //modified by baldwin on 12/23/08
  graphics2d.setPaint(messageColor);

  // set the font to Helvetica bold style and size 16
  graphics2d.setFont(new Font("Helvetica",Font.BOLD,16));

  // draw the message
  graphics2d.drawString(message,xPos,yPos);

}

/**
 * Method to draw a string at the given location on the picture
 * @param text the text to draw
```

```
 * @param xPos the left x for the text
 * @param yPos the top y for the text
 */
public void drawString(String text, int xPos, int yPos)
{
  addMessage(text,xPos,yPos);
}

/**
  * Method to create a new picture by scaling the current
  * picture by the given x and y factors
  * @param xFactor the amount to scale in x
  * @param yFactor the amount to scale in y
  * @return the resulting picture
  */
 public Picture scale(double xFactor, double yFactor)
 {
   // set up the scale tranform
   AffineTransform scaleTransform = new AffineTransform();
   scaleTransform.scale(xFactor,yFactor);

   // create a new picture object that is the right size
   Picture result = new Picture((int) (getWidth() * xFactor),
                                (int) (getHeight() * yFactor));

   // get the graphics 2d object to draw on the result
   Graphics graphics = result.getGraphics();
   Graphics2D g2 = (Graphics2D) graphics;

   // draw the current image onto the result image scaled
   g2.drawImage(this.getImage(),scaleTransform,null);

   return result;
 }

 /**
  * Method to create a new picture of the passed width.
  * The aspect ratio of the width and height will stay
  * the same.
  * @param width the desired width
  * @return the resulting picture
  */
 public Picture getPictureWithWidth(int width)
 {
   // set up the scale tranform
   double xFactor = (double) width / this.getWidth();
   Picture result = scale(xFactor,xFactor);
   return result;
 }

 /**
```

```
   * Method to create a new picture of the passed height.
   * The aspect ratio of the width and height will stay
   * the same.
   * @param height the desired height
   * @return the resulting picture
   */
  public Picture getPictureWithHeight(int height)
  {
    // set up the scale tranform
    double yFactor = (double) height / this.getHeight();
    Picture result = scale(yFactor,yFactor);
    return result;
  }

 /**
  * Method to load a picture from a file name and show it in a picture frame
  * @param fileName the file name to load the picture from
  * @return true if success else false
  */
 public boolean loadPictureAndShowIt(String fileName)
 {
   boolean result = true;  // the default is that it worked

   // try to load the picture into the buffered image from the file name
   result = load(fileName);

   // show the picture in a picture frame
   show();

   return result;
 }

 /**
  * Method to write the contents of the picture to a file with
  * the passed name
  * @param fileName the name of the file to write the picture to
  */
 public void writeOrFail(String fileName) throws IOException
 {
   String extension = this.extension; // the default is current

   // create the file object
   File file = new File(fileName);
   File fileLoc = file.getParentFile();

   // canWrite is true only when the file exists already! (alexr)
   if (!fileLoc.canWrite()) {
       // System.err.println("can't write the file but trying anyway? ...");
        throw new IOException(fileName +
        " could not be opened. Check to see if you can write to the directory.");
   }
```

```
  // get the extension
  int posDot = fileName.indexOf('.');
  if (posDot >= 0)
      extension = fileName.substring(posDot + 1);

  // write the contents of the buffered image to the file as jpeg
  ImageIO.write(bufferedImage, extension, file);

}

/**
 * Method to write the contents of the picture to a file with
 * the passed name without throwing errors
 * @param fileName the name of the file to write the picture to
 * @return true if success else false
 */
public boolean write(String fileName)
{
    try {
        this.writeOrFail(fileName);
        return true;
    } catch (Exception ex) {
        System.out.println("There was an error trying to write " + fileName);
        return false;
    }

}

/**
 * Method to set the media path by setting the directory to use
 * @param directory the directory to use for the media path
 */
public static void setMediaPath(String directory) {
  FileChooser.setMediaPath(directory);
}

/**
 * Method to get the directory for the media
 * @param fileName the base file name to use
 * @return the full path name by appending
 * the file name to the media directory
 */
public static String getMediaPath(String fileName) {
  return FileChooser.getMediaPath(fileName);
}

 /**
  * Method to get the coordinates of the enclosing rectangle after this
  * transformation is applied to the current picture
  * @return the enclosing rectangle
```

```
   */
  public Rectangle2D getTransformEnclosingRect(AffineTransform trans)
  {
    int width = getWidth();
    int height = getHeight();
    double maxX = width - 1;
    double maxY = height - 1;
    double minX, minY;
    Point2D.Double p1 = new Point2D.Double(0,0);
    Point2D.Double p2 = new Point2D.Double(maxX,0);
    Point2D.Double p3 = new Point2D.Double(maxX,maxY);
    Point2D.Double p4 = new Point2D.Double(0,maxY);
    Point2D.Double result = new Point2D.Double(0,0);
    Rectangle2D.Double rect = null;

    // get the new points and min x and y and max x and y
    trans.deltaTransform(p1,result);
    minX = result.getX();
    maxX = result.getX();
    minY = result.getY();
    maxY = result.getY();
    trans.deltaTransform(p2,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p3,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p4,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());

    // create the bounding rectangle to return
    rect = new Rectangle2D.Double(minX,minY,maxX - minX + 1, maxY - minY + 1);
    return rect;
  }

 /**
  * Method to return a string with information about this picture
  * @return a string with information about the picture
  */
 public String toString()
 {
   String output = "Simple Picture, filename " + fileName +
     " height " + getHeight() + " width " + getWidth();
   return output;
```

```
 }

} // end of SimplePicture class
```

-end-

6.3.1.5 Java OOP: Panels, Labels, Text Fields, and Buttons[35]

6.3.1.5.1 Table of Contents

6.3.1.5.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [36] .

6.3.1.5.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.1.5.2.1.1 Figures

6.3.1.5.2.1.2 Listings

[35] This content is available online at <http://cnx.org/content/m44352/1.2/>.
[36] http://cnx.org/content/m44148/latest/

6.3.1.5.3 Preview

In this module, you will learn how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

Just as you did in earlier modules, you will modify the **World** class to make it possible to get access to the **JFrame** object that is encapsulated in a **World** object. However, because I explained those modifications to the **World** class in earlier modules, I won't repeat the explanation here. You can find a modified version of the World class in Java OOP: Background Color, Text Color, Mouse Clicks, etc. [37]

Program specifications

Write a program named **Prob05** that uses the class definition named **Prob05** shown in Listing 9 (p. 1508) along with Ericson's media library to produce the graphic output images shown in Figure 1 (p. 1500) , Figure 2 (p. 1501) , and Figure 3 (p. 1502) .

Program output at startup

Figure 1 (p. 1500) shows the image that appears on the screen when the program first starts running.

[37] http://cnx.org/content/m44351/latest/#Listing_10

Program output at startup.

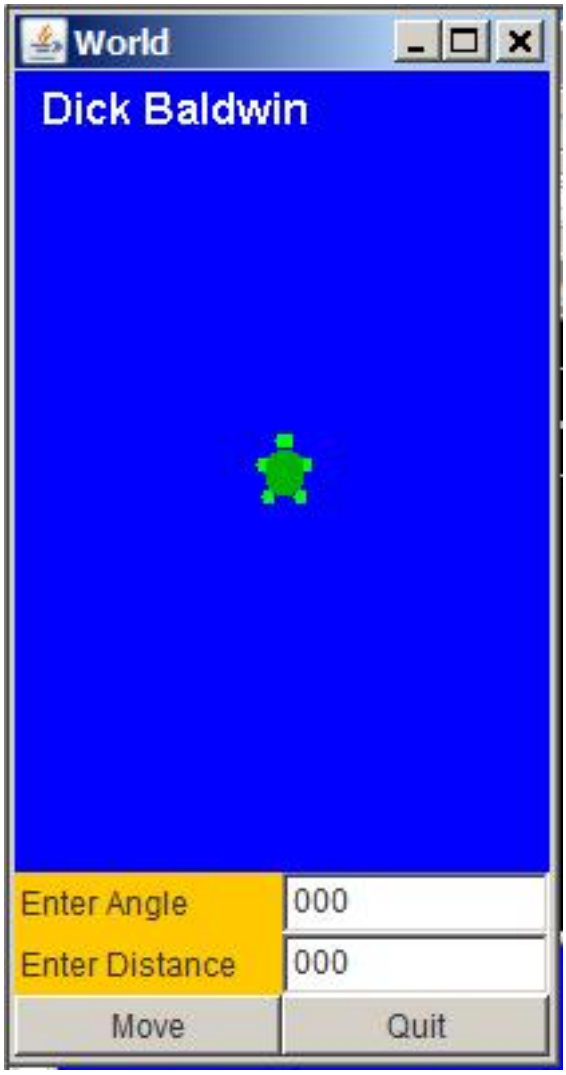


Figure 6.13: Program output at startup.

Program output after one click

Figure 2 (p. 1501) shows the output image after the user enters the values shown into the two text fields and clicks the **Move** button once.

Program output after one click.

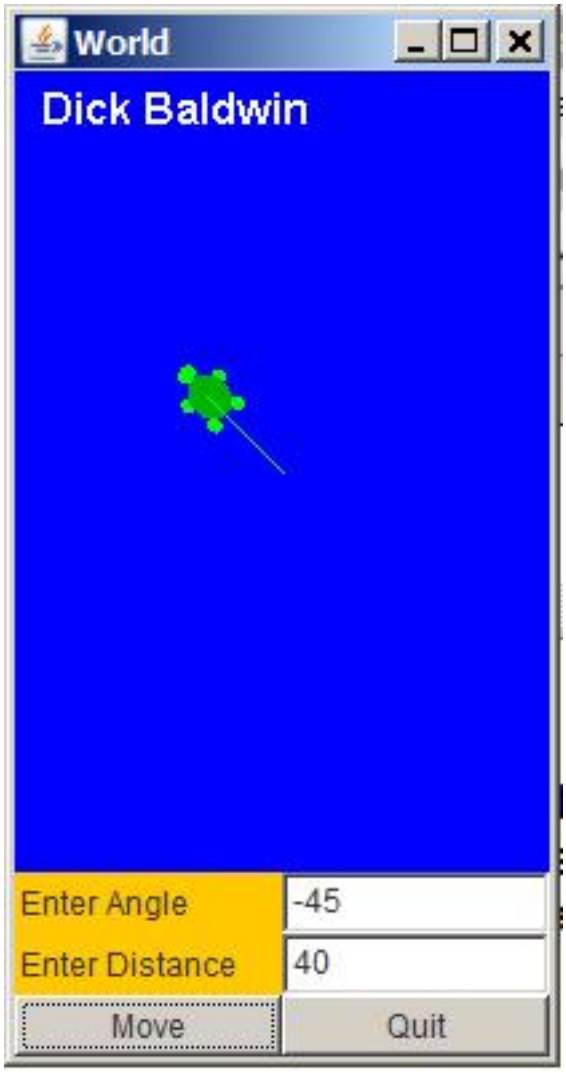


Figure 6.14: Program output after one click.

Program output after several clicks

Figure 3 (p. 1502) shows the output image after the user enters several different sets of values into the text fields and clicks the **Move** button several times.

Program output after several clicks.

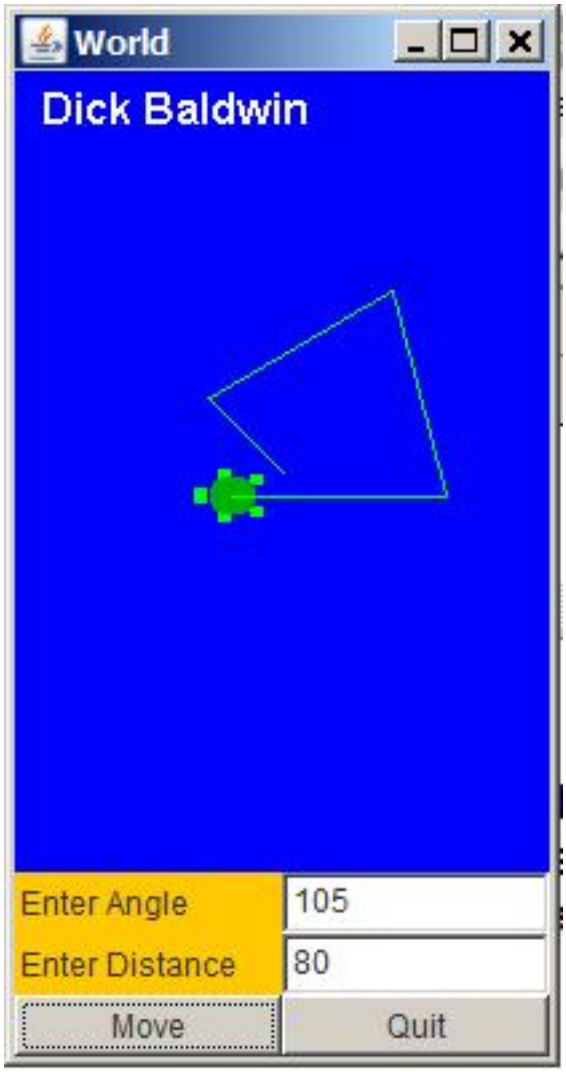


Figure 6.15: Program output after several clicks.

Labels, text fields, and buttons

As shown in Figure 1 (p. 1500) , this program adds two buttons, two labels, and two text fields to form a GUI at the bottom of the **World** object. Although it isn't obvious, those six components are contained in a **Panel** object.

Each time you enter numeric values into the angle and distance fields and then click the **Move** button, the turtle will turn by that angle in degrees and move by that distance in pixels.

Program termination

The program terminates and returns control to the operating system when the user clicks the **Quit** button.

Use AWT components instead of Swing components

The GUI at the bottom of the **World** object is comprised of ***AWT*** components instead of ***Swing*** components.

Required text output

In addition to the output image described above, your program must produce some rather inconsequential text output on the command- line screen.

6.3.1.5.4 Discussion and sample code

The class named Prob05

You can view the driver class named **Prob05** at the beginning of the source code in Listing 9 (p. 1508) . You are already familiar with the code in the **main** method of that class from earlier modules so I won't waste time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob05Runner** . Once that object is instantiated, the program goes idle waiting for an event to happen. Events happen when the user enters text into the text fields or presses one of the buttons shown in Figure 1 (p. 1500) .

The class named Prob05Runner

I will explain this program in fragments. A complete listing of the program is provided in Listing 9 (p. 1508) near the end of the module

The class named **Prob05Runner** begins in Listing 1 (p. 1503) .

Listing 6.48: Beginning of the class named Prob05Runner.

```
 class Prob05Runner{
Turtle turtle = null;
Picture picture = null;
World world = new World(200,300);
```

There's nothing new in Listing 1 (p. 1503) . You have seen code like this in several earlier modules.

Instantiate GUI components

Listing 2 (p. 1503) instantiates the GUI components that are used to construct the GUI at the bottom of Figure 1 (p. 1500) .

Listing 6.49: Instantiate GUI components.

```
   Panel mainPanel = new Panel();
Label angleLabel = new Label("Enter Angle");
TextField angleField = new TextField("000");
Label distanceLabel = new Label("Enter Distance");
TextField distanceField = new TextField("000");
Button moveButton = new Button("Move");
Button quitButton = new Button("Quit");

int angle = 0;
int distance = 0;
```

References to the GUI components are saved in instance variables with descriptive names.

GUI component initialization

Except for the **Panel** object, each GUI component is initialized with information that is appropriate to the type of component. Compare the initialization values in Listing 2 (p. 1503) with the image in Figure 1 (p. 1500) for a better understanding of what I mean by this.

Brief description of the GUI components

You are encouraged to visit the Sun Java documentation and read about the following AWT components:

Panel : A panel provides space in which an application can attach any other components, including other panels.

- **Label** : A Label object is a component for placing text in a container. A label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly.
- **TextField** : A text component that allows for the editing of a single line of text.
- **Button** : This class creates a labeled button. The application can cause some action to happen when the button is pushed.

Figure 1 (p. 1500) contains a **Panel** object at the bottom. The panel contains two **Label** objects *(shown as orange)* , two **TextField** objects *(white)* , and two **Button** objects *(gray)* . Those seven objects are instantiated in Listing 2 (p. 1503) .

The angle and distance variables

Listing 2 (p. 1503) also declares and initializes two instance variables named **angle** and **distance** . *(Note that these two variables would be automatically initialized to zero if I didn't initialize them, but I prefer to initialize them explicitly in order to make the code more self-documenting.)*

Beginning of the constructor

The constructor for the class named **Prob05Runner** begins in Listing 3 (p. 1504) .

Listing 6.50: Beginning of the constructor.

```
  public Prob05Runner(){
System.out.println("Dick Baldwin");

mainPanel.setBackground(Color.ORANGE);

mainPanel.setLayout(new GridLayout(0,2));
```

The constructor begins by displaying my name on the command line screen. This is inconsequential insofar as the overall operation of the program is concerned.

Set the panel background color to orange

Then Listing 3 (p. 1504) sets the background color of the panel to orange. This is what causes the two labels in Figure 1 (p. 1500) to appear to be orange. They are actually transparent except for the text. The orange color shows through causing the labels to appear to be orange.

Set the layout to GridLayout

Setting the layout manager controls how the components that are placed in the panel will be arranged. A **GridLayout** causes all components to be the same size arranged in rows and columns.

Overloaded constructors

There are several overloaded versions of the **GridLayout** constructor. For the constructor used in Listing 3 (p. 1504) , the parameters specify the number of rows and the number of columns in that order. Specifying the number of rows as 0 and the number of columns as 2 means that the layout manager will accept any number of rows but only two columns.

The order of component placement

The intersections of the rows and columns create ***cells*** . Components are placed in the cells in left to right, top to bottom order as they are added to the panel.

Add the six GUI components to the panel

Listing 4 (p. 1504) adds the six GUI components to the panel in the left to right, top to bottom order described above.

Listing 6.51: Add the six GUI components to the panel.

```
    mainPanel.add(angleLabel);
mainPanel.add(angleField);
```

```
mainPanel.add(distanceLabel);
mainPanel.add(distanceField);
mainPanel.add(moveButton);
mainPanel.add(quitButton);
```

At this point, the panel has been populated with GUI components, but the panel itself has not been added to the **JFrame** object that forms the **World** object. That is accomplished in Listing 5 (p. 1505) .

Get the frame and add the panel to the frame

Listing 5 (p. 1505) gets a reference to the **World** object's frame and adds the panel to the SOUTH location in that frame.

Listing 6.52: Get the frame and add the panel to the frame.

```
    JFrame frame = world.getFrame();
frame.getContentPane().add(
                        mainPanel,BorderLayout.SOUTH);

frame.pack();
```

You can surmise from the word SOUTH that the panel is added to the bottom of the frame. To learn more about this, visit the **BorderLayout** class in Sun's Java documentation.

Pack the frame

As you learned in an earlier module, it is very important that you pack the frame at this point. Packing the frame causes the frame to adjust its dimensions in order to accommodate all of the components that have been added to it. In this case, the frame contains a **Picture** object *(placed there when the **World** was constructed)* and a **Panel** object placed there in Listing 5 (p. 1505) .

What about the size of the panel?

Exactly how the panel decides what size it needs to be to accommodate the six GUI components is a fairly complicated topic, so I won't go into it here. However, if you do much work developing GUIs, you definitely need to understand the process. I have explained the automatic sizing process in several tutorials on my website.

Set the background to blue and add a turtle

Listing 6 (p. 1505) sets the background color of the world to blue and adds a turtle to the world.

Listing 6.53: Set the background to blue and add a turtle.

```
    //Initialize the picture.
picture = world.getPicture();
//Set picture background to BLUE
picture.setAllPixelsToAColor(Color.BLUE);
//Display the student's name on the picture.
picture.addMessage("Dick Baldwin",10,20);
//Add a turtle to the world. This causes the
// world to be repainted.
turtle = new Turtle(world);
```

There is nothing new in Listing 6 (p. 1505) . I have explained code very similar to this code in earlier modules.

Could stop at this point

If we were to stop programming at this point, the program would be executable, and would produce the output shown in Figure 1 (p. 1500) when it is run. However it would be completely passive. By that, I mean that entering values into the text fields and clicking the buttons at the bottom of Figure 1 (p. 1500)

would have no effect. However, the buttons would appear to be active from a visual viewpoint because the animation behavior is built into objects of the **Button** class.

Register listener objects

In order to cause the buttons to impact the behavior of the program, we must instantiate and register listener object on the buttons. I will do that using anonymous classes.

Define, instantiate, and register a listener on the Move button

The code in Listing 7 (p. 1506) :

- Defines an anonymous **ActionListener** class.
- Instantiates an anonymous object of that class.
- Registers that object on the **Move** button

Listing 6.54: Define, instantiate, and register a listener on the Move button.

```
    moveButton.addActionListener(
  new ActionListener(){

    public void actionPerformed(ActionEvent e){
      angle = Integer.parseInt(angleField.getText());
      distance = Integer.parseInt(
                           distanceField.getText());
      turtle.turn(angle);
      turtle.forward(distance);
    }//end action performed

  }//end newActionListener
 );//end addActionListener
```

The actionPerformed method

As you learned in an earlier module, the code in the **actionPerformed** method is executed each time the user presses the **Move** button.

The **actionPerformed** method in Listing 7 (p. 1506) begins by getting the text from each of the text fields, converting the text to type **int** , and saving the **int** values in the variables named **angle** and **distance** .

Then Listing 7 (p. 1506) calls the **turn** and **forward** methods of the **Turtle** class to cause the turtle to turn by the specified angle and then move forward by the specified distance.

Action listener to terminate the program

Listing 8 (p. 1506) registers an action listener on the **Quit** button, which will cause the program to terminate when the user clicks the button.

Listing 6.55: Action listener to terminate the program.

```
    quitButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){

      System.exit(0);

    }//end action performed
  }//end newActionListener
 );//end addActionListener
```

```
  }//end constructor
  //-----------------------------------------------------//

}//end class Prob05Runner
```

As you can see in Listing 8 (p. 1506) , this code causes the **exit** method of the **System** class to be called when the user clicks the **Quit** button. According to the Sun documentation, a call to the **exit** method *"Terminates the currently running Java Virtual Machine."*

The end of the constructor and the end of the class

Listing 8 (p. 1506) also shows the end of the constructor for the **Prob05Runner** class and the end of the class.

6.3.1.5.5 Run the program

I encourage you to copy the code from Listing 9 (p. 1508) . Compile the code and execute it. Experiment with the code, making changes and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.1.5.6 Summary

In this module, you learned how to create and service a graphical user interface containing panels, labels, text fields, and buttons.

6.3.1.5.7 What's next?

In the next module, you will learn about:

- Alpha transparency
- A buffered image of type TYPE_INT_ARGB
- The ability to use the **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators,
- The use of the drawImage method of the **Graphics** class.

6.3.1.5.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Panels, Labels, Text Fields, and Buttons
- File: Java3110.htm
- Revised: 08/20/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.1.5.9 Complete program listing

A complete listing of the program discussed in this lesson is shown in Listing 9 (p. 1508) below.

Listing 6.56: Source code for Prob05.

```
    /*File Prob05 Copyright 2008 R.G.Baldwin
 *Revised 12/31/08
**********************************************************/
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.TextField;
import java.awt.Label;
import java.awt.Button;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

import javax.swing.JFrame;

public class Prob05{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob05Runner();
  }//end main method
}//end class Prob05
//End program specifications.
/*-------------------------------------------------------//

**********************************************************/

class Prob05Runner{
  Turtle turtle = null;
  Picture picture = null;
  World world = new World(200,300);
  Panel mainPanel = new Panel();
  Label angleLabel = new Label("Enter Angle");
  TextField angleField = new TextField("000");
  Label distanceLabel = new Label("Enter Distance");
  TextField distanceField = new TextField("000");
  Button moveButton = new Button("Move");
  Button quitButton = new Button("Quit");
  int angle = 0;
```

```
int distance = 0;

public Prob05Runner(){
  System.out.println("Dick Baldwin");

  //Construct the GUI.
  mainPanel.setBackground(Color.ORANGE);
  mainPanel.setLayout(new GridLayout(0,2));
  mainPanel.add(angleLabel);
  mainPanel.add(angleField);
  mainPanel.add(distanceLabel);
  mainPanel.add(distanceField);
  mainPanel.add(moveButton);
  mainPanel.add(quitButton);

  //Get a reference to the world frame and add the GUI
  // to the frame.
  JFrame frame = world.getFrame();
  frame.getContentPane().add(
                         mainPanel,BorderLayout.SOUTH);


  frame.pack();

  //Initialize the picture.
  picture = world.getPicture();
  //Set picture background to BLUE
  picture.setAllPixelsToAColor(Color.BLUE);
  //Display the student's name on the picture.
  picture.addMessage("Dick Baldwin",10,20);
  //Add a turtle to the world. This causes the
  // world to be repainted.
  turtle = new Turtle(world);

  //----------------------------------------------------//
  //Register anonymous listeners on the two buttons.
  moveButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){
        angle = Integer.parseInt(angleField.getText());
        distance = Integer.parseInt(
                              distanceField.getText());
        turtle.turn(angle);
        turtle.forward(distance);
      }//end action performed
    }//end newActionListener
  );//end addActionListener

  quitButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){
```

```
          System.exit(0);
        }//end action performed
      }//end newActionListener
    );//end addActionListener

  }//end constructor
  //-------------------------------------------------------//

}//end class Prob05Runner
```

-end-

6.3.2 Part 2

6.3.2.1 Java OOP: Using Alpha Transparency with Ericson's Media Library[38]

6.3.2.1.1 Table of Contents

6.3.2.1.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [39] .

6.3.2.1.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.2.1.2.1.1 Figures

[38] This content is available online at <http://cnx.org/content/m44911/1.1/>.
[39] http://cnx.org/content/m44148/latest/

6.3.2.1.2.1.2 Listings

6.3.2.1.3 Preview

The primary objective of this module is to incorporate alpha transparency into the use of Ericson's media library.

Two approaches

There are at least two ways to incorporate alpha transparency into Ericson's media library, The easiest way, which is not necessarily the best way, is to make a relatively simple modification to a constructor in Ericson's **SimplePicture** class. That is the approach used in this module.

The second approach

The second approach is more complicated, but does not require the modification of the classes in Ericson's library. That is probably a better approach due simply to the fact that modifications to Ericson's library are not required. However, that approach is not shown in this module.

Outside research

This program may require a significant amount of outside research on the part of the student in order to learn about:

- Alpha transparency
- A buffered image of type **TYPE_INT_ARGB**
- The ability to use Ericson's **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators, and
- The use of the **drawImage** method of the **Graphics** class.

The getBasicPixel and setBasicPixel methods

The program uses the **getBasicPixel** and **setBasicPixel** methods from Ericson's library along with bitwise operations to set the alpha value for all the pixels in a cropped and flipped image of a butterfly to a hexadecimal value of 5F.

Modification to the SimplePicture class

The student must modify the **SimplePicture** class to cause the buffered image used to store the image to be **TYPE_INT_ARGB** instead of **TYPE_INT_RGB** , which is its normal type.

Crop, flip, and set alpha values

Then the student must write a method that will crop and flip an image of a butterfly and set the value of every alpha byte to a hexadecimal value of 5F.

Draw a partially transparent image of a butterfly

Finally, the student must use the standard **drawImage** method of the **Graphics** class to draw the image of the butterfly onto an image of a beach with transparency.

Brief program specifications

Write a program named **Prob06** that uses the class definition for the class named **Prob06** in Listing 7 (p. 1521) along with Ericson's media library and the image files named **Prob06a.jpg** [40] and **Prob06b.jpg** [41] to produce the three graphic output images shown in Figure 1 (p. 1512) , Figure 2 (p. 1513) , and Figure 3 (p. 1514) .

[40] http://cnx.org/content/m44911/latest/Prob06a.jpg
[41] http://cnx.org/content/m44911/latest/Prob06b.jpg

Image from file named Prob06a.

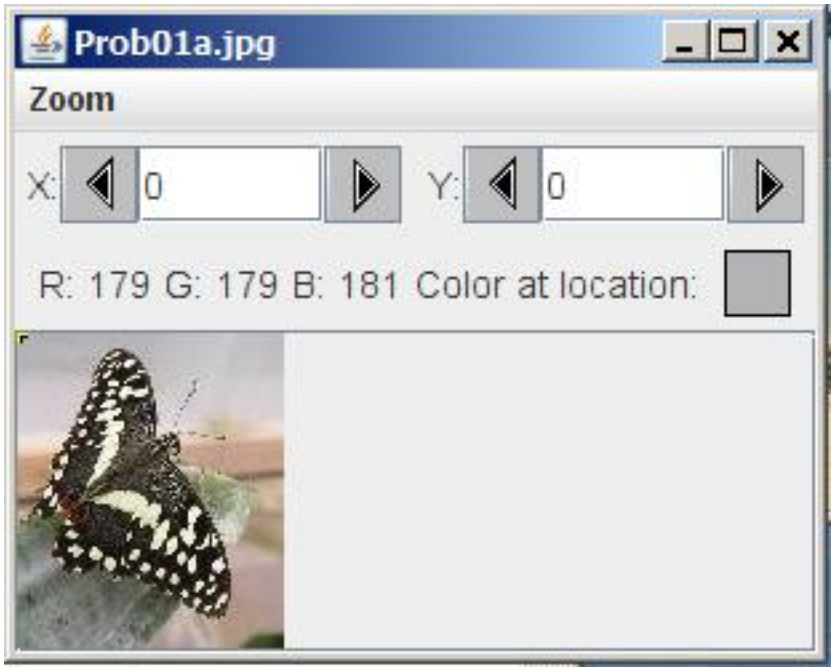


Figure 6.16: Image from file named Prob06a.

Image from file named Prob06b.

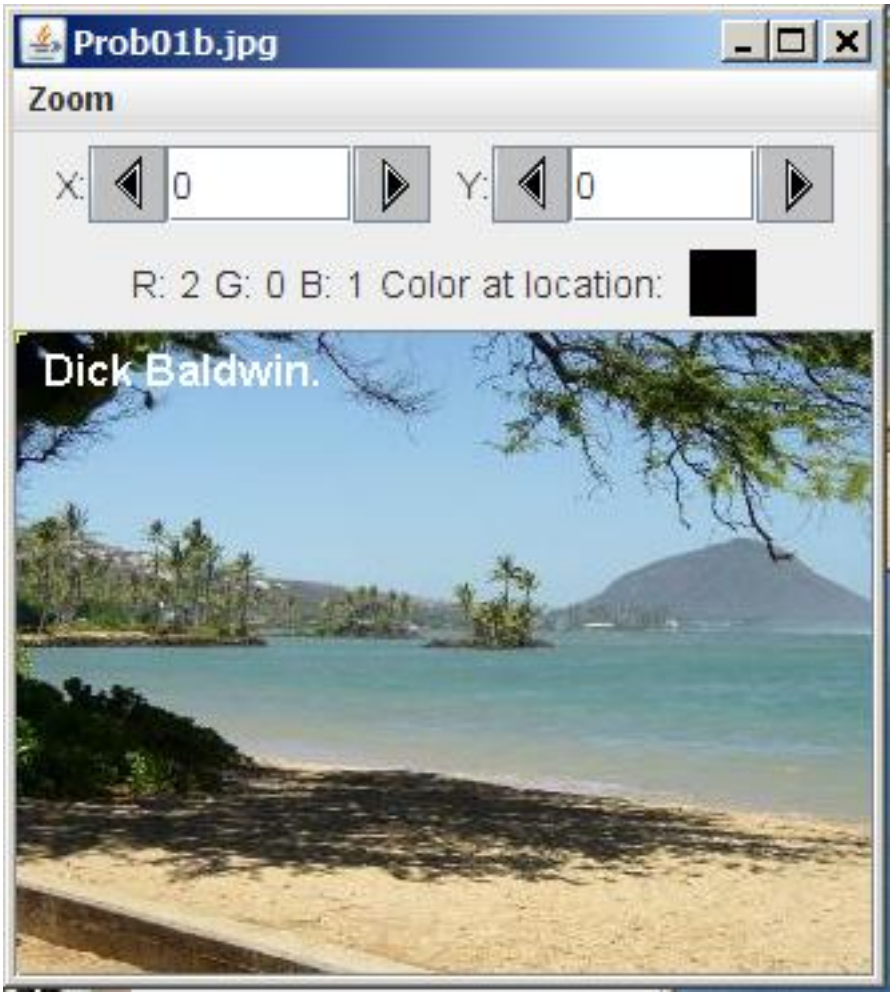


Figure 6.17: Image from file named Prob06b.

Processed output image.

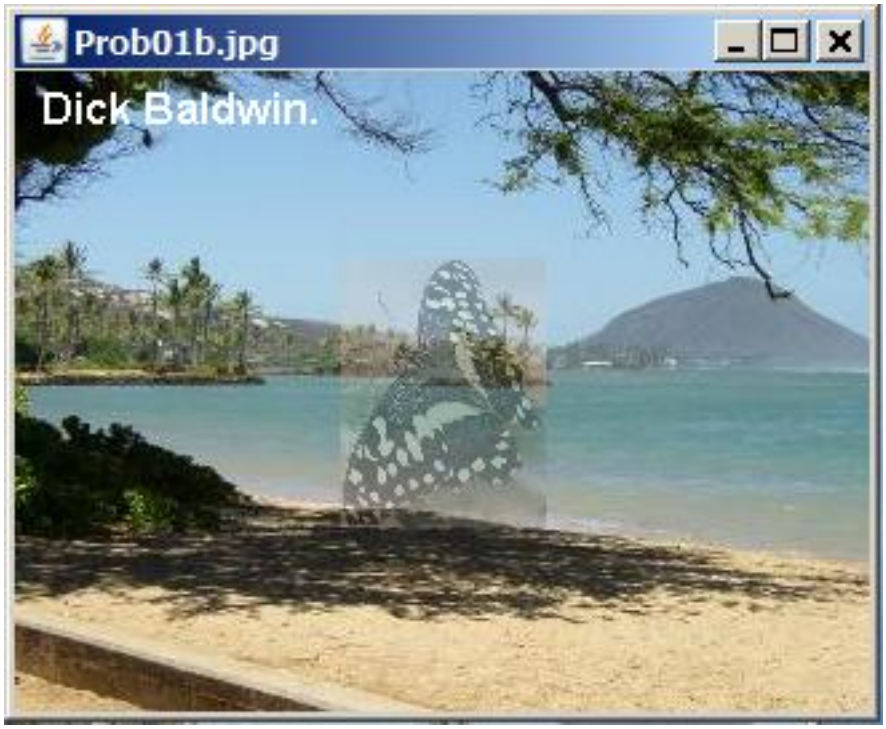


Figure 6.18: Processed output image.

Define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob06** given in Listing 7 (p. 1521) .

A partially transparent image of a butterfly

Just in case you haven't noticed it, the final image of the beach contains a partially transparent image of a butterfly superimposed and centered on the beach image.

Modification to the SimplePicture class

In order to write this program, you will need to modify the class from Ericson's media library named **SimplePicture** .

Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through.

Transparency

The degree of transparency can range from being completely transparent at one extreme to being totally opaque at the other extreme. In this case, the butterfly image shown in Figure 3 (p. 1514) is about 37-percent opaque *(or 63-percent transparent)* .

Outside research

You will probably need to do some outside research in order to write this program. For example, you will need to learn about the following topics and probably some other topics as well:

- Alpha transparency

- BufferedImage objects of TYPE_INT_ARGB
- The representation of a pixel as type int
- Bit manipulation of pixels
- The drawImage method of the Graphics class

Required text output

In addition to the output images described above, your program must produce the text output shown in Figure 4 (p. 1515) on the command- line screen.

Required text output.

```
    Dick Baldwin.
Dick Baldwin
Picture, filename Prob06a.jpg height 118 width 100
Picture, filename Prob06b.jpg height 240 width 320
Picture, filename None height 101 width 77
```

Figure 6.19: Required text output.

You must substitute your name for my name wherever my name appears both in the images and on the command-line screen.

6.3.2.1.4 General background information

The image in a **SimplePicture** object is stored in an object of the **BufferedImage** class, which is a class in the standard Sun Java library.

Image data formats

An examination of the documentation for the **BufferedImage** class shows that the red, green, blue, and alpha values for each pixel can be formatted in about fourteen different ways in an object of the **BufferedImage** class.

No alpha data

Some of those formats, including the way that information is stored in a **SimplePicture** object, don't include an alpha value.

Modification of the SimplePicture class

One way to modify the **SimplePicture** class to force it to accommodate alpha transparency data is to modify one of the constructors for the **SimplePicture** class as shown in Listing 1 (p. 1515) . Note that **BufferedImage.TYPE_INT_RGB** was replaced by **BufferedImage.TYPE_INT_ARGB** in Listing 1. *(There are probably other ways that you can modify the class to achieve the same result as well.)*

Listing 6.57: Modification of the SimplePicture class.

```
/**
```

```
  * A constructor that takes the width and height desired
  * for a picture and creates a buffered image of that
  * size.  This constructor doesn't show the picture.
  */
 public  SimplePicture(int width, int height){
//Disable the following statement
//   bufferedImage = new BufferedImage(
//            width, height, BufferedImage.TYPE_INT_RGB);

   //Modify constructor to support alpha transparency.
   System.out.println("Dick Baldwin");
   bufferedImage = new BufferedImage(
             width, height, BufferedImage.TYPE_INT_ARGB);

   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }//end constructor
```

Future Picture objects will accommodate alpha transparency

Having made this modification, future objects instantiated from the **SimplePicture** class using this constructor will accommodate alpha transparency. *(The **SimplePicture** class is the superclass of the **Picture** class.)*

Display the student's name

Note that the constructor in Listing 1 (p. 1515) is also modified to cause it to display the student's name, which is a requirement of the program.

No complete listing of SimplePicture provided

Because of the simplicity of this modification, a complete listing of the modified **SimplePicture** class will not be provided in this module.

6.3.2.1.5 Discussion and sample code

6.3.2.1.5.1 The class named Prob06

You can view the driver class named **Prob06** at the beginning of the source code in Listing 7 (p. 1521) . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob06Runner** and calls the **run** method on that object. When the **run** method returns, the code in the **main** method displays some information about the three images and terminates.

(Because there are images on the screen, the program does not actually terminate until the user forces it to terminate.)

6.3.2.1.5.2 The class named Prob06Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 7 (p. 1521) near the end of the module

The class named **Prob06Runner** begins in Listing 2 (p. 1516) , which shows the constructor for the class.

Listing 6.58: Beginning of the class named Prob06Runner.

```
  class Prob06Runner{

public Prob06Runner(){//constructor
  System.out.println("Dick Baldwin.");
}//end constructor
```

The constructor simply displays the student's name to satisfy one of the requirements of the program.

The run method

The run method, which is called from the **main** method in Listing 7 (p. 1521) , is shown in its entirety in Listing 3 (p. 1517) .

Listing 6.59: The run method.

```
  public Picture[] run(){
//Insert executable code here
Picture picA = new Picture("Prob06a.jpg");
picA.explore();
Picture picB = new Picture("Prob06b.jpg");
picB.addMessage("Dick Baldwin.",10,20);
picB.explore();

Picture picC = cropAndFlip(picA,4,5,80,105);

copyPictureWithCrop(picC,picB,122,70);

picB.show();

Picture[] output = {picA,picB,picC};
return output;
}//end run
```

New material

The only thing in Listing 3 (p. 1517) that is new to this module is the pair of calls to the following methods. I will explain these methods in the paragraphs that follow:

- **cropAndFlip**
- **copyPictureWithCrop**

Beginning of the cropAndFlip method

The **cropAndFlip** method begins in Listing 4 (p. 1517) . This method receives an incoming reference to a **Picture** object. It crops the picture to a set of specified coordinate values and flips it around a vertical line at its center.

Listing 6.60: Beginning of the cropAndFlip method.

```
  private Picture cropAndFlip(
            Picture pic,int x1,int y1,int x2,int y2){
Picture output = new Picture(x2-x1+1,y2-y1+1);

int width = output.getWidth();
Pixel pixel = null;
Color color = null;
```

```
    for(int col = x1;col < (x2+1);col++){
      for(int row = y1;row < (y2+1);row++){
        color = pic.getPixel(col,row).getColor();
        pixel = output.getPixel(width-col+x1-1,row-y1);
        pixel.setColor(color);
      }//end inner loop
    }//end outer loop
```

Receives a reference to the butterfly image

Note from Listing 3 (p. 1517) that the **cropAndFlip** method receives a reference to the **Picture** object of the butterfly that is displayed in Figure 1 (p. 1512) .

Also note that the butterfly in Figure 1 (p. 1512) is facing toward the right while the butterfly in the output image in Figure 3 (p. 1514) has been cropped to a smaller size and is facing toward the left.

Crop and flip is not new

The capability to crop and flip an image is not new to this module. However, the **cropAndFlip** method also makes the image partially transparent as shown in Figure 3 (p. 1514) . That capability is new to this module. I will explain how that is done shortly.

A call to the modified SimplePicture constructor

Although there is nothing new in the code in Listing 4 (p. 1517) , it is important to note that the first statement in Listing 4 (p. 1517) causes the **SimplePicture** constructor that was modified in Listing 1 (p. 1515) to be called.

As a result, the **Picture** object referred to by the reference variable named **output** in Listing 4 (p. 1517) will accommodate alpha transparency data.

Make the pixels partially transparent

The code in Listing 5 (p. 1518) uses a pair of nested **for** loops to iterate through all of the pixels in the picture referred to by **output** and modify each pixel.

The four statements in the body of the inner loop in Listing 5 (p. 1518) cause the current pixel to become partially transparent.

Listing 6.61: Make the pixels partially transparent.

```
        width = output.getWidth();
    int height = output.getHeight();
    pixel = null;
    color = null;
    for(int col = 0;col < width;col++){
      for(int row = 0;row < height;row++){

        int basicPixel = output.getBasicPixel(col,row);

        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | 0x5F000000;

        output.setBasicPixel(col,row,basicPixel);

      }//end inner loop
    }//end outer loop

    return output;
  }//end crop and flip
```

The getBasicPixel method

According to Ericson's documentation, the **getBasicPixel** method will *"return the pixel value as an int for the given x and y location."* In other words, a call to the **getBasicPixel** method will return an **int** value containing the red, green, blue, and alpha values for the pixel at the specified location.

A bitwise AND operation

Listing 5 (p. 1518) uses a bitwise ***AND*** operation *(note the single ampersand)* to force the eight most significant bits *(the alpha byte)* in the **int** representation of the current pixel to zero while preserving the bit values stored in the least significant 24 bits.

A bitwise OR operation

Then Listing 5 (p. 1518) uses a bitwise ***OR*** operation (|) to store the hexadecimal value 5F in the eight most significant bits *(the alpha byte)* without changing the values stored in the 24 least significant bits.

The alpha byte

The value of the alpha byte can range from 0 to 255. When rendered using a mechanism that supports alpha transparency, an alpha value of zero causes the pixel to be totally transparent.

Similarly, an alpha value of 255 causes the pixel to be totally opaque.

Values between zero and 255 cause the pixel to be rendered as partially opaque or partially transparent, whichever terminology you prefer.

Thirty-seven percent opaque

If I did the arithmetic correctly, a hexadecimal value of 5F represents a decimal value of 95. Therefore, this value will cause the pixel to be about 37-percent opaque *(or 63-percent transparent)* .

The setBasicPixel method

As the name implies, the **setBasicPixel** method can be used to *"set the value of a pixel in the picture from an int."*

Therefore, the last statement in the body of the inner loop in Listing 5 (p. 1518) replaces the value of the current pixel with the modified value containing a value of 95 in the alpha byte.

The end of the cropAndFlip method

When the pair of nested **for** loops in Listing 5 (p. 1518) terminates, the **cropAndFlip** method returns control to the **run** method in Listing 3 (p. 1517) , returning a copy of the reference from the variable named **output** *(see Listing 4 (p. 1517))* in the process.

Save the Picture object's reference

The returned reference is stored in the reference variable named **picC** in Listing 3 (p. 1517) .

At this point, **picC** contains a reference to a butterfly image that has been cropped, flipped, and formatted into a buffered image that contains alpha transparency information.

Call the copyPictureWithCrop method

Listing 3 (p. 1517) immediately calls the **copyPictureWithCrop** method passing copies of the references stored in **picC** and **picB** along with a pair of integer coordinate values.

The copyPictureWithCrop method

The **copyPictureWithCrop** method is shown in its entirety in Listing 6 (p. 1519) .

Listing 6.62: The copyPictureWithCrop method.

```
private void copyPictureWithCrop(
     Picture source,Picture dest,int xOff,
                                 int yOff){

Graphics destGraphics = dest.getGraphics();
Image sourceImage = source.getImage();
destGraphics.drawImage(sourceImage,
                       xOff,
                       yOff,
                       null);
```

```
  }//end copyPictureWithCrop method
}//end class Prob06Runner
```

The purpose of the **copyPictureWithCrop** method is to copy a source picture onto a destination picture with an offset on each axis.

An exercise for the student

I won't attempt to explain the code in Listing 6 (p. 1519) in this module. Instead, I will simply suggest that you go to Google and search for the following or similar keywords:

baldwin java drawImage

You will find many tutorials that I have written that deal with topics in this area.

Modify the destination pixel colors

I will tell you that the use of the **drawImage** method in Listing 6 (p. 1519) modifies the destination picture in such a way that the color of each pixel in the resulting image is a combination of the colors in the original destination image and the corresponding pixel in the source image.

An illusion of transparency

If a source pixel is totally transparent, it has no effect on the color of the destination pixel.

If the source pixel is totally opaque, the color of the destination pixel is changed to the color of the source pixel.

For alpha values between these two extremes, the final color of the destination pixel produces the illusion of a partially transparent image in front of the original destination image.

Termination of the copyPictureWithCrop method

When the **copyPictureWithCrop** method terminates in Listing 6 (p. 1519) , control returns to the run method in Listing 3 (p. 1517) .

Listing 3 (p. 1517) calls the **show** method to display the image in the now-modified **Picture** object referred to by **picB** , as shown in Figure 3 (p. 1514) .

Return a reference to an array object

Then the **run** method encapsulates references to each of the three images in an array object and returns control to the **main** method in Listing 7 (p. 1521) , returning a copy of the array object's reference in the process.

The **main** method in Listing 7 (p. 1521) displays information about each of the three **Picture** objects, producing the output shown in Figure 4 (p. 1515) . Then the main method terminates.

Images don't go away immediately

Because there are images belonging to the program still on the screen, the program doesn't return control to the operating system. It will simply wait until it is forced to terminate by the user before returning control to the operating system.

Clicking the X-buttons in the upper-right corners of the images will simply hide the frames and won't terminate the program. Some extra work is required to deal with this issue.

6.3.2.1.6 Run the program

I encourage you to copy the code from Listing 7 (p. 1521) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.2.1.7 Summary

In this module, you learned about:

- Alpha transparency
- A buffered image of type TYPE_INT_ARGB
- The ability to use the **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND and OR operators,

- The use of the **drawImage** method of the **Graphics** class.

You modified the **SimplePicture** class to cause the buffered image used to store the image to be TYPE_INT_ARGB instead of TYPE_INT_RGB, which is its normal type.

You wrote a method that cropped and flipped an image of a butterfly.

You used the **getBasicPixel** and **setBasicPixel** methods from Ericson's library along with bitwise operations to set the alpha value for all the pixels in the cropped and flipped image of the butterfly to a hexadecimal value of 5F.

Finally, you used the standard **drawImage** method of the **Graphics** class to draw the image of the butterfly onto an image of a beach with transparency.

6.3.2.1.8 What's next?

In the next module, you will learn how to use a slider to continuously change the opacity of an image and to draw that modified image onto a background image.

6.3.2.1.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Using Alpha Transparency with Ericson's Media Library
- File: Java3112.htm
- Published: 05/13/12
- Revised: 09/05/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.2.1.10 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 7 (p. 1521) below.

Listing 6.63: Complete program listing.

```
    /*File Prob06 Copyright 2008 R.G.Baldwin
Revised 12/31/08
*******************************************************/
import java.awt.Color;
```

```
import java.awt.Graphics;
import java.awt.Image;

public class Prob06{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Picture[] pictures = new Prob06Runner().run();
    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob06
//=======================================================//

class Prob06Runner{

  public Prob06Runner(){//constructor
    System.out.println("Dick Baldwin.");
  }//end constructor
  //-------------------------------------------------------//
  public Picture[] run(){
    //Insert executable code here
    Picture picA = new Picture("Prob06a.jpg");
    picA.explore();
    Picture picB = new Picture("Prob06b.jpg");
    picB.addMessage("Dick Baldwin.",10,20);
    picB.explore();

    Picture picC = cropAndFlip(picA,4,5,80,105);
    copyPictureWithCrop(picC,picB,122,70);

    picB.show();

    Picture[] output = {picA,picB,picC};
    return output;
  }//end run
  //-------------------------------------------------------//

  //Crops a picture to the specified coordinate values and
  // flips it around a vertical line at its center.
  //Also makes it partially transparent
  private Picture cropAndFlip(
                Picture pic,int x1,int y1,int x2,int y2){
    Picture output = new Picture(x2-x1+1,y2-y1+1);

    int width = output.getWidth();
    Pixel pixel = null;
    Color color = null;
    for(int col = x1;col < (x2+1);col++){
      for(int row = y1;row < (y2+1);row++){
        color = pic.getPixel(col,row).getColor();
```

```
        pixel = output.getPixel(width-col+x1-1,row-y1);
        pixel.setColor(color);
      }//end inner loop
    }//end outer loop

    width = output.getWidth();
    int height = output.getHeight();
    pixel = null;
    color = null;
    for(int col = 0;col < width;col++){
      for(int row = 0;row < height;row++){

        int basicPixel = output.getBasicPixel(col,row);
        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | 0x5F000000;
        output.setBasicPixel(col,row,basicPixel);
      }//end inner loop
    }//end outer loop


    return output;
  }//end crop and flip
  //-------------------------------------------------//

  //Copies the source picture onto the destination
  // picture with an offset on both axes.
  private void copyPictureWithCrop(
           Picture source,Picture dest,int xOff,
                                       int yOff){

      Graphics destGraphics = dest.getGraphics();
      Image sourceImage = source.getImage();
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end copyPictureWithCrop method
}//end class Prob06Runner
```

-end-

6.3.2.2 Java OOP: Controlling Opacity with a Slider[42]

6.3.2.2.1 Table of Contents

[42] This content is available online at <http://cnx.org/content/m44912/1.3/>.

6.3.2.2.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [43] .

6.3.2.2.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.2.2.2.1.1 Figures

6.3.2.2.2.1.2 Listings

6.3.2.2.3 Preview

The primary objective of this module is to illustrate how to use a slider to continuously change the opacity of an image and to draw that image onto a background image.

Two approaches

This module builds on an earlier module involving transparency. In that module, you learned that there are at least two ways to incorporate alpha transparency into Ericson's media library, The easiest way, which is not necessarily the best way, is to make a relatively simple modification to a constructor in Ericson's **SimplePicture** class. That is the approach used in this module.

[43] http://cnx.org/content/m44148/latest/

The second approach

The second approach is more complicated, but does not require the modification of the classes in Ericson's library. That is probably a better approach due simply to the fact that modifications to Ericson's library are not required. However, that approach is not shown in this module.

Outside research

As with the earlier module, the program that I will explain in this module may require a significant amount of outside research on the part of the student in order to learn about:

- Alpha transparency
- A buffered image of type **TYPE_INT_ARGB**
- The ability to use Ericson's **getBasicPixel** and **setBasicPixel** methods,
- The use of the bitwise AND, OR, and left-shift operators.
- The use of the **drawImage** method of the **Graphics** class.

Modification to the SimplePicture class

The student must modify the **SimplePicture** class to cause the buffered image used to store the image to be **TYPE_INT_ARGB** instead of **TYPE_INT_RGB** , which is its normal type.

Generally speaking, this program:

- Instantiates a new visual object that extends the **JFrame** class and contains a **JSlider** object.
- Instantiates **Picture** objects from two image files *(beach and butterfly)* along with some blank **Picture** objects of the same size.
- Defines a method named **setOpacity** that can be called to set the opacity of every pixel in a picture to a specified value.
- Defines a method named **drawPictureOnPicture** that can be called to draw one picture onto another picture.
- Registers a **ChangeEvent** handler on the slider to:
  - Extract a percent-opacity value from the slider based on the position of the thumb.
  - Apply that opacity value to the butterfly image.
  - Draw the modified butterfly image on the beach image and display it.

Brief program specifications

Write a program named **Prob07** that uses the class definition for the class named **Prob07** in Listing 10 (p. 1535) along with Ericson's media library and the image files named Prob07a.jpg [44] and Prob07b.jpg [45] to produce the two output images shown in Figure 1 (p. 1526) .

[44] http://cnx.org/content/m44912/latest/Prob07a.jpg
[45] http://cnx.org/content/m44912/latest/Prob07b.jpg

Screen output at startup.

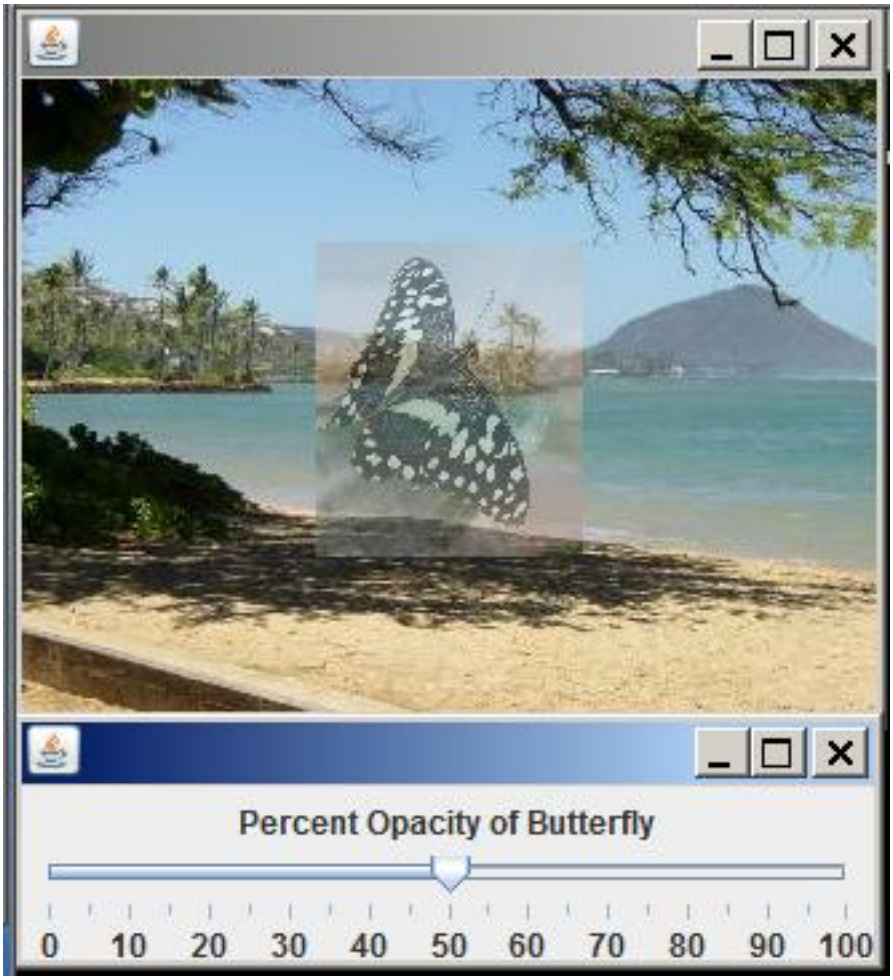


Figure 6.20: Screen output at startup.

Two output images

Note that Figure 1 (p. 1526) actually consists of two output images, one positioned below the other.

Move the thumb to the left

When you move the thumb on the slider to the left, the butterfly becomes less opaque *(more transparent)* as shown in Figure 2 (p. 1527) with total transparency at the extreme left end of the slider.

Twenty-percent opacity.

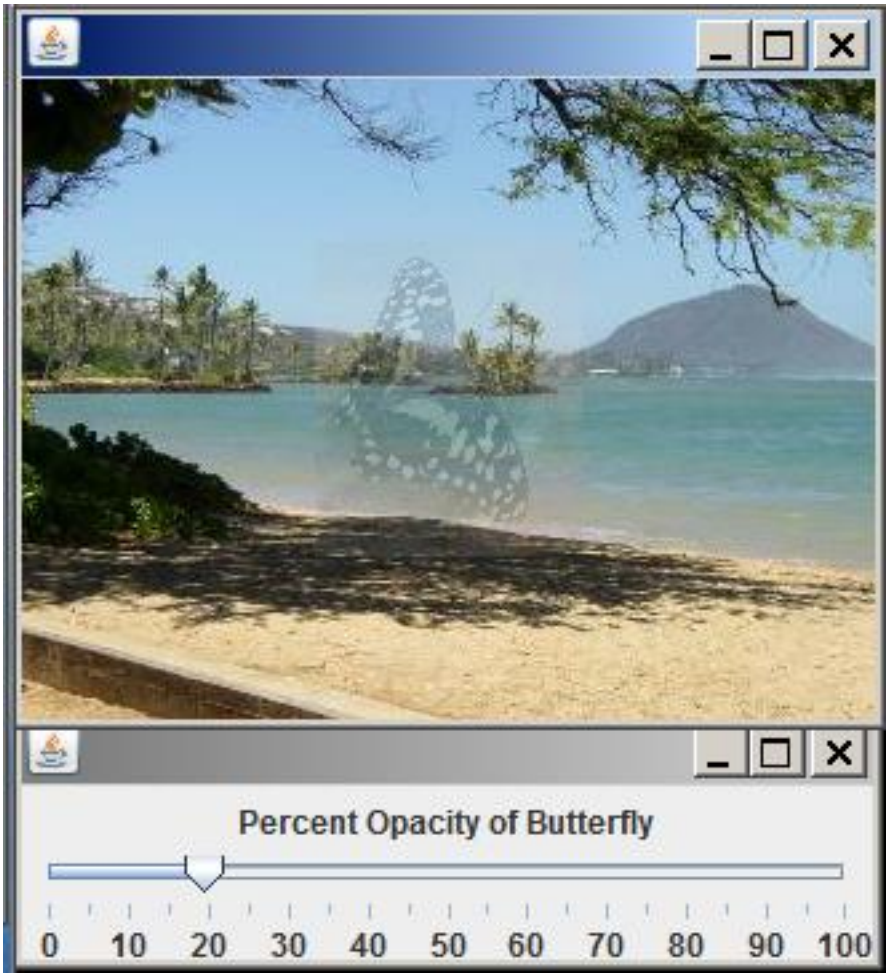


Figure 6.21: Twenty-percent opacity.

Move the thumb to the right

When you move the thumb on the slider to the right, the butterfly becomes more opaque *(less transparent)* as shown in Figure 3 (p. 1528) with total opacity at the extreme right end of the slider.

Eighty-percent opacity.

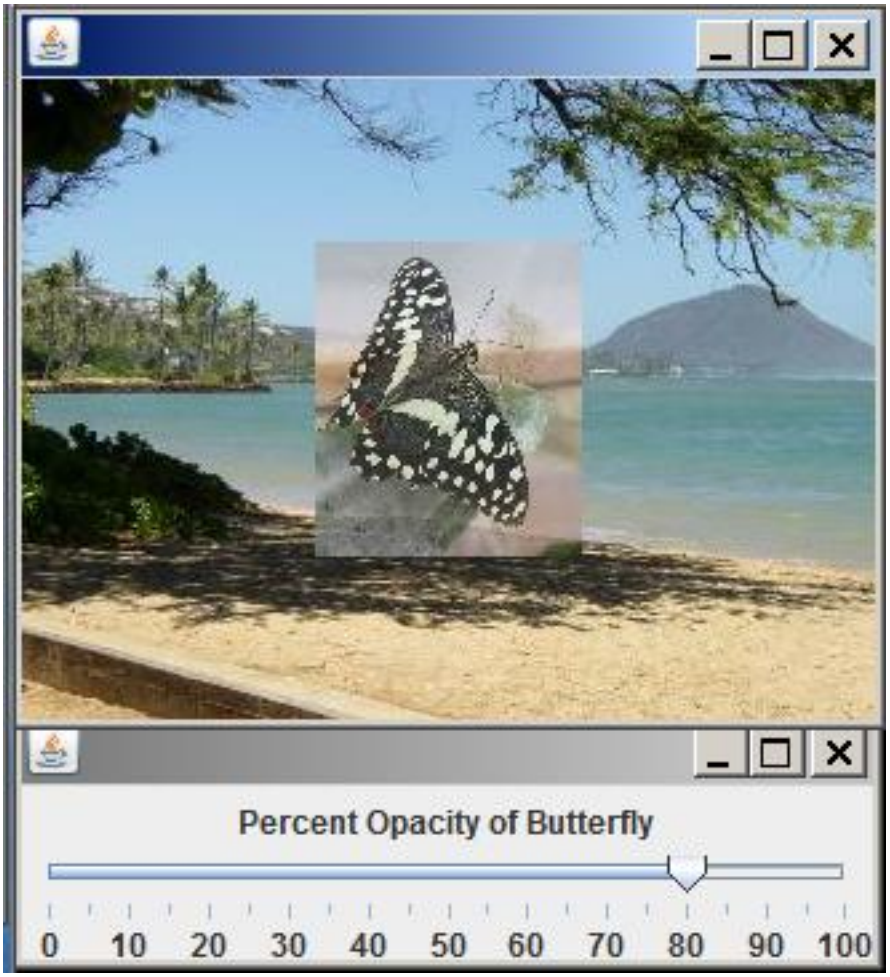


Figure 6.22: Eighty-percent opacity.

Define new classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob07** given in Listing 10 (p. 1535) .

6.3.2.2.4 General background information

The image in a **SimplePicture** object is stored in an object of the **BufferedImage** class, which is a class in the standard Sun Java library.

Image data formats

An examination of the documentation for the **BufferedImage** class shows that the red, green, blue, and alpha values for each pixel can be formatted in about fourteen different ways in an object of the **BufferedImage** class.

No alpha data

Some of those formats, including the way that information is stored in a **SimplePicture** object, don't include an alpha value.

Modification of the SimplePicture class

One way to modify the **SimplePicture** class to force it to accommodate alpha transparency data is to modify one of the constructors for the **SimplePicture** class as shown in Listing 1 (p. 1529) . Note the change indicated by comments in Listing 1 . *(There are probably other ways that you can modify the class to achieve the same result as well.)*

Listing 6.64: Modification of the SimplePicture class.

```
    /**
  * A constructor that takes the width and height desired
  * for a picture and creates a buffered image of that
  * size.  This constructor doesn't show the picture.
  */
 public  SimplePicture(int width, int height){
//Disable the following statement
//   bufferedImage = new BufferedImage(
//            width, height, BufferedImage.
```

Future Picture objects will accommodate alpha transparency

Having made this modification, future objects instantiated from the **SimplePicture** class using this constructor will accommodate alpha transparency. *(The **SimplePicture** class is the superclass of the **Picture** class.)*

No complete listing of SimplePicture provided

Because of the simplicity of this modification, a complete listing of the modified **SimplePicture** class will not be provided in this module.

6.3.2.2.5 Discussion and sample code

6.3.2.2.5.1 The class named Prob07

You can view the driver class named **Prob07** at the beginning of the source code in Listing 10 (p. 1535) . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Briefly, the **main** method instantiates a new object of the class named **Prob07Runner** and calls the **run** method on that object. When the **run** method returns, the GUI shown in Figure 1 (p. 1526) has been displayed on the screen.

At that point, the program simply goes into an idle state and waits for the user to take some action that causes an event to be fired. When an event is fired, it is handled and the program goes idle again waiting for another event.

(Because there are images on the screen, the program does not actually terminate until the user forces it to terminate.)

6.3.2.2.5.2 The class named Prob07Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 10 (p. 1535) near the end of the module.

Beginning of the class named Prob07Runner

The class named **Prob07Runner** begins in Listing 2 (p. 1530) .

Listing 6.65: Beginning of the class named Prob07Runner.

```
class Prob07Runner extends JFrame{

private JPanel mainPanel = new JPanel();
private JPanel titlePanel = new JPanel();
private JSlider slider = new JSlider();

private Picture background = new Picture("Prob07b.jpg");
private Picture butterfly = new Picture("Prob07a.jpg");


private int backgroundWidth = background.getWidth();
private int backgroundHeight = background.getHeight();
private int butterflyWidth = butterfly.getWidth();
private int butterflyHeight = butterfly.getHeight();

private Picture display =
          new Picture(backgroundWidth,backgroundHeight);
private Picture tempPicture =
            new Picture(butterflyWidth,butterflyHeight);

private Image image = null;
private Graphics graphics = null;
```

Class extends JFrame

Note that this class extends **JFrame** . An object of this class forms the lower part of the image shown in Figure 1 (p. 1526) that contains the slider.

The code in Listing 1 (p. 1529) is straightforward and shouldn't require an explanation.

When Listing 2 finishes executing...

When the code in Listing 2 (p. 1530) has finished executing, four new **Picture** objects have been instantiated and referred to by the following reference variables:

- **background** - The beach scene shown in the background in Figure 1 (p. 1526) .
- **butterfly** - Contains an opaque image of the butterfly shown in Figure 1 (p. 1526) .
- **display** - Empty picture the same size as the beach scene.
- **tempPicture** - Empty picture the same size as the butterfly.

In addition, a pair of working variables named **image** and **graphics** of the types **Image** and **Graphics** have been declared.

Finally, when the code in Listing 2 (p. 1530) has finished executing, two new **JPanel** objects and one new **JSlider** object have been instantiated and referred to by the variables named **mainPanel** , **titlePanel** , and **slider** .

Beginning of the constructor

The beginning of the constructor is shown in Listing 3 (p. 1530) .

Listing 6.66: Beginning of the constructor.

```
  public Prob07Runner(){//constructor
//Do some initial setup.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(5);
slider.setPaintTicks(true);
slider.setPaintLabels(true);

mainPanel.setLayout(new BorderLayout());
titlePanel.add(new JLabel(
                     "Percent Opacity of Butterfly"));
mainPanel.add(titlePanel,BorderLayout.NORTH);
mainPanel.add(slider,BorderLayout.CENTER);

getContentPane().add(mainPanel);

setSize(backgroundWidth + 7,97);
setLocation(0,backgroundHeight + 25);
setVisible(true);
```

Although it may be necessary for you to go to Sun's Java documentation to learn about the detailed behavior of some of the methods that are called in Listing 3 (p. 1530) , the code in Listing 3 (p. 1530) is straightforward and should not require further explanation.

Display the initial background image

Listing 4 (p. 1531) displays the initial background image.

Instantiating and destroying a lot of new **Picture** objects as the user moves the slider to change the opacity would be very inefficient. To avoid this inefficiency, this program gets images from existing **Picture** objects and draws them on existing **Picture** objects without modifying the originals.

Listing 6.67: Display the initial image.

```
    graphics = display.getGraphics();
graphics.drawImage(background.getImage(),0,0,null);
```

Display the butterfly at 50-percent opacity

Listing 5 (p. 1531) calls the **setOpacity** and **drawPictureOnPicture** methods to set the opacity of the butterfly and draw it onto the display with 50-percent opacity. The image of the butterfly is centered on the background.

Listing 6.68: Display the butterfly at 50-percent opacity.

```
butterfly = setOpacity(butterfly,50);
drawPictureOnPicture(
              butterfly,
              display,
              backgroundWidth/2 - butterflyWidth/2,
              backgroundHeight/2 - butterflyHeight/2);
display.show();
```

Put the constructor on hold

At this point, I will put the discussion of the constructor on hold and explain the **setOpacity** and **drawPictureOnPicture** methods.

The setOpacity method

The **setOpacity** method is shown in its entirety in Listing 6 (p. 1532) .

Listing 6.69: The setOpacity method.

```
private Picture setOpacity(
                Picture pic,double percentOpacity){

  int opacity = (int)(255*percentOpacity/100);
  int opacityMask = opacity << 24;

  for(int col = 0;col < butterflyWidth;col++){
    for(int row = 0;row < butterflyHeight;row++){
      //Get the pixel in basic int format.
      int basicPixel = pic.getBasicPixel(col,row);

      //Set the alpha value for the pixel.
      basicPixel = basicPixel & 0x00FFFFFF;
      basicPixel = basicPixel | opacityMask;

      //Set the modified pixel into tempPicture.
      tempPicture.setBasicPixel(col,row,basicPixel);
    }//end inner loop
  }//end outer loop

  return tempPicture;

}//end setOpacity
```

This method copies an incoming picture into an existing temporary picture, setting the alpha value for every pixel to a specified value in the process. Then it returns the modified picture object's reference where it is saved in the reference variable named **butterfly** in Listing 5 (p. 1531) ,

A bitwise left-shift operation

The only thing in Listing 6 (p. 1532) that is new to this module is the use of a bitwise left-shift operation.

A 24-bit left shift

Listing 6 (p. 1532) converts the incoming **percentOpacity** value to an integer value ranging from 0 to 255. This value resides in the least significant eight bits of an **int** variable named **opacity** .

Then Listing 6 (p. 1532) applies the bitwise left-shift operator *(two left angle brackets)* to shift those eight bits into the eight most significant bits and stores the result in another **int** variable named **opacityMask** .

Apply the opacityMask to the pixels

A pair of nested **for** loops is used to set the alpha value of every pixel to the value of **opacityMask** using an overall bit-masking methodology that I explained in an earlier module.

The drawPictureOnPicture method

After the alpha value for every pixel in the butterfly image has been set to the specified opacity, Listing 5 (p. 1531) calls the method named **drawPictureOnPicture** to draw the modified butterfly image on the beach scene as shown in Figure 1 (p. 1526) .

The **drawPictureOnPicture** method is shown in its entirety in Listing 7 (p. 1533) .

Listing 6.70: The drawPictureOnPicture method.

```
  private void drawPictureOnPicture(
                Picture source,Picture dest,int xOff,
                                          int yOff){

  Graphics destGraphics = dest.getGraphics();
  Image sourceImage = source.getImage();
  destGraphics.drawImage(sourceImage,
                         xOff,
                         yOff,
                         null);
}//end drawPictureOnPicture method
```

This method draws the source picture onto the destination picture with an offset on both axes. There is nothing in Listing 7 (p. 1533) that I haven't explained in an earlier module.

Return to the explanation of the constructor

You are already familiar with the use of anonymous inner classes to create and register listener objects on Java source objects. The slider is a source object.

Listing 8 (p. 1533) begins the registration of an anonymous **ChangeEvent** listener on the slider.

Listing 6.71: Begin the registration of an event handler on the slider.

```
    slider.addChangeListener(
  new ChangeListener(){
    public void stateChanged(ChangeEvent e){
      //Draw a new copy of the background on the
      // display.
      graphics = display.getGraphics();
      graphics.drawImage(
                     background.getImage(),0,0,null);
```

Restore the background image

Each time the slider fires a **ChangeEvent** , this event handler draws a new background image on the display. This erases what was previously drawn there, restoring a pristine image of the beach scene.

Draw a partially opaque butterfly image on the background

Then it uses the current value of the slider to set the opacity of the butterfly image and draws it centered on the display on top of the background image.

A series of events

The slider fires a series of **ChangeEvents** as the user moves the thumb on the slider. Listing 8 (p. 1533) begins the definition of the event handler method named **stateChanged** , which is registered on the slider. This method is called each time the slider fires a **ChangeEvent** .

Listing 8 (p. 1533) draws a new copy of the beach background image on the **Picture** object referred to by the reference variable named **background** . This image replaces the image that was previously drawn there.

Draw the butterfly and repaint

Listing 9 (p. 1534) calls the **setOpacity** and **drawPictureOnPicture** methods to:

- Set the opacity of the butterfly to the value currently represented by the position of the thumb on the slider. This is the value returned by the slider's **getValue** method.
- Draw the butterfly image on the background image.

Listing 6.72: Draw the butterfly and repaint.

```
            //Set the opacity of butterfly and copy it onto
          // the display. Then repaint the display.
          butterfly =
                  setOpacity(butterfly,slider.getValue());
          drawPictureOnPicture(
                  butterfly,
                  display,
                  backgroundWidth/2 - butterflyWidth/2,
                  backgroundHeight/2 - butterflyHeight/2);

          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------------//
  }//end constructor
```

Repaint the image

Then Listing 9 (p. 1534) calls the **repaint** method to cause the modified image to be rendered onto the computer screen.

The end of the program

Listing 9 (p. 1534) also signals the end of the constructor, the end of the class named **Prob07Runner** , and the end of the program.

6.3.2.2.6 Run the program

I encourage you to copy the code from Listing 10 (p. 1535) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.2.2.7 Summary

In this module, you learned how to use a slider to continuously change the opacity of an image and draw that image onto a background image.

6.3.2.2.8 What's next?

In the next module, you will learn how to use a slider to continuously change the threshold detection level of an edge detector and to draw the edge-detected image on the screen.

6.3.2.2.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Controlling Opacity with a Slider

- File: Java3114.htm
- Published: 05/13/12
- Revised: 09/06/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.2.2.10 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 1535) below.

Listing 6.73: Complete program listing.

```
    /*File Prob07 Copyright 2008 R.G.Baldwin
**********************************************************/
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class Prob07{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob07Runner();
  }//end main method
}//end class Prob07
//=========================================================//

class Prob07Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
```

```
  private JSlider slider = new JSlider();

  private Picture background = new Picture("Prob07b.jpg");
  private Picture butterfly = new Picture("Prob07a.jpg");


  private int backgroundWidth = background.getWidth();
  private int backgroundHeight = background.getHeight();
  private int butterflyWidth = butterfly.getWidth();
  private int butterflyHeight = butterfly.getHeight();

  private Picture display =
            new Picture(backgroundWidth,backgroundHeight);
  private Picture tempPicture =
              new Picture(butterflyWidth,butterflyHeight);

  private Image image = null;
  private Graphics graphics = null;

  public Prob07Runner(){//constructor
    //Do some initial setup.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                         "Percent Opacity of Butterfly"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(backgroundWidth + 7,97);
    setLocation(0,backgroundHeight + 25);
    setVisible(true);

    //Draw and display the initial image with 50-percent
    // opacity. In order to avoid instantiating and
    // destroying a lot of Picture objects, the
    // procedure is to simply get images from existing
    // picture objects and draw them on other existing
    // picture objects.
    graphics = display.getGraphics();
    graphics.drawImage(background.getImage(),0,0,null);

    //Set the opacity of butterfly and draw it onto the
    // display. In this case, the opacity is set to
```

```
  // 50-percent. The image of the butterfly is centered
  // on the background.
  butterfly = setOpacity(butterfly,50);
  drawPictureOnPicture(
               butterfly,
               display,
               backgroundWidth/2 - butterflyWidth/2,
               backgroundHeight/2 - butterflyHeight/2);
  display.show();
  //-------------------------------------------------//
  //Register an anonymous listener object on the slider.
  //Each time the slider fires a ChangeEvent, this event
  // handler draws a new background image on the
  // display. This erases what was previously drawn
  // there. Then it uses the current value of the slider
  // to set the opacity of the butterfly image and
  // draws it on the display on top of the background
  // image. It is centered on the background image.
  slider.addChangeListener(
    new ChangeListener(){
      public void stateChanged(ChangeEvent e){
        //Draw a new copy of the background on the
        // display.
        graphics = display.getGraphics();
        graphics.drawImage(
                        background.getImage(),0,0,null);

        //Set the opacity of butterfly and copy it onto
        // the display. Then repaint the display.
        butterfly =
                setOpacity(butterfly,slider.getValue());
        drawPictureOnPicture(
                butterfly,
                display,
                backgroundWidth/2 - butterflyWidth/2,
                backgroundHeight/2 - butterflyHeight/2);
        display.repaint();
      }//end stateChanged
    }//end new ChangeListener
  );//end addChangeListener
  //-------------------------------------------------//
}//end constructor
//-------------------------------------------------//


//This method copies an incoming picture into an
// existing temporary picture, setting the alpha value
// for every pixel to a specified value. Then it returns
// the modified temporary picture object.
private Picture setOpacity(
                     Picture pic,double percentOpacity){
```

```
    int opacity = (int)(255*percentOpacity/100);
    int opacityMask = opacity << 24;

    for(int col = 0;col < butterflyWidth;col++){
      for(int row = 0;row < butterflyHeight;row++){
        //Get the pixel in basic int format.
        int basicPixel = pic.getBasicPixel(col,row);

        //Set the alpha value for the pixel.
        basicPixel = basicPixel & 0x00FFFFFF;
        basicPixel = basicPixel | opacityMask;

        //Set the modified pixel into tempPicture.
        tempPicture.setBasicPixel(col,row,basicPixel);
      }//end inner loop
    }//end outer loop

    return tempPicture;

  }//end setOpacity
  //-----------------------------------------------------//

  //Draws the source picture onto the destination
  // picture with an offset on both axes.
  private void drawPictureOnPicture(
                    Picture source,Picture dest,int xOff,
                                              int yOff){

      Graphics destGraphics = dest.getGraphics();
      Image sourceImage = source.getImage();
      destGraphics.drawImage(sourceImage,
                             xOff,
                             yOff,
                             null);
  }//end drawPictureOnPicture method
}//end class Prob07Runner
```

-end-

6.3.2.3 Java OOP: Controlling an Edge Detector with a Slider[46]

6.3.2.3.1 Table of Contents

[46] This content is available online at <http://cnx.org/content/m44913/1.2/>.

6.3.2.3.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [47] .

6.3.2.3.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.2.3.2.1.1 Figures

6.3.2.3.2.1.2 Listings

6.3.2.3.3 Preview

An earlier module *(see 3D Displays, Color Distance, and Edge Detection [48])* explained how to implement an edge detection algorithm for a fixed detection threshold.

The primary objective of this module is to illustrate how to use a slider to continuously change the detection threshold of an edge detector and to draw the edge-detected image on the screen.

Brief program specifications

Write a program named **Prob08** that uses the class definition for the class named **Prob08** shown in Listing 6 (p. 1546) and Ericson's media library along with the image file named Prob08.jpg [49] *(shown in Figure 1 (p. 1540))* to produce the two graphic output images shown in Figure 2 (p. 1541) .

[47] http://cnx.org/content/m44148/latest/
[48] http://www.developer.com/java/other/article.php/3798646/3D-Displays-Color-Distance-and-Edge-Detection.htm
[49] http://cnx.org/content/m44913/latest/Prob08.jpg

Contents of the file named Prob08.jpg.

Figure 6.23: Contents of the file named Prob08.jpg.

Output images.

Figure 6.24: Output images.

Note that Figure 2 (p. 1541) consists of two separate output images. The image containing the slider is positioned directly below the image of the butterfly.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob08** given in Listing 6 (p. 1546) .

The output images

The top image shown in Figure 2 (p. 1541) is an image of a butterfly to which an edge detection algorithm has been applied.

The bottom image in Figure 2 (p. 1541) is a slider that is used to control the edge-detection threshold.

Detect by rows and by columns

The edge-detection algorithm performs edge detection on a **Picture** object by rows and also by columns.

All edges that are detected by processing adjacent pixels on a row are marked in red. All edges that are detected by processing adjacent pixels on a column are marked in black.

If a pixel is determined to be on an edge using both approaches, it ends up being black. If an edge is not detected, the corresponding pixel is marked in white.

Program behavior

At startup, the thumb on the slider is positioned at the 50-percent mark and the image has been edge-detected using a threshold value of 50.

As you move the slider to the right, the threshold value increases up to a value of 100, which in turn causes the amount of white area in the image to increase.

As you move the slider to the left, the threshold value decreases down to a value of zero, which in turn causes the amount of white area in the image to decrease.

The program must terminate and return control to the operating system when you click the large X in the upper-right corner of the GUI containing the slider.

In addition to the output images described above, your program must display your name on the command-line screen.

6.3.2.3.4 Discussion and sample code

6.3.2.3.4.1 The class named Prob08

You can view the driver class named **Prob08** at the beginning of the source code in Listing 6 (p. 1546) . You are already familiar with the code in the **main** method of that class from earlier modules so I won't spend any time explaining it.

Brief description

Briefly, the **main** method instantiates a new object of the class named **Prob08Runner** and calls the **run** method on that object. When the **run** method returns, the GUI shown in Figure 2 (p. 1541) has been displayed on the screen.

At that point, the program simply goes into an idle state and waits for the user to take some action that causes an event to be fired. When an event is fired, it is handled and the program goes idle again waiting for another event.

(Because there are images on the screen, the program does not terminate until the user forces it to terminate.)

6.3.2.3.4.2 The class named Prob08Runner

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 6 (p. 1546) near the end of the module.

Beginning of the class named Prob08Runner

The class named **Prob08Runner** begins in Listing 1 (p. 1542) .

Listing 6.74: Beginning of the class named Prob08Runner.

```
class Prob08Runner extends JFrame{

private JPanel mainPanel = new JPanel();
private JPanel titlePanel = new JPanel();
private JSlider slider = new JSlider();

private Picture butterfly =
                      new Picture("Prob08.jpg");
```

```
  private int butterflyWidth =
                              butterfly.getWidth();
  private int butterflyHeight =
                             butterfly.getHeight();

  private Picture display =
       new Picture(butterflyWidth,butterflyHeight);

  private Pixel pix1;
  private Pixel pix2;
  private Pixel displayPixel;

  private double distance = 0;
```

There is nothing in Listing 1 (p. 1542) that I haven't explained in earlier modules, so I won't repeat those explanations here.

Beginning of the constructor

The constructor begins in Listing 2 (p. 1543) .

Listing 6.75: Beginning of the constructor.

```
    public Prob08Runner(){//constructor
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  slider.setMajorTickSpacing(10);
  slider.setMinorTickSpacing(5);
  slider.setPaintTicks(true);
  slider.setPaintLabels(true);

  mainPanel.setLayout(new BorderLayout());
  titlePanel.add(new JLabel(
                    "Edge Detection Threshold"));
  mainPanel.add(titlePanel,BorderLayout.NORTH);
  mainPanel.add(slider,BorderLayout.CENTER);

  getContentPane().add(mainPanel);

  setSize(butterflyWidth + 7,97);
  setLocation(0,butterflyHeight + 25);
  setVisible(true);

  //Produce the initial display with a threshold
  // value of 50, which matches the initial
  // position of the pointer on the slider.
  display = edgeDetector(butterfly,50);

  display.show();
```

Put the constructor on hold

The only thing that is new in Listing 2 (p. 1543) is the call to the method named **edgeDetector** . I will put the explanation of the constructor on hold while I explain the method named **edgeDetector** .

Beginning of the edgeDetector method

The **edgeDetector** method begins in Listing 3 (p. 1544) .

This method performs edge detection on a **Picture** object by rows and also by columns.

As mentioned earlier, all edges that are detected by processing adjacent pixels on a row are marked in red.

All edges that are detected by processing adjacent pixels on a column are marked in black.

If a pixel is determined to be on an edge using both approaches, it ends up being black.

Listing 6.76: Beginning of the edgeDetector method.

```
  private Picture edgeDetector(
                  Picture picture,int threshold){
for(int row = 0;row < butterflyHeight - 1;row++){
  for(int col = 0;col < butterflyWidth - 1;col++){
    pix1 = picture.getPixel(col,row);
    displayPixel = display.getPixel(col,row);

    //First process two adjacent pixels on the
    // same row.
    pix2 = picture.getPixel(col + 1,row);

    //Get and save the color distance between the
    // two pixels.
    distance = pix1.colorDistance(pix2.getColor());

    //Compare the color distance to the threshold
    // and set the color of the pixel in the
    // display picture accordingly.
    if(distance > threshold){
      displayPixel.setColor(Color.RED);
    }else{
      displayPixel.setColor(Color.WHITE);
    }//end else
```

A pair of nested for loops

Listing 3 (p. 1544) shows the beginning of a pair of nested **for** loops that I used to detect the edges in the image of the butterfly. The code in Listing 3 (p. 1544) detects edges using adjacent pixels on each row and sets the color of the corresponding pixel in the output picture to red if an edge is detected.

(Note that the output picture referred to by ***display*** *was instantiated as an instance variable in Listing 1 (p. 1542) .)*

I explained an edge-detection algorithm in great detail in an earlier module titled 3D Displays, Color Distance, and Edge Detection [50] . The code in Listing 3 (p. 1544) is essentially an implementation of that algorithm where the detection threshold is an input parameter to the method.

Detect using adjacent pixels in each column

Listing 4 (p. 1544) detects edges using adjacent pixels on the columns.

Listing 6.77: Detect using adjacent pixels in each column.

[50] http://www.developer.com/java/other/article.php/3798646/3D-Displays-Color-Distance-and-Edge-Detection.htm

```
          //Now process two adjacent pixels in the same
        // column using the same approach.
        pix2 = picture.getPixel(col,row + 1);
        distance = pix1.colorDistance(pix2.getColor());
        //Compare the color distance to the threshold
        // and change pixel color accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.BLACK);
        }//end if

      }//end inner loop
    }//end outer loop

    return display;
  }//end edgeDetector
```

If an edge is detected in Listing 4 (p. 1544) , the color of the corresponding pixel in the output picture is set to black.

Return the output picture and terminate

The **edgeDetector** method returns a reference to the output picture and terminates in Listing 4 (p. 1544) .

(While writing this, I realized that because the variable named ***display*** *is an instance variable, the program would also work properly if the* ***edgeDetector*** *method were to return void.)*

Returning to the constructor...

Returning to where we left off in Listing 2 (p. 1543) , the **show** method is called on the output picture referred to by **display** causing an image similar to that shown in Figure 2 (p. 1541) to be displayed.

Register a ChangeEvent listener on the slider

Listing 5 (p. 1545) registers an anonymous listener object on the slider. Each time the slider fires a **ChangeEvent** , the method named **stateChanged** is executed.

Listing 6.78: Register a ChangeEvent listener on the slider.

```
        slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          display = edgeDetector(
                      butterfly,slider.getValue());
          display.repaint();

        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------//
  }//end constructor
  //-------------------------------------------//
}//end class Prob08Runner
```

Call the edgeDetector method

The **stateChanged** method calls the **edgeDetector** method to get a new edge-detected image for which the threshold is the current value of the slider.

Then the display is repainted showing the new image on the screen.

The end of the program

Listing 5 (p. 1545) also signals the end of the constructor, the end of the class, and the end of the program.

6.3.2.3.5 Run the program

I encourage you to copy the code from Listing 6 (p. 1546) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.2.3.6 Summary

In this module, you learned how to use a slider to continuously change the threshold detection level of an edge detector and to draw the edge-detected image on the screen.

6.3.2.3.7 What's next?

In the next module, you will learn how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

6.3.2.3.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Controlling an Edge Detector with a Slider
- File: Java3116.htm
- Published: 05/13/12
- Revised: 09/06/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.2.3.9 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 6 (p. 1546) .

Listing 6.79: Complete program listing.

```
    /*File Prob08 Copyright 2008 R.G.Baldwin
 *Revised 12/31/08

*******************************************************/
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.awt.BorderLayout;
import java.awt.Color;

public class Prob08{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob08Runner();
  }//end main method
}//end class Prob08
//=======================================================//

class Prob08Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider();

  private Picture butterfly = new Picture("Prob08.jpg");

  private int butterflyWidth = butterfly.getWidth();
  private int butterflyHeight = butterfly.getHeight();

  private Picture display =
              new Picture(butterflyWidth,butterflyHeight);

  private Pixel pix1;
  private Pixel pix2;
  private Pixel displayPixel;

  private double distance = 0;

  public Prob08Runner(){//constructor
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
```

```
                            "Edge Detection Threshold"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);

    setSize(butterflyWidth + 7,97);
    setLocation(0,butterflyHeight + 25);
    setVisible(true);

    //Produce the initial display with a threshold value
    // of 50, which matches the initial position of the
    // pointer on the slider.
    display = edgeDetector(butterfly,50);

    display.show();
    //----------------------------------------------------//
    //Register an anonymous listener object on the slider.
    //Each time the slider fires a ChangeEvent, this event
    // handler calls the edgeDetector method to get a new
    // edge-detected image for which the threshold is the
    // current value of the slider. Then the display is
    // repainted showing the new image.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          display = edgeDetector(
                              butterfly,slider.getValue());
          display.repaint();

        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //----------------------------------------------------//
  }//end constructor
  //------------------------------------------------------//
  /*This method performs edge detection on a Picture
   *object by rows and also by columns.
   *All edges that are detected by processing adjacent
   *pixels on a row are marked in red.
   *All edges that are detected by processing adjacent
   *pixels on a column are marked in black.
   *If a pixel is determined to be on an edge using both
   *approaches, it ends up being black.
  */

  private Picture edgeDetector(
                            Picture picture,int threshold){
    for(int row = 0;row < butterflyHeight - 1;row++){
      for(int col = 0;col < butterflyWidth - 1;col++){
```

```
        pix1 = picture.getPixel(col,row);
        displayPixel = display.getPixel(col,row);

        //First process two adjacent pixels on the same
        // row.
        pix2 = picture.getPixel(col + 1,row);

        //Get and save the color distance between the two
        // pixels.
        distance = pix1.colorDistance(pix2.getColor());

        //Compare the color distance to the threshold and
        // set the color of the pixel in the display
        // picture accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.RED);
        }else{
          displayPixel.setColor(Color.WHITE);
        }//end else

        //Now process two adjacent pixels in the same
        // column using the same approach.
        pix2 = picture.getPixel(col,row + 1);
        distance = pix1.colorDistance(pix2.getColor());
        //Compare the color distance to the threshold and
        // change pixel color accordingly.
        if(distance > threshold){
          displayPixel.setColor(Color.BLACK);
        }//end if

      }//end inner loop
    }//end outer loop

    return display;
  }//end edgeDetector
}//end class Prob08Runner
```

-end-

6.3.2.4 Java OOP: Controlling an Image-Scaling Program with a Slider[51]

6.3.2.4.1 Table of Contents

[51] This content is available online at <http://cnx.org/content/m44914/1.2/>.

6.3.2.4.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [52] .

6.3.2.4.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.2.4.2.1.1 Figures

6.3.2.4.2.1.2 Listings

6.3.2.4.3 Preview

The primary objective of this module is to illustrate how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

Brief program specifications

Write a program named **Prob09** that uses the class definition for the class named **Prob09** shown in Listing 3 (p. 1556) and Ericson's media library along with the image files named Prob09a.jpg [53] *(see Figure 1 (p. 1551))* and Prob09b.jpg [54] *(see Figure 2 (p. 1552))* to produce the graphic output images shown in Figure 3 (p. 1553) .

[52] http://cnx.org/content/m44148/latest/
[53] http://cnx.org/content/m44914/latest/Prob09a.jpg
[54] http://cnx.org/content/m44914/latest/Prob09b.jpg

Prob09a.jpg.

Figure 6.25: Prob09a.jpg.

Prob09b.jpg.

Figure 6.26: Prob09b.jpg.

Output images.

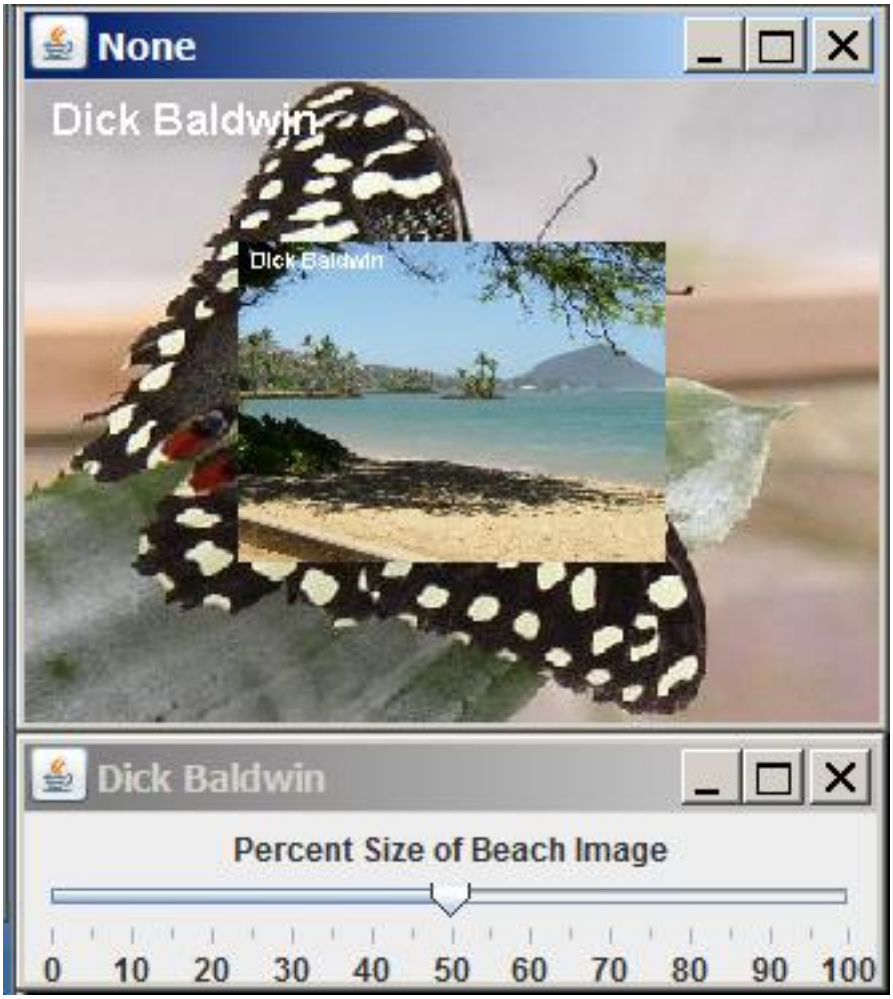


Figure 6.27: Output images.

6.3.2.4.4 General background information

The overall structure of this program is very similar to a program that I explained in an earlier module titled Controlling an Edge Detector with a Slider [55] . The only significant difference between the two programs is the code that is executed when the slider fires a **ChangeEvent** .

I will explain the code that is new and different in this program and refer you back to the earlier module for an explanation of the remainder of the code. You can view the entire program in Listing 3 (p. 1556) near the end of the module.

[55] http://cnx.org/content/m44913/latest

6.3.2.4.5 Discussion and sample code

Register a listener object on the slider

The code in Listing 1 (p. 1554) registers an anonymous listener object on the slider.

Listing 6.80: Register a listener object on the slider.

```
    slider.addChangeListener(
  new ChangeListener(){
    public void stateChanged(ChangeEvent e){

      //Restore the background image of the butterfly.
      graphics = display.getGraphics();
      graphics.drawImage(
                     butterfly.getImage(),0,0,null);

      drawScaledPictureOnPicture(beach,
                                 display,
                                 slider.getValue());
      display.repaint();
    }//end stateChanged
  }//end new ChangeListener
);//end addChangeListener
```

Each time the slider fires a **ChangeEvent** , the method named **stateChanged** is executed.

Behavior of the stateChanged method

The **stateChanged** method restores the background image of the butterfly. Then it calls the method named **drawScaledPictureOnPicture** to draw a scaled version of the beach on top of the background image using the slider value as the scale factor.

The slider value ranges from 0 to 100. This represents a scale factor as a percent of 1.0. In other words, the beach image is never scaled to a size that is larger than the size of the original beach image.

The image of the beach is always aligned with the center of the **contentPane** of the **JFrame** .

When the **drawScaledPictureOnPicture** method returns, the **repaint** method is called to cause the new image to be rendered on the computer screen.

The method named drawScaledPictureOnPicture

The method named **drawScaledPictureOnPicture** is shown in its entirety in Listing 2 (p. 1554) .

Listing 6.81: The method named drawScaledPictureOnPicture.

```
  private void drawScaledPictureOnPicture(
                                  Picture source,
                                  Picture dest,
                                  double scaleFactor){

transform = new AffineTransform();

double translateX = dest.getWidth()/2
               - source.getWidth()*scaleFactor/100/2;
double translateY = dest.getHeight()/2
             - source.getHeight()*scaleFactor/100/2;;
transform.translate(translateX,translateY);
transform.scale(scaleFactor/100.0,scaleFactor/100.0);
```

```
    //Get the Graphics2D object used to draw on the
    // destination picture.
    g2 = (Graphics2D)dest.getGraphics();

    //Scale and draw the source image on the destination
    // image.
    g2.drawImage(source.getImage(),transform,null);

  }//end drawScaledPictureOnPicture method
```

Use an affine transform

This method uses an affine transform to first translate and then scale the source picture and draws the scaled source picture onto the center of the destination picture.

I have published numerous tutorials that explain the use of Affine transforms in Java, including Applying Affine Transforms to Picture Objects [56] . I won't repeat those explanations in this module, but will simply refer you to the earlier tutorials.

Keywords for a Google search

You can find my other tutorials that explain Affine transforms by using Google to search for the following keywords:

```
null
```

6.3.2.4.6 Run the program

I encourage you to copy the code from Listing 3 (p. 1556) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.2.4.7 Summary

In this module, you learned how to use a slider to continuously change the size of an image and to draw the scaled image onto a background image.

6.3.2.4.8 What's next?

In the next module, you will learn how to use a JSlider object along with Affine Transforms to control the rotation of an image.

6.3.2.4.9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Controlling an Image-Scaling Program with a Slider
- File: Java3118.htm
- Published: 05/13/12
- Revised: 09/07/12

[56] http://www.dickbaldwin.com/java/Java358.htm

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.2.4.10 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 3 (p. 1556) below.

Listing 6.82: Complete program listing.

```
    /*File Prob09 Copyright 2008 R.G.Baldwin
Revised 09/13/10

*******************************************************/
import java.awt.geom.AffineTransform;
import java.awt.Graphics2D;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class Prob09{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob09Runner();
  }//end main method
}//end class Prob09
//End program specifications.
/////////////////////////////////////////////////////
/////////////////////////////////////////////////////

/*-----------------------------------------------//
```

```
**************************************************/
class Prob09Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();
  private JSlider slider = new JSlider(0,100,0);

  private Picture beach = new Picture("Prob09a.jpg");
  private Picture butterfly = new Picture("Prob09b.jpg");

  private int beachWidth = beach.getWidth();
  private int beachHeight = beach.getHeight();

  private Picture display =
            new Picture(beachWidth,beachHeight);

  private Image image = null;
  private Graphics graphics = null;
  private AffineTransform transform = null;
  private Graphics2D g2 = null;

  public Prob09Runner(){//constructor

    System.out.println("Dick Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    beach.addMessage("Dick Baldwin",10,20);
    butterfly.addMessage("Dick Baldwin",10,20);

    slider.setMajorTickSpacing(10);
    slider.setMinorTickSpacing(5);
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);

    mainPanel.setLayout(new BorderLayout());
    titlePanel.add(new JLabel(
                          "Percent Size of Beach Image"));
    mainPanel.add(titlePanel,BorderLayout.NORTH);
    mainPanel.add(slider,BorderLayout.CENTER);

    getContentPane().add(mainPanel);
    //pack();
    setSize(beachWidth + 7,97);
    setTitle("Dick Baldwin");
    setLocation(0,beachHeight + 25);
    setVisible(true);

    //Draw and display the background image of the
    // butterfly.
    graphics = display.getGraphics();
```

```
    graphics.drawImage(butterfly.getImage(),0,0,null);

    display.show();
    //-------------------------------------------------//
    //Register an anonymous listener object on the slider.
    //Each time the slider fires a ChangeEvent, this event
    // handler restores the background image of the
    // butterfly. Then it draws a scaled version of the
    // beach on top of the background image using the
    // slider value, which ranges from 0 to 100 as the
    // scale factor as a percent of 1.0. The image of the
    // beach is always aligned with the center
    // of the contentPane of the JFrame.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){

          //Restore the background image of the butterfly.
          graphics = display.getGraphics();
          graphics.drawImage(
                         butterfly.getImage(),0,0,null);

          drawScaledPictureOnPicture(beach,
                                     display,
                                     slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------------//
  }//end constructor
  //-------------------------------------------------//

  //Scales and draws the source picture onto the center
  // of the destination picture.
  private void drawScaledPictureOnPicture(
                                      Picture source,
                                      Picture dest,
                                      double scaleFactor){

    transform = new AffineTransform();

    double translateX = dest.getWidth()/2
                    - source.getWidth()*scaleFactor/100/2;
    double translateY = dest.getHeight()/2
                  - source.getHeight()*scaleFactor/100/2;;
    transform.translate(translateX,translateY);
    transform.scale(scaleFactor/100.0,scaleFactor/100.0);

    //Get the Graphics2D object used to draw on the
    // destination picture.
```

```
    g2 = (Graphics2D)dest.getGraphics();

    //Scale and draw the source image on the destination
    // image.
    g2.drawImage(source.getImage(),transform,null);

  }//end drawScaledPictureOnPicture method
}//end class Prob09Runner
```

-end-

6.3.2.5 Java OOP: Controlling Image Rotation with a Slider and Affine Transforms[57]

6.3.2.5.1 Table of Contents

6.3.2.5.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [58] .

6.3.2.5.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.2.5.2.1.1 Figures

[57] This content is available online at <http://cnx.org/content/m44915/1.3/>.
[58] http://cnx.org/content/m44148/latest/

6.3.2.5.2.1.2 Listings

6.3.2.5.3 Preview

In this lecture, I will explain a program that uses **Affine Transforms** to rotate an image by a specified angle around a specified anchor point.

Then the program translates the image so as to center it in a **JFrame** object.

Specification of the rotation angle

A **JSlider** is used to specify the rotation angle.

The range of the slider is from 0 to 360 degrees.

The position of the thumb on the slider specifies a counter-clockwise rotation angle in degrees.

Program output at startup

The thumb on the slider is at 45 degrees when the program starts running.

This causes the initial rotation angle of the butterfly image to be 45 degrees counter-clockwise as shown in Figure 1 (p. 1561) .

Program output at startup.

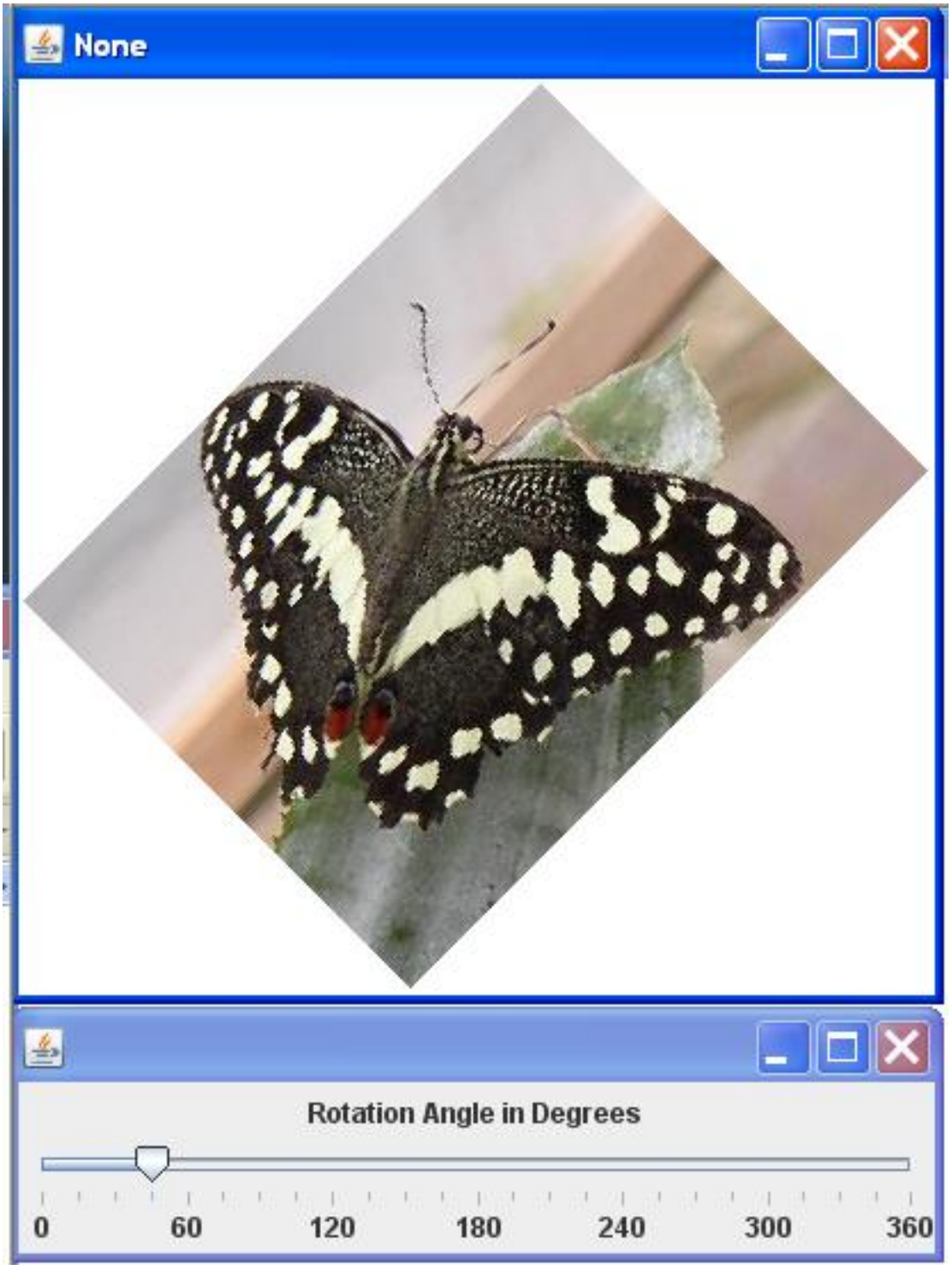


Figure 6.28: Program output at startup.

Corners barely graze inner-edges of the frame's borders

Note that for the initial rotation angle shown in Figure 1 (p. 1561) , the corners of the image almost touch the inner-edges of the borders on the frame.

If you run the program and rotate the image, you will see that the size of the frame is such that the corners of the image barely graze the inner-edges for those cases where the diagonals of the image are horizontal and vertical.

Program output for slider at zero degrees

Figure 2 (p. 1563) shows the program output when the thumb on the slider has been moved to zero degrees at the far-left end of the slider.

Program output for slider at zero.

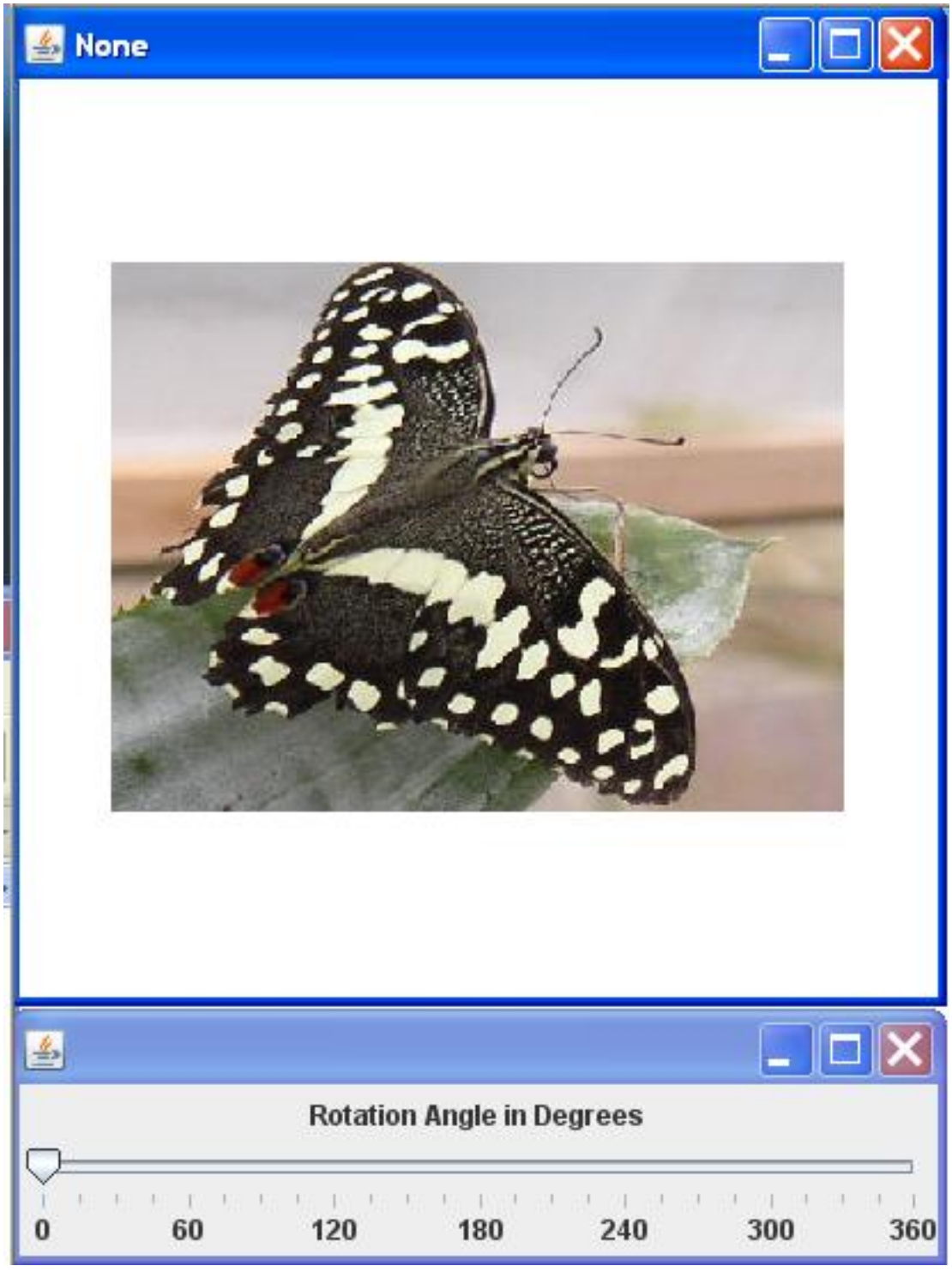


Figure 6.29: Program output for slider at zero.

Program output for slider at 240 degrees

Figure 3 (p. 1565) shows the result of positioning the thumb on the slider at 240 degrees.

Program output for slider at 240.

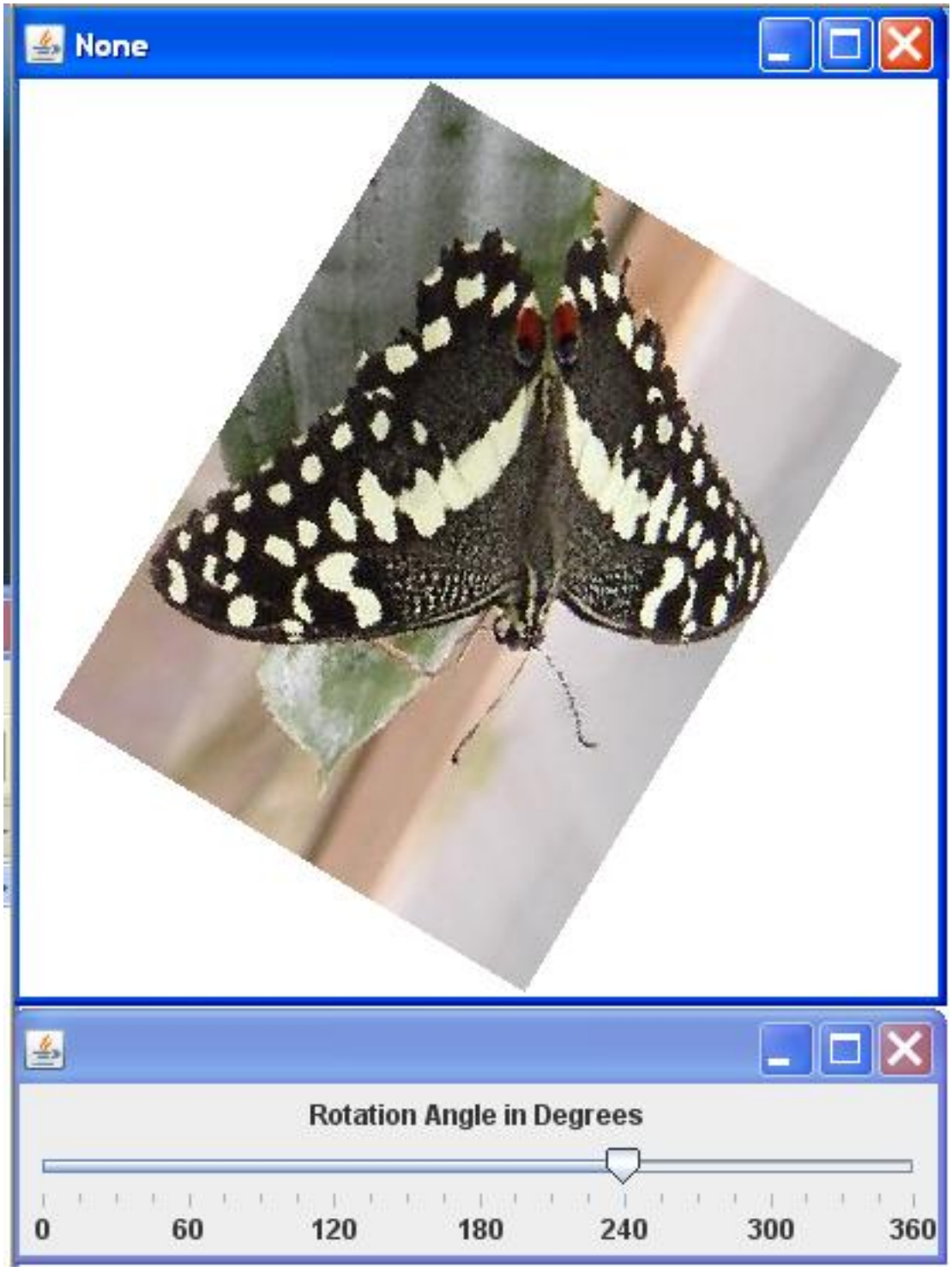


Figure 6.30: Program output for slider at 240.

Brief program specifications

Write a program named **Prob10** that uses the **Prob10** class definition shown in Listing 10 (p. 1571) and Ericson's media library along with the image file named Prob10.jpg [59] to produce the graphic output images shown in Figure 1 (p. 1561) through Figure 3 (p. 1565) .

The butterfly image rotates smoothly around its center as you move the slider back and forth.

The program must terminate and return control to the operating system when you click the large X in the upper-right corner of the GUI containing the slider.

General background information

The overall structure of this program is very similar to programs that I explained in earlier lectures titled

- Controlling an Edge Detector with a Slider [60] , and
- Controlling an Image-Scaling Program with a Slider [61] .

The most significant difference...

Is the code that is executed when the slider fires a **ChangeEvent** . There are a few other differences as well.

New and different code

I will explain the code that is new and different in this program.

I will refer you back to the earlier lectures for an explanation of the remainder of the code.

You can view the entire program in Listing 10 (p. 1571) .

6.3.2.5.4 Discussion and sample code

Beginning of the class named Prob10Runner

The code in the **main** method in Listing 10 (p. 1571) instantiates a new object of the class named **Prob10Runner** .

Listing 1 (p. 1566) shows the beginning of the class named **Prob10Runner** .

Listing 1 (p. 1566) declares several instance variables and initializes some of them.

I will refer back to some of these variables in later paragraphs.

Listing 6.83: Beginning of the class named Prob10Runner.

```
class Prob10Runner extends JFrame{

private JPanel mainPanel = new JPanel();
private JPanel titlePanel = new JPanel();

//Instantiate a new slider setting the limits and the
// initial position of the thumb.
private JSlider slider = new JSlider(0,360,45);

private Picture butterfly = new Picture("Prob10.jpg");
private Picture background = null;

private int butterflyWidth = butterfly.getWidth();
private int butterflyHeight = butterfly.getHeight();

private Picture display = null;
private int displayWidth = 0;
```

[59] http://cnx.org/content/m44915/latest/Prob10.jpg
[60] http://cnx.org/content/m44913/latest
[61] http://cnx.org/content/m44914/latest

```
private int displayHeight = 0;

private Image image = null;
private Graphics graphics = null;
private AffineTransform rotateTransform = null;
private AffineTransform translateTransform = null;
private Graphics2D g2 = null;
```

Beginning of the constructor for the Prob10Runner class

Listing 2 (p. 1567) shows the beginning of the constructor for the class named **Prob10Runner** .

Listing 6.84: Beginning of the constructor for the Prob10Runner class.

```
  public Prob10Runner(){//constructor

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Compute the minimum dimensions allowed for the
// display window that will contain the butterfly
// image rotated at any angle. This turns out to
// be a square with the length of each side equal
// to the diagonal length of the butterfly picture.
//The length of each side was increased by one
// pixel to guard against loss of precision when
// converting from double to int.
int diagonal = 1 + (int)(Math.sqrt(
                    butterflyWidth*butterflyWidth +
                    butterflyHeight*butterflyHeight));

//Instantiate the picture in which the rotated
// butterfly image will be displayed.
display = new Picture(diagonal,diagonal);
displayWidth = displayHeight = diagonal;

//This picture provides a white background the same
// size as the display picture.
background = new Picture(diagonal,diagonal);
```

Not much that is new here

The only thing that might be considered new in Listing 2 (p. 1566) is the code that computes the minimum dimensions for a display window that will contain the butterfly image rotated by any angle.

The rationale for this computation is explained in the comments.

Construct the GUI for the slider

The code in Listing 3 (p. 1567) constructs the GUI that contains the slider shown in Figure 1 (p. 1561) .

Although the code in Listing 3 (p. 1567) is a little tedious, there are no new concepts associated with that code.

Listing 6.85: Construct the GUI for the slider.

```
    //Construct the GUI for the slider.
slider.setMajorTickSpacing(60);
slider.setMinorTickSpacing(15);
```

```
slider.setPaintTicks(true);
slider.setPaintLabels(true);

mainPanel.setLayout(new BorderLayout());
titlePanel.add(new JLabel(
                        "Rotation Angle in Degrees"));
mainPanel.add(titlePanel,BorderLayout.NORTH);
mainPanel.add(slider,BorderLayout.CENTER);

getContentPane().add(mainPanel);
pack();//Adjust the size of the slider GUI.

//Compute an improved size and location for the
// GUI containing the slider.
setSize(displayWidth + 2 * getInsets().left,
        mainPanel.getHeight() + slider.getHeight());
setLocation(0,displayHeight + getInsets().top
                            + getInsets().bottom + 1);
setVisible(true);
```

Rotate and display the butterfly

The code in Listing 4 (p. 1568) calls the method named **rotatePicture** , passing the initial value of the thumb on the slider as a parameter to cause the initial display of the butterfly to be properly rotated.

Then Listing 4 (p. 1568) calls the **show** method on the display to cause the rotated butterfly image to be displayed as shown in Figure 1 (p. 1561) .

Listing 6.86: Rotate and display the butterfly.

```
    rotatePicture(slider.getValue());

display.show();
```

Register a ChangeListener on the slider

The code in Listing 5 (p. 1568) registers an anonymous **ChangeListener** object on the slider.

Listing 6.87: Register a ChangeListener on the slider.

```
    slider.addChangeListener(
  new ChangeListener(){
    public void stateChanged(ChangeEvent e){
      //Restore the background image of the display
      // to all white.
      graphics = display.getGraphics();
      graphics.drawImage(
                     background.getImage(),0,0,null);

      //Rotate the butterfly image, draw it on the
      // display, and repaint the display on the
      // screen..
      rotatePicture(slider.getValue());
      display.repaint();
    }//end stateChanged
```

```
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------------//
  }//end constructor
```

Each time the slider fires a **ChangeEvent** , the **stateChanged** method in Listing 5 (p. 1568) is executed.

The **stateChanged** method begins by restoring the background image of thedisplay.

Then the **stateChanged** method calls the **rotatePicture** method to draw a rotated version of thebutterfly on top of the background image using theslider value, *(which ranges from 0 to +360)* , asthe rotation angle in degrees.

The image of thebutterfly *(see Figure 1* (p. 1561)) is always centered in the displaypicture.

Finally, the **stateChanged** method causes the display to be repainted to force the rotated image to appear on the screen.

End of the constructor

Listing 5 (p. 1568) signals the end of the constructor for the class named **Prob10Runner** .

Beginning of the rotatePicture method

The **rotatePicture** method begins in Listing 6 (p. 1569) .

Listing 6.88: Beginning of the rotatePicture method.

```
  private void rotatePicture(double angle){

//Set up the rotation transform
rotateTransform = new AffineTransform();
//Negate the angle for counter-clockwise rotation.
rotateTransform.rotate(-Math.toRadians(angle),
                       butterflyWidth/2,
                       butterflyHeight/2);
```

The **rotatePicture** method accepts a rotation angle in degrees as an incoming parameter.

The **rotatePicture** method

- rotates a butterfly image by that angle around its center,
- translates the rotated image, and
- draws the rotated image in the center of a display picture.

Set up the rotation transform

Listing 6 (p. 1569) begins by instantiating a new object of the **AffineTransform** class.

Then it calls the **rotate** method on the transform object to configure that object for use in rotating the image of the butterfly.

Overloaded versions of the *rotate* method

There are several overloaded versions of the **rotate** method.

The version called in Listing 6 (p. 1569) requires three parameters:

- The rotation angle in radians
- The X coordinate of the anchor point *(point around which the rotation will take place)* .
- The Y coordinate of the anchor point.

A positive rotation value indicates a clockwise rotation.

Convert from degrees to radians

Listing 6 (p. 1569) converts the incoming angle in degrees to radians

The code in Listing 6 (p. 1569) applies a minus sign to convert the angle from a positive value to a negative value for ***counter-clockwise*** rotation, and passes that value as the rotation parameter.

Specify the rotation *anchor point*

Listing 6 (p. 1569) also specifies the center of the butterfly image as the anchor point.

Set up the translation transform

Listing 7 (p. 1570) sets up the translation transform that will be used to translate the rotated image to the center of the new **Picture** object.

Listing 6.89: Set up the translation transform.

```
    translateTransform = new AffineTransform();
translateTransform.translate(
                (displayWidth - butterflyWidth)/2,
                (displayHeight - butterflyHeight)/2);
```

Listing 7 (p. 1570) instantiates a new **AffineTransform** object.

Then it calls the **translate** method to configure that object for use in translating *(moving)* the rotated image of the butterfly.

The translation transform object will be concatenated with the rotation object from Listing 6 (p. 1569) to produce an **AffineTransform** object that can ***rotate*** and ***translate*** , in that order.

Concatenate the transforms

Listing 8 (p. 1570) concatenates the **AffineTransform** object referred to by **translateTransform** with the transform object referred to by **rotateTransform** .

Listing 6.90: Concatenate the transforms.

```
    translateTransform.concatenate(rotateTransform);
```

The resulting composite transform

The concatenation in Listing 8 (p. 1570) results in an **Affine Transform** object referred to by **translateTransform** that will first rotate the image around its center and then translate the rotated image to the center of the new **Picture** object.

The order of operations is very important

When rotating and translating images, it is important that the two operations be performed in the correct order.

Otherwise, the results might not be what you want.

Transform and draw the butterfly

Listing 9 (p. 1570) applies the composite transform to the butterfly image and draws the transformed image on the output picture as shown in Figure 1 (p. 1561) , Figure 2 (p. 1563) , and Figure 3 (p. 1565) .

Listing 6.91: Transform and draw the butterfly image.

```
    Graphics2D g2 = (Graphics2D)display.getGraphics();
g2.drawImage(butterfly.getImage(),
             translateTransform,
             null);

  }//end rotatePicture

}//end class Prob10Runner
```

The end of the program

Listing 9 (p. 1570) also signals the end of the **rotatePicture** method, the end of the **Prob10Runner** class, and the end of the program.

6.3.2.5.5 Run the program

I encourage you to copy the code from Listing 10 (p. 1571) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.2.5.6 Summary

In this lecture, you learned how to use a **JSlider** object along with **Affine Transforms** to control the rotation of an image.

6.3.2.5.7 What's next?

In the next module, you will learn how to open an image file *(specified by a string in a text field)* in a **PictureExplorer** object.

6.3.2.5.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Controlling Image Rotation with a Slider and Affine Transforms
- File: Java3120.htm
- Published: 09/07/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.2.5.9 Complete program listing

A complete listing of the program discussed in this lecture is shown in Listing 10 (p. 1571) below.

Listing 6.92: Complete program listing.

```
    /*File Prob10 Copyright 2008 R.G.Baldwin
Revised 09/14/10

*******************************************************/
import java.awt.geom.AffineTransform;
import java.awt.Graphics2D;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class Prob10{
  public static void main(String[] args){
    new Prob10Runner();
  }//end main method
}//end class Prob10
/*******************************************************/

class Prob10Runner extends JFrame{

  private JPanel mainPanel = new JPanel();
  private JPanel titlePanel = new JPanel();

  //Instantiate a new slider setting the limits and the
  // initial position of the thumb.
  private JSlider slider = new JSlider(0,360,45);

  private Picture butterfly = new Picture("Prob10.jpg");
  private Picture background = null;

  private int butterflyWidth = butterfly.getWidth();
  private int butterflyHeight = butterfly.getHeight();

  private Picture display = null;
  private int displayWidth = 0;
  private int displayHeight = 0;

  private Image image = null;
  private Graphics graphics = null;
  private AffineTransform rotateTransform = null;
  private AffineTransform translateTransform = null;
  private Graphics2D g2 = null;
```

```
public Prob10Runner(){//constructor

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  //Compute the minimum dimensions allowed for the
  // display window that will contain the butterfly
  // image rotated at any angle. This turns out to
  // be a square with the length of each side equal
  // to the diagonal length of the butterfly picture.
  //The length of each side was increased by one
  // pixel to guard against loss of precision when
  // converting from double to int.
  int diagonal = 1 + (int)(Math.sqrt(
                      butterflyWidth*butterflyWidth +
                      butterflyHeight*butterflyHeight));

  //Instantiate the picture in which the rotated
  // butterfly image will be displayed.
  display = new Picture(diagonal,diagonal);
  displayWidth = displayHeight = diagonal;

  //This picture provides a white background the same
  // size as the display picture.
  background = new Picture(diagonal,diagonal);

  //Construct the GUI for the slider.
  slider.setMajorTickSpacing(60);
  slider.setMinorTickSpacing(15);
  slider.setPaintTicks(true);
  slider.setPaintLabels(true);

  mainPanel.setLayout(new BorderLayout());
  titlePanel.add(new JLabel(
                          "Rotation Angle in Degrees"));
  mainPanel.add(titlePanel,BorderLayout.NORTH);
  mainPanel.add(slider,BorderLayout.CENTER);

  getContentPane().add(mainPanel);
  pack();//Adjust the size of the slider GUI.

  //Compute an improved size and location for the
  // GUI containing the slider.
  setSize(displayWidth + 2 * getInsets().left,
          mainPanel.getHeight() + slider.getHeight());
  setLocation(0,displayHeight + getInsets().top
                              + getInsets().bottom + 1);
  setVisible(true);

  //Place the butterfly image in the display picture
  // with a rotation angle specified by the initial
  // position of the thumb on the slider.
```

```
    rotatePicture(slider.getValue());

    display.show();
    //-------------------------------------------------//
    //Register an anonymous listener object on the slider.
    //Each time the slider fires a ChangeEvent, this event
    // handler restores the background image of the
    // display. Then it draws a rotated version of the
    // butterfly on top of the background image using the
    // slider value, which ranges from 0 to +360, as
    // the rotation angle in degrees. The image of the
    // butterfly is always centered in the display
    // picture.
    slider.addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          //Restore the background image of the display
          // to all white.
          graphics = display.getGraphics();
          graphics.drawImage(
                         background.getImage(),0,0,null);
          //Rotate the butterfly image, draw it on the
          // display, and repaint the display on the
          // screen..
          rotatePicture(slider.getValue());
          display.repaint();
        }//end stateChanged
      }//end new ChangeListener
    );//end addChangeListener
    //-------------------------------------------------//
  }//end constructor
  //-------------------------------------------------//

  //This method accepts a rotation angle in degrees. It
  // rotates a butterfly image by that angle around its
  // center, translates, and draws the rotated image in
  // the center of a display picture.
  private void rotatePicture(double angle){

    //Set up the rotation transform
    rotateTransform = new AffineTransform();
    //Negate the angle for counter-clockwise rotation.
    rotateTransform.rotate(-Math.toRadians(angle),
                           butterflyWidth/2,
                           butterflyHeight/2);

    //Set up the translation transform that will translate
    // the rotated image to the center of the new Picture
    // object.
    translateTransform = new AffineTransform();
```

```
    translateTransform.translate(
                    (displayWidth - butterflyWidth)/2,
                    (displayHeight - butterflyHeight)/2);

    //Concatenate the two transforms so that the image
    // will first be rotated around its center and then
    // translated to the center of the new Picture object.
    translateTransform.concatenate(rotateTransform);

    //Get the graphics context of the display picture,
    // apply the transform to the butterfly image, and
    // draw the transformed picture on the display
    // picture.
    Graphics2D g2 = (Graphics2D)display.getGraphics();
    g2.drawImage(butterfly.getImage(),
                 translateTransform,
                 null);

  }//end rotatePicture

}//end class Prob10Runner
```

-end-

6.3.3 Part 3

6.3.3.1 Java OOP: Opening an Image File in a PictureExplorer Object[62]

6.3.3.1.1 Table of Contents

6.3.3.1.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [63] .

[62] This content is available online at <http://cnx.org/content/m44916/1.2/>.
[63] http://cnx.org/content/m44148/latest/

6.3.3.1.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.3.1.2.1.1 Figures

- Figure 1 (p. 1576) . Program output at startup.
- Figure 2 (p. 1577) . Screen shot after the image is loaded.

6.3.3.1.2.1.2 Listings

- Listing 1 (p. 1578) . Complete program listing.
- Listing 2 (p. 1579) . The driver class.
- Listing 3 (p. 1580) . Beginning of the Prob11Runner class.
- Listing 4 (p. 1580) . Beginning of the constructor.
- Listing 5 (p. 1581) . Beginning of anonymous listener class.
- Listing 6 (p. 1581) . Completion of the anonymous listener class.

6.3.3.1.3 Preview

This program demonstrates how to specify an image file in a text field, and open the image in a new **PictureExplorer** object.

Program output at startup

Figure 1 (p. 1576) shows the program output at startup.

Program output at startup.

Figure 6.31: Program output at startup.

Pre-loaded image file name

For convenience, the text field is pre-loaded with the name of an image file that is located in the current directory.

Press Enter to load the image

Pressing the Enter key while the text field has the focus will cause the image to be loaded into a **PictureExplorer** object, and will cause the **PictureExplorer** object to be displayed as shown in Figure 2 (p. 1577) .

Screen shot after the image is loaded.

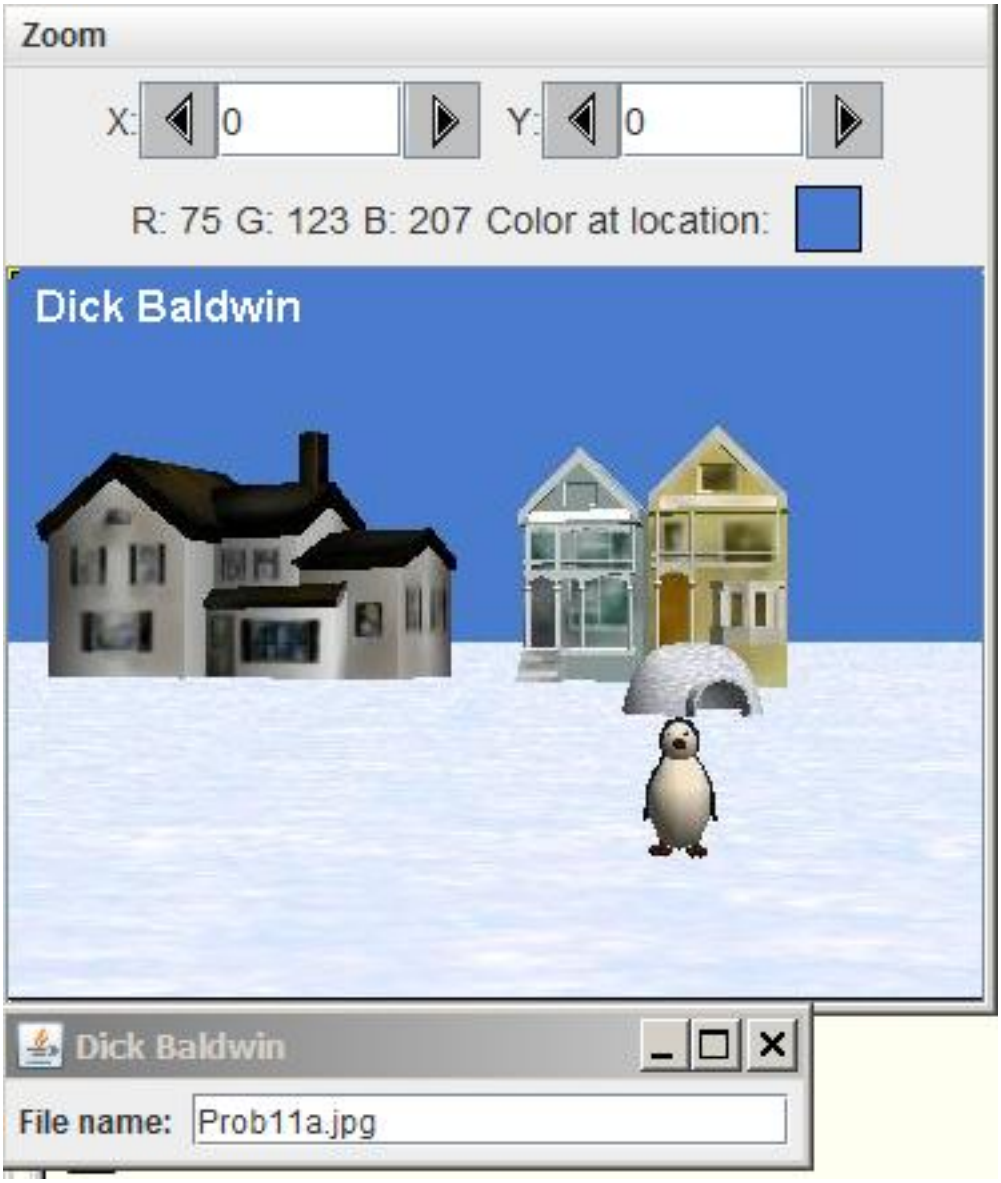


Figure 6.32: Screen shot after the image is loaded.

Control panel is relocated

Note in Figure 2 (p. 1577) that the control panel containing the text field has been automatically relocated to a position immediately below the **PictureExplorer** object.

Another image file in the current directory

Entering the path and name of any jpeg file will cause that file to be loaded into a new **PictureExplorer** object.

6.3.3.1.4 Complete program listing

A complete listing of the program discussed in this lecture is shown in Listing 1 (p. 1578) below.

Listing 6.93: Complete program listing.

```
    /*File Prob11 Copyright 2012 R.G.Baldwin
The purpose of this program is to demonstrate the use of
a JTextField object to specify the name of an image file,
which is then loaded and displayed in a PictureExplorer
object.

The text field is pre-loaded with the name of an image
file that is in the current directory (Prob11a.jpg).

An image file named Prob11b.jpg is also in the current
directory and can be loaded and displayed by entering
the name in the text field.

Any image file on the disk can be loaded and displayed
by entering the path and name of the image file.
**********************************************************/

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.WindowConstants;

public class Prob11{
  public static void main(String[] args){
    new Prob11Runner();
  }//end main method
}//end class Prob11
//========================================================//

class Prob11Runner extends JFrame{
  private Prob11Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel fileNamePanel = new JPanel();
  private JTextField inputFileNameField =
                        new JTextField("Prob11a.jpg",20);

  private String fileName = "no file specified";
  //--------------------------------------------------//

  public Prob11Runner(){//constructor
    fileNamePanel.add(new JLabel("File name: "));
```

```
    fileNamePanel.add(inputFileNameField);

    //Add the fileNamePanel to the content pane, adjust
    // to the correct size, and set the title.
    getContentPane().add(fileNamePanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible, set the focus, and establish
    // a reference to the GUI object.
    setVisible(true);
    inputFileNameField.requestFocus();
    jFrameObj = this;

    //-----------------------------------------------//
    //Register listeners on the user input field.
    //-----------------------------------------------//

    inputFileNameField.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          fileName = inputFileNameField.getText();

          pix = new Picture(fileName);
          pix.addMessage("Dick Baldwin",10,20);
          explorer = new PictureExplorer(pix);

          //Set the location for the control GUI
          // immediately below the PictureExplorer object,
          // and set its default close operation.
          setLocation(0,pix.getHeight() + 128);
          jFrameObj.setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
        }//end action performed
      }//end newActionListener
    );//end addActionListener
    //-----------------------------------------------//
  }//end constructor
  //-------------------------------------------------//
}//end class Prob11Runner
```

6.3.3.1.5 Will explain the code in fragments

As usual, I will break the code down, and explain it in fragments.

The driver class

The driver class is shown in Listing 2 (p. 1579) .

There is nothing new in Listing 2 (p. 1579) .

Listing 6.94: The driver class.

```
    public class Prob11{
  public static void main(String[] args){
    new Prob11Runner();
  }//end main method
}//end class Prob11
```

Beginning of the Prob11Runner class

The **Prob11Runner** class begins in Listing 3 (p. 1580) .

Listing 3 (p. 1580) declares several instance variables that will be used later in the program, initializing some of them.

Note that the class extends **JFrame** , so an object of the class *"is a"* **JFrame** object.

Listing 6.95: Beginning of the Prob11Runner class.

```
    class Prob11Runner extends JFrame{
  private Prob11Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel fileNamePanel = new JPanel();
  private JTextField inputFileNameField =
              new JTextField("Prob11a.jpg",20);

  private String fileName = "no file specified";
```

Beginning of the constructor

The constructor for the **Prob11Runner** class begins in Listing 4 (p. 1580) .

The code in Listing 4 (p. 1580) performs the physical construction of the GUI show in Figure 1 (p. 1576) .

The code in Listing 4 (p. 1580) saves a reference to the object so that it can be accessed later from within a listener object.

Listing 6.96: Beginning of the constructor.

```
      public Prob11Runner(){//constructor
    fileNamePanel.add(new JLabel("File name: "));
    fileNamePanel.add(inputFileNameField);

    //Add the fileNamePanel to the content pane, adjust
    // to the correct size, and set the title.
    getContentPane().add(fileNamePanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible, set the focus, and establish
    // a reference to the GUI object.
    setVisible(true);
    inputFileNameField.requestFocus();
    jFrameObj = this;
```

Register a listener on the text field

Listing 5 (p. 1581) shows the beginning of code that instantiates an **ActionListener** object of an anonymous listener class, and registers the listener object on the text field shown in Figure 1 (p. 1576) .

Listing 6.97: Beginning of anonymous listener class.

```
    inputFileNameField.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      fileName = inputFileNameField.getText();

      pix = new Picture(fileName);
      pix.addMessage("Dick Baldwin",10,20);

      explorer = new PictureExplorer(pix);
```

Get image and create Picture object

Listing 5 (p. 1581) begins by getting the name of the image file from the text field, using that image file to create a new **Picture** object, and adding a name in the upper-left corner of the picture.

Create new PictureExplorer object

Then Listing 5 (p. 1581) instantiates a new **PictureExplorer** object that encapsulates the **Picture** object that was created from the image file.

Completion of the anonymous listener class

Listing 6 (p. 1581) shows the completion of the anonymous listener class, as well as the end of the constructor, and the end of the class named **Prob11Runner** .

Listing 6.98: Completion of the anonymous listener class.

```
          setLocation(0,pix.getHeight() + 128);

        jFrameObj.setDefaultCloseOperation(
                        WindowConstants.EXIT_ON_CLOSE);
      }//end action performed
    }//end newActionListener
  );//end addActionListener
  //-----------------------------------------------------//
 }//end constructor
 //-------------------------------------------------------//
}//end class Prob11Runner
```

Relocate the GUI

Listing 6 (p. 1581) begins by setting the location of the GUI to a location that is immediately below the **PictureExplorer** object.

Relocate the GUI (cont'd)

Note that the value of 128 pixels was experimentally determined to be the approximate difference between the height of the **PictureExplorer** object and the height of the **Picture** object that it encapsulates.

(This difference may not be correct for different display options on different operating systems.)

Set the default close operation

Listing 6 (p. 1581) ends by setting the behavior of the X-button in the GUI to a value that will cause it to terminate the program when the button is clicked.

6.3.3.1.6 Run the program

I encourage you to copy the code from Listing 1 (p. 1578) and download the image file named Prob11a.jpg [64] . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.3.1.7 What's next?

In the next module, you will learn how to extract color data from a selected pixel in a **PictureExplorer** object, and how to display a color value in a text field. You will also learn how to disable the X-button in the **PictureExplorer** object, and how to use a reference to the **JFrame** object that serves as a container for the **PictureExplorer** object.

6.3.3.1.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Opening an Image File in a PictureExplorer Object
- File: Java3122.htm
- Published: 09/08/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.3.3.2 Java OOP: Extracting pixel color data from a PictureExplorer object[65]

6.3.3.2.1 Table of Contents

[64] http://cnx.org/content/m44916/latest/Prob11a.jpg
[65] This content is available online at <http://cnx.org/content/m44917/1.2/>.

6.3.3.2.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [66] .

6.3.3.2.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.3.2.2.1.1 Figures

6.3.3.2.2.1.2 Listings

6.3.3.2.3 Preview

In this module, I will show you how to extract color data from a selected pixel in a **PictureExplorer** object, and how to display the value in a text field.

I will also show you how to disable the **X-button** in the **PictureExplorer** object, and how to get and use a reference to the **JFrame** object that serves as a container for the **PictureExplorer** object.

Program output at startup

Figure 1 (p. 1584) shows a screen shot of the program output at startup.

Note that the text field in the red panel near the bottom contains all zeros.

[66] http://cnx.org/content/m44148/latest/

Program output at startup.

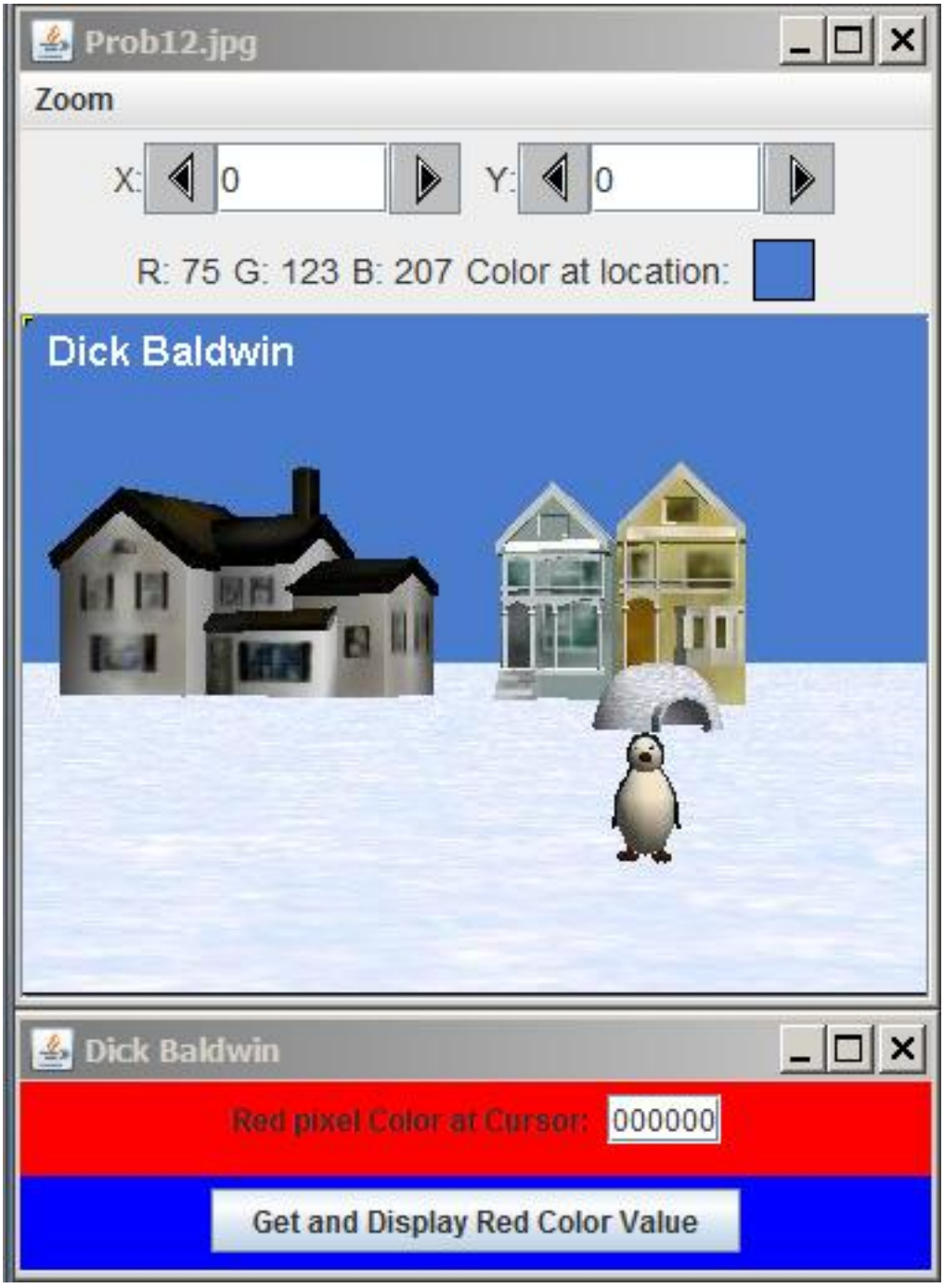


Figure 6.33: Program output at startup.

Program output after clicking the button

Figure 2 (p. 1586) shows the screen output after moving the **PictureExplorer** crosshair cursor to the brown door immediately above the penguin, and clicking the button labeled *"Get and Display Red Color Value."*

Program output after clicking the button.

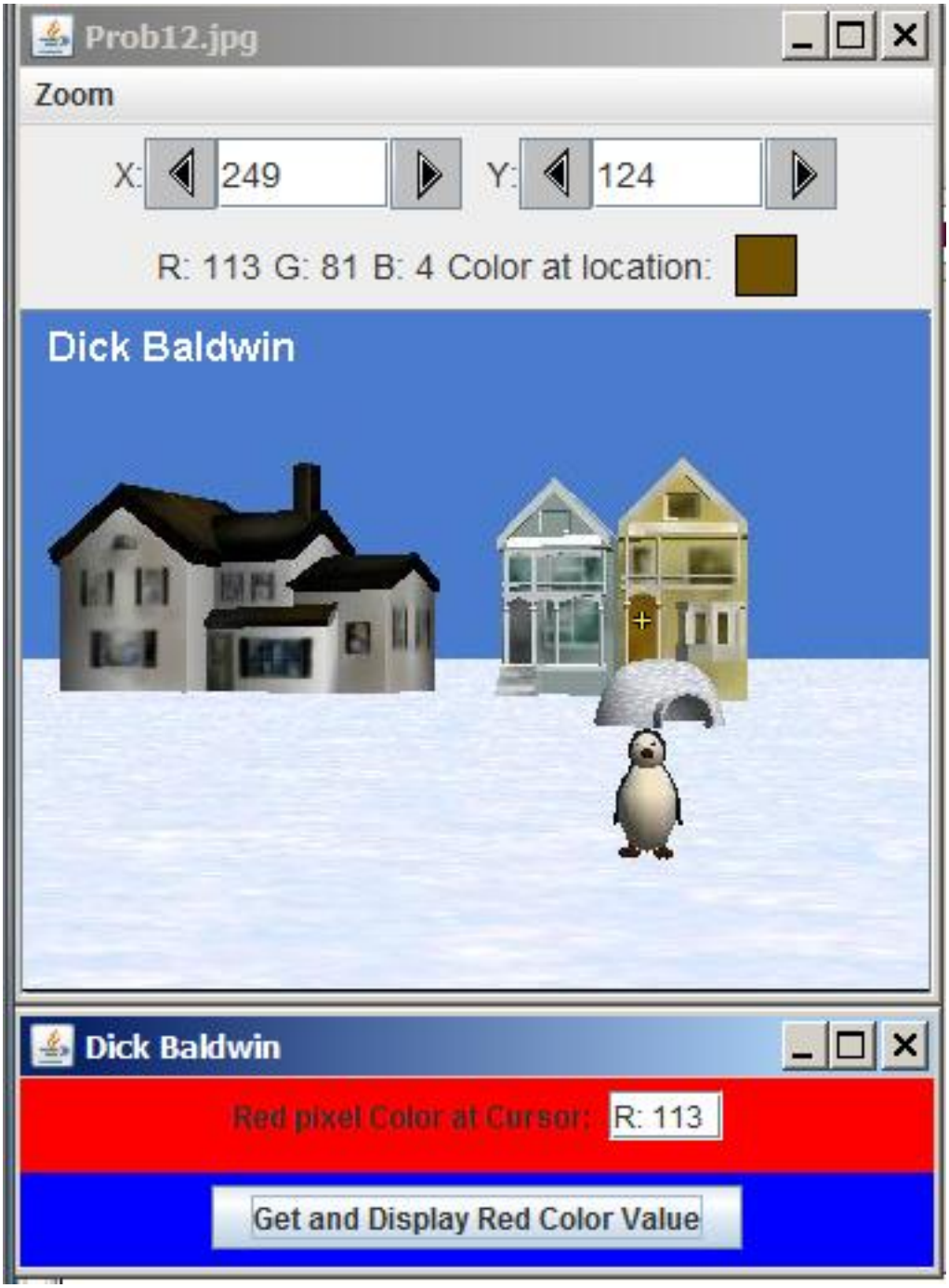


Figure 6.34: Program output after clicking the button.

The red color value

The text field near the bottom of Figure 2 (p. 1586) displays the red color value of the pixel at the location of the crosshair cursor on the door in the image.

Compare this value with the value of the red color value displayed near the top of the **PictureExplorer** object. You will see that they match.

6.3.3.2.4 Discussion and sample code

Two source code files are required for this program:

- The file named **Prob12.java**
- A modified version of Ericson's file named **PictureExplorer.java**

A complete listing of **Prob12.java** is shown in Listing 10 (p. 1591) .

A complete listing of the modified **PictureExplorer.java** file is shown in Listing 11 (p. 1594) .

Modifications to PictureExplorer.java

The modifications that I made can be found by searching the source code in Listing 11 (p. 1594) for the word "Baldwin".

The modifications were:

- I added a *getter* method to cause the red color value to be accessible from outside the **PictureExplorer** object.
- I added a *getter* method that returns a reference to the **JFrame** object containing the **PictureExplorer** object.
- I disabled the call to the **setDefaultCloseOperation** method for the **PictureExplorer** object.

Addition of getter methods

The code for the each of the new *getter* methods is shown in Listing 1 (p. 1587) .

These two methods simply return values that already exist in the **PictureExplorer** object.

Listing 6.99: Addition of getter methods.

```
    //===Methods added by Baldwin on 05/15/12===============//
  //Method to get the red color value as text.
  public String getRValue(){
    return rValue.getText();
  }//end getRValue

  //Method to get a reference to the JFrame containing
  // the PictureExplorer object.
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()
//===End methods added by Baldwin on 05/15/12===========//
```

Disabled the call to setDefaultCloseOperation method

Comment indicators were used to disable the call to the **setDefaultCloseOperation** method as shown in Listing 2 (p. 1587) .

This code was replaced by code that you will see later that causes the **JFrame** to *do nothing* when the X-button on the **PictureExplorer** object is clicked.

Listing 6.100: Disabled the call to setDefaultCloseOperation method.

```
    /*Disabled by Baldwin on 5/15/12
   pictureFrame.setDefaultCloseOperation(
                                JFrame.DISPOSE_ON_CLOSE);
*/
```

No more modifications

That concludes the discussion of modifications to Ericson's **PictureExplorer** source code.

The driver class named Prob12

The driver class is shown in Listing 3 (p. 1588) .

There is nothing new here.

Listing 6.101: The driver class for Prob12.

```
    public class Prob12{
  public static void main(String[] args){
    new Prob12Runner();
  }//end main method
}//end class Prob12
```

Beginning of the class named Prob12Runner

The class named **Prob12Runner** begins in Listing 4 (p. 1588) .

The code in Listing 4 (p. 1588) declares several instance variables and initializes some of them.

Listing 6.102: Beginning of the class named Prob12Runner.

```
    class Prob12Runner extends JFrame{
  private JFrame explorerFrame = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();

  private JTextField redField = new JTextField("000000");

  private JButton getDataButton = new JButton(
                       "Get and Display Red Color Value");
  private String fileName = "Prob12.jpg";
```

Beginning of the constructor

The constructor for the class named **Prob12Runner** begins in Listing 5 (p. 1588) .

The code in Listing 5 (p. 1588) performs the physical construction of the GUI control panel at the bottom of Figure 1 (p. 1584) .

The final position of the GUI control panel will be established after the size of the **PictureExplorer** object's **JFrame** is known.

Listing 6.103: Beginning of the constructor.

```
        controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);
```

```
    colorPanel.setBackground(Color.RED);
    colorPanel.add(new JLabel(
                         "Red pixel Color at Cursor: "));
    colorPanel.add(redField);

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(getDataButton);

    getContentPane().add(controlPanel);
    setTitle("Dick Baldwin");

    setVisible(true);
```

Construct a Picture object

Listing 6 (p. 1589) creates a new **Picture** object, and adds a name to the picture before using it to construct a **PictureExplorer** object.

Listing 6.104: Construct a Picture object.

```
        pix = new Picture(fileName);
    pix.addMessage("Dick Baldwin",10,20);
```

Construct a PictureExplorer object

Listing 7 (p. 1589) begins by constructing a new **PictureExplorer** object that encapsulates the **Picture** object instantiated in Listing 6 (p. 1589) , and by saving a reference to the **PictureExplorer** object in the instance variable named **explorer** .

Listing 6.105: Construct a PictureExplorer object.

```
        explorer = new PictureExplorer(pix);

    explorerFrame = explorer.getFrame();

    explorerFrame.setDefaultCloseOperation(
                    WindowConstants.DO_NOTHING_ON_CLOSE);
```

Get and use a reference to the JFrame object

Then Listing 7 (p. 1589) calls the **PictureExplorer** object's new **getFrame** method to get a reference to the **JFrame** object that contains the **PictureExplorer** .

This reference is used in Listing 7 (p. 1589) to call the **setDefaultCloseOperation** method to cause the **JFrame** to *do nothing* when the X-button on the **JFrame** is clicked.

The reference is also used in Listing 8 (p. 1589) to establish the size and location of the GUI control panel.

Set the size and location of the GUI control panel

Now that the **PictureExplorer** object exists, Listing 8 (p. 1589) sets the size and location of the GUI control panel as shown in Figure 1 (p. 1584) .

Listing 6.106: Set the size and location of the GUI control panel.

```
        //Set the size of the control GUI.
    setSize(explorerFrame.getWidth(),110);

    //Set the location for the control GUI
    setLocation(0,explorerFrame.getHeight());

    //Set the default close operation for the control GUI.
    setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
```

Note that it isn't necessary to guess about the actual height of the **JFrame** as was the case in an earlier module. The reference to the **JFrame** object provides that information.

Set the default close operation for the control GUI

Finally, Listing 8 (p. 1589) sets the default close operation on the control GUI to cause the program to terminate when the user clicks the X-button on the GUI.

Register a listener object on the button

Listing 9 (p. 1590) registers an **ActionListener** object on the button to cause it to call the **PictureExplorer** object's new **getRValue** method to get the red color value for the pixel at the current **PictureExplorer** crosshair cursor position, and to display that value in the text field as shown in Figure 2 (p. 1586) when the user clicks the button.

Listing 6.107: Register a listener object on the button.

```
      getDataButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){

        redField.setText(explorer.getRValue());

      }//end action performed
    }//end newActionListener
  );//end addActionListener
  //-----------------------------------------------//

 }//end constructor
}//end class Prob12Runner
```

End of the program

Listing 9 (p. 1590) also signals the end of the constructor and the end of the class named **Prob12Runner**.

6.3.3.2.5 Run the program

I encourage you to copy the code from Listing 10 (p. 1591) and Listing 11 (p. 1594) and download the image file named Prob12.jpg [67] . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

[67] http://cnx.org/content/m44917/latest/Prob12.jpg

6.3.3.2.6 Summary

You learned how to extract color data from a selected pixel in a **PictureExplorer** object, and to display the value in a text field.

You also learned how to disable the X-button in the **PictureExplorer** object, and how to use a reference to the **JFrame** object that serves as a container for the PictureExplorer object.

6.3.3.2.7 What's next?

In the next module, you will learn how to handle document events on text fields containing color values. You will also learn how to create a color swatch.

6.3.3.2.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Extracting pixel color data from a PictureExplorer object
- File: Java3124.htm
- Published: 09/08/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.3.2.9 Complete program listings

Complete listings of the source code discussed in this module are shown in Listing 10 (p. 1591) and Listing 11 (p. 1594) below.

Listing 6.108: Prob12.java.

```
    /*File Prob12 Copyright 2012 R.G.Baldwin
The purpose of this program is to demonstrate how to get
a color value from a PictureExplorer object and to display
it in JTextField object.

It also demonstrates how to access and use a reference
to a JFrame object that serves as the container for a
PictureExplorer object.
```

```
Finally, it demonstrates how to disable the X-button in
a PictureExplorer object.

Note that this program requires access to a modified
version of the PictureExplorier class.
**********************************************************/

import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.Color;


public class Prob12{
  public static void main(String[] args){
    new Prob12Runner();
  }//end main method
}//end class Prob12
//========================================================//

class Prob12Runner extends JFrame{
  private JFrame explorerFrame = null;
  private PictureExplorer explorer = null;
  private Picture pix;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();

  private JTextField redField = new JTextField("000000");

  private JButton getDataButton = new JButton(
                       "Get and Display Red Color Value");
  private String fileName = "Prob12.jpg";
  //--------------------------------------------------------//

  public Prob12Runner(){//constructor

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.RED);
    colorPanel.add(new JLabel(
```

```
                      "Red pixel Color at Cursor: "));
  colorPanel.add(redField);

  buttonPanel.setBackground(Color.BLUE);
  buttonPanel.add(getDataButton);

  getContentPane().add(controlPanel);
  setTitle("Dick Baldwin");

  setVisible(true);

  //Create a Picture object and add a name to
  // the picture before using it to construct the
  // PictureExplorer object.
  pix = new Picture(fileName);
  pix.addMessage("Dick Baldwin",10,20);

  //Create a PictureExplorer object containing the
  // picture and set its default close operation.
  explorer = new PictureExplorer(pix);
  //Get a ref to the JFrame in the PictureExplorer
  // object.
  explorerFrame = explorer.getFrame();
  explorerFrame.setDefaultCloseOperation(
                  WindowConstants.DO_NOTHING_ON_CLOSE);

  //Set the size of the control GUI.
  setSize(explorerFrame.getWidth(),110);

  //Set the location for the control GUI immediately
  // below the PictureExplorer object, and set its
  // default close operation.
  setLocation(0,explorerFrame.getHeight());
  setDefaultCloseOperation(
                       WindowConstants.EXIT_ON_CLOSE);

  //-----------------------------------------------//
  //Register a listener object on the button.
  //-----------------------------------------------//
  getDataButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){

        redField.setText(explorer.getRValue());

      }//end action performed
    }//end newActionListener
  );//end addActionListener
  //-----------------------------------------------//

}//end constructor
```

```
}//end class Prob12Runner
```

Listing 6.109: Modified PictureExplorer.java.

```
    import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;

/*
05/15/12
Modified by Baldwin to add getter methods to cause
the following values to be accessible from outside the
PictureExplorer object:

Red color value

Reference to the JFrame containing the PictureExplorer
object.

Also disabled the call to setDefaultCloseOperation
*/

/**
 * Displays a picture and lets you explore the picture by displaying the x, y, red,
 * green, and blue values of the pixel at the cursor when you click a mouse button or
 * press and hold a mouse button while moving the cursor.  It also lets you zoom in or
 * out.  You can also type in a x and y value to see the color at that location.
 *
 * Originally created for the Jython Environment for Students (JES).
 * Modified to work with DrJava by Barbara Ericson
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Keith McDermottt, gte047w@cc.gatech.edu
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class PictureExplorer implements MouseMotionListener, ActionListener, MouseListener
{

 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;

 //Main gui variables
 private JFrame pictureFrame;
 private JScrollPane scrollPane;

 //information bar variables
 private JLabel xLabel;
```

```
private JButton xPrevButton;
private JButton yPrevButton;
private JButton xNextButton;
private JButton yNextButton;
private JLabel yLabel;
private JTextField xValue;
private JTextField yValue;
private JLabel rValue;
private JLabel gValue;
private JLabel bValue;
private JLabel colorLabel;
private JPanel colorPanel;

// menu components
private JMenuBar menuBar;
private JMenu zoomMenu;
private JMenuItem twentyFive;
private JMenuItem fifty;
private JMenuItem seventyFive;
private JMenuItem hundred;
private JMenuItem hundredFifty;
private JMenuItem twoHundred;
private JMenuItem fiveHundred;

/** The picture being explored */
private DigitalPicture picture;

/** The image icon used to display the picture */
private ImageIcon scrollImageIcon;

/** The image display */
private ImageDisplay imageDisplay;

/** the zoom factor (amount to zoom) */
private double zoomFactor;

/** the number system to use, 0 means starting at 0, 1 means starting at 1 */
private int numberBase=0;

/**
 * Public constructor
 * @param picture the picture to explore
 */
public PictureExplorer(DigitalPicture picture)
{
  // set the fields
  this.picture=picture;
  zoomFactor=1;

  // create the window and set things up
  createWindow();
```

```
 }

//===Methods added by Baldwin on 05/15/12==============//
  //Method to get the red color value as text.
  public String getRValue(){
    return rValue.getText();
  }//end getRValue

  //Method to get a reference to the JFrame containing
  // the PictureExplorer object.
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()
//===End methods added by Baldwin on 05/15/12==========//




 /**
  * Changes the number system to start at one
  */
 public void changeToBaseOne()
 {
   numberBase=1;
 }

 /**
  * Set the title of the frame
  *@param title the title to use in the JFrame
  */
 public void setTitle(String title)
 {
   pictureFrame.setTitle(title);
 }

 /**
  * Method to create and initialize the picture frame
  */
 private void createAndInitPictureFrame()
 {
   pictureFrame = new JFrame(); // create the JFrame
   pictureFrame.setResizable(true);  // allow the user to resize it
   pictureFrame.getContentPane().setLayout(new BorderLayout()); // use border layout

   /*Disabled by Baldwin on 5/15/12
   pictureFrame.setDefaultCloseOperation(
                                JFrame.DISPOSE_ON_CLOSE);
   */
```

```
  pictureFrame.setTitle(picture.getTitle());
  PictureExplorerFocusTraversalPolicy newPolicy = new PictureExplorerFocusTraversalPolicy();
  pictureFrame.setFocusTraversalPolicy(newPolicy);

}

/**
 * Method to create the menu bar, menus, and menu items
 */
private void setUpMenuBar()
{
  //create menu
  menuBar = new JMenuBar();
  zoomMenu = new JMenu("Zoom");
  twentyFive = new JMenuItem("25%");
  fifty = new JMenuItem("50%");
  seventyFive = new JMenuItem("75%");
  hundred = new JMenuItem("100%");
  hundred.setEnabled(false);
  hundredFifty = new JMenuItem("150%");
  twoHundred = new JMenuItem("200%");
  fiveHundred = new JMenuItem("500%");

  // add the action listeners
  twentyFive.addActionListener(this);
  fifty.addActionListener(this);
  seventyFive.addActionListener(this);
  hundred.addActionListener(this);
  hundredFifty.addActionListener(this);
  twoHundred.addActionListener(this);
  fiveHundred.addActionListener(this);

  // add the menu items to the menus
  zoomMenu.add(twentyFive);
  zoomMenu.add(fifty);
  zoomMenu.add(seventyFive);
  zoomMenu.add(hundred);
  zoomMenu.add(hundredFifty);
  zoomMenu.add(twoHundred);
  zoomMenu.add(fiveHundred);
  menuBar.add(zoomMenu);

  // set the menu bar to this menu
  pictureFrame.setJMenuBar(menuBar);
}

/**
 * Create and initialize the scrolling image
 */
private void createAndInitScrollingImage()
{
```

```
  scrollPane = new JScrollPane();

  BufferedImage bimg = picture.getBufferedImage();
  imageDisplay = new ImageDisplay(bimg);
  imageDisplay.addMouseMotionListener(this);
  imageDisplay.addMouseListener(this);
  imageDisplay.setToolTipText("Click a mouse button on a pixel to see the pixel information");
  scrollPane.setViewportView(imageDisplay);
  pictureFrame.getContentPane().add(scrollPane, BorderLayout.CENTER);
}

/**
 * Creates the JFrame and sets everything up
 */
private void createWindow()
{
  // create the picture frame and initialize it
  createAndInitPictureFrame();

  // set up the menu bar
  setUpMenuBar();

  //create the information panel
  createInfoPanel();

  //creates the scrollpane for the picture
  createAndInitScrollingImage();

  // show the picture in the frame at the size it needs to be
  pictureFrame.pack();
  pictureFrame.setVisible(true);
}

/**
 * Method to set up the next and previous buttons for the
 * pixel location information
 */
private void setUpNextAndPreviousButtons()
{
  // create the image icons for the buttons
  Icon prevIcon = new ImageIcon(SoundExplorer.class.getResource("leftArrow.gif"),
                                "previous index");
  Icon nextIcon = new ImageIcon(SoundExplorer.class.getResource("rightArrow.gif"),
                                "next index");
  // create the arrow buttons
  xPrevButton = new JButton(prevIcon);
  xNextButton = new JButton(nextIcon);
  yPrevButton = new JButton(prevIcon);
  yNextButton = new JButton(nextIcon);

  // set the tool tip text
```

```
xNextButton.setToolTipText("Click to go to the next x value");
xPrevButton.setToolTipText("Click to go to the previous x value");
yNextButton.setToolTipText("Click to go to the next y value");
yPrevButton.setToolTipText("Click to go to the previous y value");

// set the sizes of the buttons
int prevWidth = prevIcon.getIconWidth() + 2;
int nextWidth = nextIcon.getIconWidth() + 2;
int prevHeight = prevIcon.getIconHeight() + 2;
int nextHeight = nextIcon.getIconHeight() + 2;
Dimension prevDimension = new Dimension(prevWidth,prevHeight);
Dimension nextDimension = new Dimension(nextWidth, nextHeight);
xPrevButton.setPreferredSize(prevDimension);
yPrevButton.setPreferredSize(prevDimension);
xNextButton.setPreferredSize(nextDimension);
yNextButton.setPreferredSize(nextDimension);

// handle previous x button press
xPrevButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    xIndex--;
    if (xIndex < 0)
      xIndex = 0;
    displayPixelInformation(xIndex,yIndex);
  }
});

// handle previous y button press
yPrevButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    yIndex--;
    if (yIndex < 0)
      yIndex = 0;
    displayPixelInformation(xIndex,yIndex);
  }
});

// handle next x button press
xNextButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    xIndex++;
    if (xIndex >= picture.getWidth())
      xIndex = picture.getWidth() - 1;
    displayPixelInformation(xIndex,yIndex);
  }
});

// handle next y button press
yNextButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    yIndex++;
```

```
      if (yIndex >= picture.getHeight())
        yIndex = picture.getHeight() - 1;
      displayPixelInformation(xIndex,yIndex);
    }
  });
}

/**
 * Create the pixel location panel
 * @param labelFont the font for the labels
 * @return the location panel
 */
public JPanel createLocationPanel(Font labelFont) {

  // create a location panel
  JPanel locationPanel = new JPanel();
  locationPanel.setLayout(new FlowLayout());
  Box hBox = Box.createHorizontalBox();

  // create the labels
  xLabel = new JLabel("X:");
  yLabel = new JLabel("Y:");

  // create the text fields
  xValue = new JTextField(Integer.toString(xIndex + numberBase),6);
  xValue.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      displayPixelInformation(xValue.getText(),yValue.getText());
    }
  });
  yValue = new JTextField(Integer.toString(yIndex + numberBase),6);
  yValue.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     displayPixelInformation(xValue.getText(),yValue.getText());
    }
  });

  // set up the next and previous buttons
  setUpNextAndPreviousButtons();

  // set up the font for the labels
  xLabel.setFont(labelFont);
  yLabel.setFont(labelFont);
  xValue.setFont(labelFont);
  yValue.setFont(labelFont);

  // add the items to the vertical box and the box to the panel
  hBox.add(Box.createHorizontalGlue());
  hBox.add(xLabel);
  hBox.add(xPrevButton);
  hBox.add(xValue);
```

```
  hBox.add(xNextButton);
  hBox.add(Box.createHorizontalStrut(10));
  hBox.add(yLabel);
  hBox.add(yPrevButton);
  hBox.add(yValue);
  hBox.add(yNextButton);
  locationPanel.add(hBox);
  hBox.add(Box.createHorizontalGlue());

  return locationPanel;
}

/**
 * Create the color information panel
 * @param labelFont the font to use for labels
 * @return the color information panel
 */

private JPanel createColorInfoPanel(Font labelFont)
{
  // create a color info panel
  JPanel colorInfoPanel = new JPanel();
  colorInfoPanel.setLayout(new FlowLayout());

  // get the pixel at the x and y
  Pixel pixel = new Pixel(picture,xIndex,yIndex);

  // create the labels
  rValue = new JLabel("R: " + pixel.getRed());
  gValue = new JLabel("G: " + pixel.getGreen());
  bValue = new JLabel("B: " + pixel.getBlue());

  // create the sample color panel and label
  colorLabel = new JLabel("Color at location: ");
  colorPanel = new JPanel();
  colorPanel.setBorder(new LineBorder(Color.black,1));

  // set the color sample to the pixel color
  colorPanel.setBackground(pixel.getColor());

  // set the font
  rValue.setFont(labelFont);
  gValue.setFont(labelFont);
  bValue.setFont(labelFont);
  colorLabel.setFont(labelFont);
  colorPanel.setPreferredSize(new Dimension(25,25));

  // add items to the color information panel
  colorInfoPanel.add(rValue);
  colorInfoPanel.add(gValue);
  colorInfoPanel.add(bValue);
```

```
  colorInfoPanel.add(colorLabel);
  colorInfoPanel.add(colorPanel);

  return colorInfoPanel;
}

/**
 * Creates the North JPanel with all the pixel location
 * and color information
 */
private void createInfoPanel()
{
  // create the info panel and set the layout
  JPanel infoPanel = new JPanel();
  infoPanel.setLayout(new BorderLayout());

  // create the font
  Font largerFont = new Font(infoPanel.getFont().getName(),
                             infoPanel.getFont().getStyle(),14);

  // create the pixel location panel
  JPanel locationPanel = createLocationPanel(largerFont);

  // create the color informaiton panel
  JPanel colorInfoPanel = createColorInfoPanel(largerFont);

  // add the panels to the info panel
  infoPanel.add(BorderLayout.NORTH,locationPanel);
  infoPanel.add(BorderLayout.SOUTH,colorInfoPanel);

  // add the info panel
  pictureFrame.getContentPane().add(BorderLayout.NORTH,infoPanel);
}

/**
 * Method to check that the current position is in the viewing area and if
 * not scroll to center the current position if possible
 */
public void checkScroll()
{
  // get the x and y position in pixels
  int xPos = (int) (xIndex * zoomFactor);
  int yPos = (int) (yIndex * zoomFactor);

  // only do this if the image is larger than normal
  if (zoomFactor > 1) {

    // get the rectangle that defines the current view
    JViewport viewport = scrollPane.getViewport();
    Rectangle rect = viewport.getViewRect();
    int rectMinX = (int) rect.getX();
```

```
    int rectWidth = (int) rect.getWidth();
    int rectMaxX = rectMinX + rectWidth - 1;
    int rectMinY = (int) rect.getY();
    int rectHeight = (int) rect.getHeight();
    int rectMaxY = rectMinY + rectHeight - 1;

    // get the maximum possible x and y index
    int maxIndexX = (int) (picture.getWidth() * zoomFactor) - rectWidth - 1;
    int maxIndexY = (int) (picture.getHeight() * zoomFactor) - rectHeight - 1;

    // calculate how to position the current position in the middle of the viewing
    // area
    int viewX = xPos - (int) (rectWidth / 2);
    int viewY = yPos - (int) (rectHeight / 2);

    // reposition the viewX and viewY if outside allowed values
    if (viewX < 0)
      viewX = 0;
    else if (viewX > maxIndexX)
      viewX = maxIndexX;
    if (viewY < 0)
      viewY = 0;
    else if (viewY > maxIndexY)
      viewY = maxIndexY;

    // move the viewport upper left point
    viewport.scrollRectToVisible(new Rectangle(viewX,viewY,rectWidth,rectHeight));
  }
}

/**
 * Zooms in the on picture by scaling the image.
 * It is extremely memory intensive.
 * @param factor the amount to zoom by
 */
public void zoom(double factor)
{
  // save the current zoom factor
  zoomFactor = factor;

  // calculate the new width and height and get an image that size
  int width = (int) (picture.getWidth()*zoomFactor);
  int height = (int) (picture.getHeight()*zoomFactor);
  BufferedImage bimg = picture.getBufferedImage();

  // set the scroll image icon to the new image
  imageDisplay.setImage(bimg.getScaledInstance(width, height, Image.SCALE_DEFAULT));
  imageDisplay.setCurrentX((int) (xIndex * zoomFactor));
  imageDisplay.setCurrentY((int) (yIndex * zoomFactor));
  imageDisplay.revalidate();
  checkScroll();  // check if need to reposition scroll
```

```
}

/**
 * Repaints the image on the scrollpane.
 */
public void repaint()
{
  pictureFrame.repaint();
}

//*****************************************//
//               Event Listeners           //
//*****************************************//

/**
 * Called when the mouse is dragged (button held down and moved)
 * @param e the mouse event
 */
public void mouseDragged(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method to check if the given x and y are in the picture
 * @param x the horiztonal value
 * @param y the vertical value
 * @return true if the x and y are in the picture and false otherwise
 */
private boolean isLocationInPicture(int x, int y)
{
  boolean result = false; // the default is false
  if (x >= 0 && x < picture.getWidth() &&
      y >= 0 && y < picture.getHeight())
    result = true;

  return result;
}

/**
 * Method to display the pixel information from the passed x and y but
 * also converts x and y from strings
 * @param xString the x value as a string from the user
 * @param yString the y value as a string from the user
 */
public void displayPixelInformation(String xString, String yString)
{
  int x = -1;
  int y = -1;
  try {
    x = Integer.parseInt(xString);
```

```
    x = x - numberBase;
    y = Integer.parseInt(yString);
    y = y - numberBase;
  } catch (Exception ex) {
  }

  if (x >= 0 && y >= 0) {
    displayPixelInformation(x,y);
  }
}

/**
 * Method to display pixel information for the passed x and y
 * @param pictureX the x value in the picture
 * @param pictureY the y value in the picture
 */
private void displayPixelInformation(int pictureX, int pictureY)
{
  // check that this x and y is in range
  if (isLocationInPicture(pictureX, pictureY))
  {
    // save the current x and y index
    xIndex = pictureX;
    yIndex = pictureY;

    // get the pixel at the x and y
    Pixel pixel = new Pixel(picture,xIndex,yIndex);

    // set the values based on the pixel
    xValue.setText(Integer.toString(xIndex  + numberBase));
    yValue.setText(Integer.toString(yIndex + numberBase));
    rValue.setText("R: " + pixel.getRed());
    gValue.setText("G: " + pixel.getGreen());
    bValue.setText("B: " + pixel.getBlue());
    colorPanel.setBackground(new Color(pixel.getRed(), pixel.getGreen(), pixel.getBlue()));

  }
  else
  {
    clearInformation();
  }

  // notify the image display of the current x and y
  imageDisplay.setCurrentX((int) (xIndex * zoomFactor));
  imageDisplay.setCurrentY((int) (yIndex * zoomFactor));
}

/**
 * Method to display pixel information based on a mouse event
 * @param e a mouse event
 */
```

```
private void displayPixelInformation(MouseEvent e)
{

  // get the cursor x and y
  int cursorX = e.getX();
  int cursorY = e.getY();

  // get the x and y in the original (not scaled image)
  int pictureX = (int) (cursorX / zoomFactor + numberBase);
  int pictureY = (int) (cursorY / zoomFactor + numberBase);

  // display the information for this x and y
  displayPixelInformation(pictureX,pictureY);

}

/**
 * Method to clear the labels and current color and reset the
 * current index to -1
 */
private void clearInformation()
{
  xValue.setText("N/A");
  yValue.setText("N/A");
  rValue.setText("R: N/A");
  gValue.setText("G: N/A");
  bValue.setText("B: N/A");
  colorPanel.setBackground(Color.black);
  xIndex = -1;
  yIndex = -1;
}

/**
 * Method called when the mouse is moved with no buttons down
 * @param e the mouse event
 */
public void mouseMoved(MouseEvent e)
{}

/**
 * Method called when the mouse is clicked
 * @param e the mouse event
 */
public void mouseClicked(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method called when the mouse button is pushed down
 * @param e the mouse event
```

```
 */
public void mousePressed(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method called when the mouse button is released
 * @param e the mouse event
 */
public void mouseReleased(MouseEvent e)
{
}

/**
 * Method called when the component is entered (mouse moves over it)
 * @param e the mouse event
 */
public void mouseEntered(MouseEvent e)
{
}

/**
 * Method called when the mouse moves over the component
 * @param e the mouse event
 */
public void mouseExited(MouseEvent e)
{
}

/**
 * Method to enable all menu commands
 */
private void enableZoomItems()
{
  twentyFive.setEnabled(true);
  fifty.setEnabled(true);
  seventyFive.setEnabled(true);
  hundred.setEnabled(true);
  hundredFifty.setEnabled(true);
  twoHundred.setEnabled(true);
  fiveHundred.setEnabled(true);
}

/**
 * Controls the zoom menu bar
 *
 * @param a the ActionEvent
 */
public void actionPerformed(ActionEvent a)
{
```

```
if(a.getActionCommand().equals("Update"))
{
  this.repaint();
}

if(a.getActionCommand().equals("25%"))
{
  this.zoom(.25);
  enableZoomItems();
  twentyFive.setEnabled(false);
}

if(a.getActionCommand().equals("50%"))
{
  this.zoom(.50);
  enableZoomItems();
  fifty.setEnabled(false);
}

if(a.getActionCommand().equals("75%"))
{
  this.zoom(.75);
  enableZoomItems();
  seventyFive.setEnabled(false);
}

if(a.getActionCommand().equals("100%"))
{
  this.zoom(1.0);
  enableZoomItems();
  hundred.setEnabled(false);
}

if(a.getActionCommand().equals("150%"))
{
  this.zoom(1.5);
  enableZoomItems();
  hundredFifty.setEnabled(false);
}

if(a.getActionCommand().equals("200%"))
{
  this.zoom(2.0);
  enableZoomItems();
  twoHundred.setEnabled(false);
}

if(a.getActionCommand().equals("500%"))
{
  this.zoom(5.0);
```

```
    enableZoomItems();
    fiveHundred.setEnabled(false);
  }
}

/**
 * Test Main.  It will ask you to pick a file and then show it
 */
public static void main( String args[])
{
  Picture p = new Picture(FileChooser.pickAFile());
  PictureExplorer test = new PictureExplorer(p);

}

/**
 * Class for establishing the focus for the textfields
 */
private class PictureExplorerFocusTraversalPolicy
               extends FocusTraversalPolicy {

       /**
        * Method to get the next component for focus
        */
       public Component getComponentAfter(Container focusCycleRoot,
                                          Component aComponent) {
           if (aComponent.equals(xValue))
             return yValue;
           else
             return xValue;
       }

       /**
        * Method to get the previous component for focus
        */
        public Component getComponentBefore(Container focusCycleRoot,
                                    Component aComponent) {
           if (aComponent.equals(xValue))
             return yValue;
           else
             return xValue;
        }

        public Component getDefaultComponent(Container focusCycleRoot) {
           return xValue;
       }

       public Component getLastComponent(Container focusCycleRoot) {
           return yValue;
       }
```

```
        public Component getFirstComponent(Container focusCycleRoot) {
            return xValue;
        }
    }

}
```

-end-

6.3.3.3 Java OOP: Handling document events on a text field and creating a color swatch[68]

6.3.3.3.1 Table of Contents

6.3.3.3.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

6.3.3.3.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

####### 6.3.3.3.2.1.1 Figures

####### 6.3.3.3.2.1.2 Listings

[68] This content is available online at <http://cnx.org/content/m44921/1.2/>.

- Listing 8 (p. 1617) . The insertUpdate method.
- Listing 9 (p. 1617) . The paintColorSwatch method.
- Listing 10 (p. 1618) . Complete program listing.

6.3.3.3.3 Preview

In this module, I will show you how to handle document events on text fields containing color values.

I will also show you how to create a color swatch that displays the color indicated by the color values in the Red, Green, and Blue text fields.

Program output at startup

Figure 1 (p. 1611) shows the program output at startup.

Note that the color values in the Red, Green, and Blue text fields are all zero, the color of the square at the right end of the Green panel is black.

Recall that black is represented by Red, Green, and Blue color values of zero.

Program output at startup.

Figure 6.35: Program output at startup.

Program output after clicking the button

Figure 2 (p. 1612) shows the program output after clicking the button labeled "Set Green Color."

Program output after clicking the button.

Figure 6.36: Program output after clicking the button.

Note that in Figure 2 (p. 1612) ,

- the color values in the Red and Blue text fields are zero,
- the color value in the Green text field is 255,
- the square *(color swatch)* at the right end of the green panel is green with a black outline
- the square is not transparent.

Recall that pure green is represented by Red and Blue color values of zero, with a Green color value of 255.

Program output after entering color values

Figure 3 (p. 1612) shows the program output after manually entering a value of 255 in the Red text field. Note that

- the color values in the Red and Green text fields are 255,
- the color value in the Blue text field is 0, and
- the square at the right end of the green panel is yellow with a black outline.

Program output after entering color values.

Figure 6.37: Program output after entering color values.

Recall that pure yellow is represented by Red and Green color values of 255, with a Blue color value of zero.

Color of square tracks color values in text fields

The square *(color swatch)* always has a black outline as shown in Figure 3 (p. 1612) .

The color in the swatch always tracks the color values in the text fields regardless of whether those values are changed manually, or they are changed by program code.

6.3.3.3.4 Discussion and sample code

A complete listing of the program discussed in this module is provided in Listing 10 (p. 1618) .

Will explain in fragments

I will break this program down and explain it in fragments.

The driver class

The driver class named **Prob13** is shown in Listing 1 (p. 1613) . There is nothing new here.

Listing 6.110: The driver class.

```
    public class Prob13{
  public static void main(String[] args){
    new Prob13Runner();
  }//end main method
}//end class Prob13
```

Beginning of the class named Prob13Runner

The class named **Prob13Runner** begins in Listing 2 (p. 1613) , which declares several instance variables and initializes some of them.

Listing 6.111: Beginning of the class named Prob13Runner .

```
    class Prob13Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new JPanel();

  private JTextField redField = new JTextField("000000");
  private JTextField greenField =
                                new JTextField("000000");
  private JTextField blueField = new JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  private JButton setColorButton =
                          new JButton("Set Green Color");
```

Beginning of constructor for Prob13Runner

The constructor begins in Listing 3 (p. 1614) . The last two statements in Listing 3 (p. 1614) may be new to you.

The **JPanel** referred to by **colorIndicatorPanel** is the color swatch shown in Figure 3 (p. 1612) .

The last two statements in Listing 3 (p. 1614) control the border color, and the size of the **JPanel** *(color swatch)* object.

Listing 6.112: Beginning of constructor for Prob13Runner.

```
  public Prob13Runner(){//constructor

setDefaultCloseOperation(
                      WindowConstants.EXIT_ON_CLOSE);

controlPanel.setLayout(new GridLayout(2,1));
controlPanel.add(colorPanel);
controlPanel.add(buttonPanel);

colorPanel.setBackground(Color.GREEN);
colorPanel.add(new JLabel("Red = "));
colorPanel.add(redField);
colorPanel.add(new JLabel(" Green = "));
colorPanel.add(greenField);
colorPanel.add(new JLabel(" Blue = "));
colorPanel.add(blueField);
colorPanel.add(colorIndicatorPanel);

colorIndicatorPanel.setBorder(
                      new LineBorder(Color.black,1));
colorIndicatorPanel.setPreferredSize(
                                 new Dimension(20,20));
```

The constructor continues in Listing 4 (p. 1614) .

There is nothing new in Listing 4 (p. 1614) with the possible exception of the call to the method named **paintColorSwatch** .

I will explain the method named **paintColorSwatch** later.

Listing 6.113: Continue the constructor.

```
    buttonPanel.setBackground(Color.BLUE);
buttonPanel.add(setColorButton);

//Color the swatch for the first time.
paintColorSwatch();

//Add the controlPanel to the content pane, adjust to
// the correct size, and set the title.
getContentPane().add(controlPanel);
pack();
setTitle("Dick Baldwin");

//Make the GUI visible
setVisible(true);
```

Register listeners on the user input components

This program registers event listener objects on four input components:

One component is the button at the bottom of Figure 3 (p. 1612) labeled **Set Green Color.**

The other three components are the three text fields labeled **Red** , **Green** , and **Blue** shown in Figure 3 (p. 1612) .

The three event listeners that are registered on the text fields are very similar. Therefore, I will explain only one of them. You can view the code for the other two in Listing 10 (p. 1618) .

Register a listener object on the button

Listing 5 (p. 1615) registers an ActionListener object on the **JButton** object referred to by **setColorButton** .

You have seen code like this in numerous previous modules. Therefore, a detailed explanation of Listing 5 (p. 1615) should not be required here.

Listing 6.114: Register a listener object on the button.

```
    setColorButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      //Set the color to green.
      redInt = 0;
      greenInt = 255;
      blueInt = 0;

      //Show the color values in the text fields.
      redField.setText("" + redInt);
      greenField.setText("" + greenInt);
      blueField.setText("" + blueInt);

    }//end action performed
  }//end new ActionListener
);//end addActionListener
```

The DocumentListener interface

The **DocumentListener** interface can be described briefly as follows:

"Interface for an observer to register to receive notifications of changes to a text document."

This interface is implemented by numerous classes including the **JTextField** class. As mentioned earlier, the **Red** , **Green** , and **Blue** text fields in Figure 3 (p. 1612) are objects of the **JTextField** class and therefore implement the **DocumentListener** interface.

DocumentListener methods

The **DocumentListener** interface declares the follow three methods:

- **changedUpdate**
  - Gives notification that an attribute or set of attributes changed.
- **insertUpdate**
  - Gives notification that there was an insert into the document.
- **removeUpdate**
  - Gives notification that a portion of the document has been removed.

Only the **insertUpdate** and **removeUpdate** methods are of interest in this program. Therefore, the **changedUpdate** method will be implemented as an empty method.

Beginning of DocumentListener

Listing 6 (p. 1616) shows the beginning of the definition, instantiation, and registration of a **DocumentListener** object on the object of type **Document** that is encapsulated in the Red text field.

Listing 6.115: Beginning of DocumentListener on Red text field.

```
  redField.getDocument().addDocumentListener(
new DocumentListener(){

  public void changedUpdate(DocumentEvent e){
    //Empty method - not needed
  }//end changedUpdate
```

Note that registration of the listener object in this case has an additional level of indirection *(getDocument)* as compared to the registration of the listener on the JButton in Listing 5 (p. 1615) . In other words, the listener is not registered on the text field. Instead, it is registered on the document encapsulated in the text field.

This listener will respond when the contents of the text field are modified, either by the program, or by the user.

(As explained earlier, the ***changedUpdate*** *method is defined as an empty method.)*

The removeUpdate method

Listing 7 (p. 1616) shows the **removeUpdate** method in its entirety.

Listing 6.116: The removeUpdate method.

```
  public void removeUpdate(DocumentEvent e){
try{
  redInt = Integer.parseInt(
                        redField.getText());
  if((redInt >= 0) && (redInt <= 255)){
    paintColorSwatch();
  }//end if
}catch(Exception ex){
  //do nothing on exception
  }//end catch
}//end removeUpdate
```

Behavior of the removeUpdate method

Listing 7 (p. 1616) calls the static **parseInt** method of the **Integer** class to convert the **String** contents of the Red text field into a value of type **int** .

The **parseInt** method can throw a **NumberFormatException** if the string does not contain a parsable integer.

The possibility of a **NumberFormatException** , *(which is a checked exception)* requires that the call to the **parseInt** method be enclosed in a **try-catch** block.

In this program, the **catch** block causes the **removeUpdate** method to ***do nothing*** if the contents of the text field cannot be converted into an **int** , which causes a **NumberFormatException** to be thrown

If a **NumberFormatException** exception is not thrown, the return value from the **parseInt** method is stored in an instance variable named **redInt** .

If the value of **redInt** is between 0 and 255 inclusive, Listing 7 (p. 1616) calls the method named **paintColorSwatch** , causing the color swatch to be repainted, using the **Red** value, along with the **Green** and **Blue** values computed elsewhere.

The insertUpdate method

The **insertUpdate** method is shown in Listing 8 (p. 1617) .

The behavior of the **insertUpdate** method is essentially the same as the behavior of the **removeUpdate** method explained earlier. Therefore, an explanation of the **insertUpdate** method should not be needed.

Listing 6.117: The insertUpdate method.

```
        public void insertUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                              redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end insertUpdate
    //--------------------------------------------//

  }//end new DocumentListener
);//end addDocumentListener
```

The end of the class definition

Listing 8 (p. 1617) also signals the end of the anonymous class definition that began in Listing 6 (p. 1616) .

The Green and Blue text fields

The Green and Blue text fields are processed using very similar **DocumentListener** objects as used for the Red text field. You can view that code in Listing 10 (p. 1618) .

That ends the discussion of the constructor for the class named **Prob13Runner** .

The method named paintColorSwatch

There were earlier references to a method named **paintColorSwatch** . The code for the **paintColorSwatch** method is shown in Listing 9 (p. 1617) .

Listing 6.118: The paintColorSwatch method.

```
  private void paintColorSwatch(){
 colorIndicatorPanel.setBackground(
                  new Color(redInt,greenInt,blueInt));
}//end paintColorSwatch
//-----------------------------------------------------//

}//end class Prob13Runner
```

The **paintColorSwatch** method sets the background color for the **JPanel** object that represents the color swatch, using values of red, green, and blue that are computed elsewhere in the program.

The end of the program

Listing 9 (p. 1617) also signals the end of the class named **Prob13Runner** and the end of the program.

6.3.3.3.5 Run the program

I encourage you to copy the code from Listing 10 (p. 1618) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.3.3.6 Summary

In this lesson, you learned how to handle document events on text fields containing color values. You also learned how to create a color swatch.

6.3.3.3.7 What's next?

In the next module, you will learn how to use a JColorChooser object to specify a color in any one of five different ways.

6.3.3.3.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Handling document events on a text field and creating a color swatch
- File: Java1326.htm
- Published: 09/11/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.3.3.9 Complete program listing

A complete listing of the source code discussed in this module is shown in Listing 10 (p. 1618) .

Listing 6.119: Complete program listing.

```
/*File Prob13 Copyright 2012 R.G.Baldwin

This program handles document events on the contents of
text fields containing color values.
```

```
The values are used to create a color swatch that displays
the color indicated by the color values in the red, green,
and blue text fields.
*******************************************************/
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;

public class Prob13{
  public static void main(String[] args){
    new Prob13Runner();
  }//end main method
}//end class Prob13
//=======================================================//

class Prob13Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new JPanel();

  private JTextField redField = new JTextField("000000");
  private JTextField greenField =
                                new JTextField("000000");
  private JTextField blueField = new JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  private JButton setColorButton =
                          new JButton("Set Green Color");
  //-----------------------------------------------------//

  public Prob13Runner(){//constructor

    setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);
```

```
    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    colorIndicatorPanel.setBorder(
                          new LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                  new Dimension(20,20));

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(setColorButton);

    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content pane, adjust to
    // the correct size, and set the title.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);

    //-----------------------------------------------------//
    //Register listeners on the user input components.
    //-----------------------------------------------------//
    setColorButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          //Set the color to green.
          redInt = 0;
          greenInt = 255;
          blueInt = 0;

          //Show the color values in the text fields.
          redField.setText("" + redInt);
          greenField.setText("" + greenInt);
          blueField.setText("" + blueInt);

        }//end action performed
```

```
  }//end new ActionListener
);//end addActionListener
//---------------------------------------------------//

//Register a document listener on the red text field.
// This listener will respond when the contents of
// the text field are modified either by the program
// or by the user.
redField.getDocument().addDocumentListener(
  new DocumentListener(){
    public void changedUpdate(DocumentEvent e){
      //Empty method - not needed
    }//end changedUpdate

    public void removeUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                              redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end removeUpdate

    public void insertUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                              redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end insertUpdate

  }//end new DocumentListener
);//end addDocumentListener
//---------------------------------------------------//

//Register a document listener on the green text
// field. Essentially the same as the above.
greenField.getDocument().addDocumentListener(
  new DocumentListener(){
    public void changedUpdate(DocumentEvent e){}

    public void removeUpdate(DocumentEvent e){
      try{
        greenInt = Integer.parseInt(
```

```
                            greenField.getText());
          if((greenInt >= 0) && (greenInt <= 255))
          {
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end removeUpdate

      public void insertUpdate(DocumentEvent e){
        try{
          greenInt = Integer.parseInt(
                            greenField.getText());
          if((greenInt >= 0) && (greenInt <= 255))
          {
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-------------------------------------------------//

  //Register a document listener on the blue text
  // field. Essentially the same as the above.
  blueField.getDocument().addDocumentListener(
    new DocumentListener(){
      public void changedUpdate(DocumentEvent e){}

      public void removeUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                            blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end removeUpdate

      public void insertUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                            blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
```

```
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-----------------------------------------------//
}//end constructor
//-------------------------------------------------//

//The purpose of this method is to color a swatch
// located next to the RGB color values.
private void paintColorSwatch(){
  colorIndicatorPanel.setBackground(
                    new Color(redInt,greenInt,blueInt));
}//end paintColorSwatch
//-------------------------------------------------//


}//end class Prob13Runner
```

-end-

6.3.3.4 Java OOP: Using a JColorChooser object[69]

6.3.3.4.1 Table of Contents

6.3.3.4.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

[69] This content is available online at <http://cnx.org/content/m44923/1.2/>.

6.3.3.4.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.3.4.2.1.1 Figures

6.3.3.4.2.1.2 Listings

6.3.3.4.3 Preview

In this module, you will learn how to use a **JColorChooser** object to specify a color in any one of five different ways:

- Swatches
- HSV
- HSL
- RGB
- CMYK

The details regarding the five different ways are not explained. Instead, an understanding of the five ways for specifying a color is left as an exercise for the student.

This module concentrates on the programming aspects of the color chooser as opposed to the aesthetic aspects of the color chooser.

You will also learn how to create brighter and darker shades of a given color.

What is a JColorChooser object?

According to the Java documentation,

"JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color."

Program output at startup

A complete listing of the program discussed in this module is provided in Listing 4 (p. 1631) near the end of the module.

Figure 1 (p. 1625) shows the program output at startup.

The program output is a GUI containing

- three text fields, one each for red, green, and blue color values
- three labels that identify the contents of each text field
- a button labeled **Choose Color**
- two buttons labeled **Brighter** and **Darker**
- a square ***color swatch*** whose color reflects the color specified by the color component values in the text fields

Program output at startup.

Figure 6.38: Program output at startup.

Program output after clicking the Choose Color button.

Figure 2 (p. 1626) shows the result of clicking the **Choose Color** button shown in Figure 1 (p. 1625) .

The bottom image in Figure 2 (p. 1626) is an object of the **JColorChooser** class.

(The color chooser object doesn't actually appear below the GUI as shown in Figure 2 (p. 1626) *. Instead, it appears in the upper-left corner of the screen and hides the GUI. I manually dragged it down below the GUI to produce this image.)*

Program output after clicking the Choose Color button.

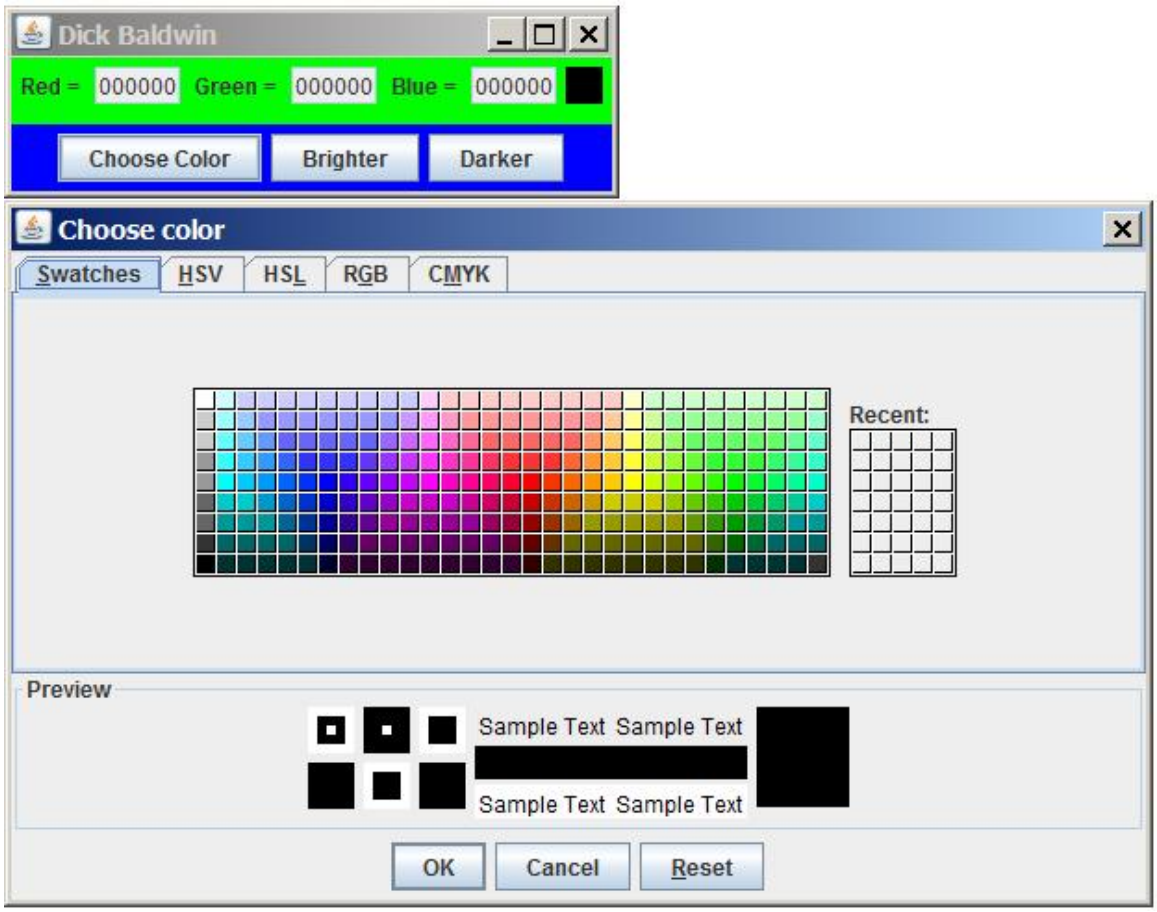


Figure 6.39: Program output after clicking the Choose Color button.

The initial color selection

Note in Figure 2 (p. 1626) that the selected color when the color chooser appears matches the color of the square color swatch in Figure 1 (p. 1625) *(black)*

Program output after selecting a reddish color and clicking the OK button.

Figure 3 (p. 1627) shows the result of

- selecting a reddish color in the color chooser, and
- clicking the OK button in the color chooser.

Program output after selecting a reddish color and clicking the OK button.

Figure 6.40: Program output after selecting a reddish color and clicking the OK button.

New color in the GUI

Note in Figure 3 (p. 1627) that

- the color selected in the color chooser now appears in the square color swatch in Figure 3 (p. 1627)
- the color swatch in Figure 3 (p. 1627) has a black border
- the color values in the three text fields describe the color showing in the swatch

(The color in the swatch is based on the three color values in the text fields.)

Program output after clicking the Darker button.

Figure 4 (p. 1627) shows the result of clicking the **Darker** button once.

Program output after clicking the Darker button.

Figure 6.41: Program output after clicking the Darker button.

A darker color

The color swatch in Figure 4 (p. 1627) shows a darker version of the color swatch shown in Figure 3 (p. 1627) .

The numeric color values in Figure 4 (p. 1627) describe the color in the color swatch, and are lower than the corresponding color values in Figure 3 (p. 1627)

A brighter color

Although not demonstrated here, the color in the swatch could be made brighter by clicking the **Brighter** button.

If you click the **Darker** button enough times, the color will go to black. Similarly, if you click the **Brighter** button enough times, the color will go to white.

6.3.3.4.4 Discussion and sample code

A complete listing of the program is provided in Listing 4 (p. 1631) .

Much of the code in this program is very similar to code that I explained in earlier modules. Therefore, I won't repeat those explanations here. Instead, I will concentrate on the code that is new and different.

Given that caveat, I will skip all the way down to the event handler for the **Choose Color** button.

Event handler for the Choose Color button

Listing 1 (p. 1628) defines, instantiates, and registers an **ActionListener** object on the **Choose Color** button.

Listing 6.120: Event handler for the Choose Color button.

```
    chooseButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      Color selColor = JColorChooser.showDialog(
                chooseButton,"Choose color",new Color(
                   redInt,greenInt,blueInt));
      if(selColor != null){
        //Don't change the color if the user cancels
        // out.
        redField.setText("" + selColor.getRed());
        greenField.setText("" + selColor.getGreen());
        blueField.setText("" + selColor.getBlue());
      }//end if

    }//end action performed
  }//end new ActionListener
);//end addActionListener
```

Show the JColorChooser object

Listing 2 (p. 1628) shows the code, *(extracted from Listing 1* (p. 1628)) that causes the **color chooser** to appear on the screen.

Listing 6.121: Show the JColorChooser object.

```
Color selColor = JColorChooser.showDialog(
                chooseButton,
                "Choose color",
                new Color(redInt,greenInt,blueInt));
```

Call the static showDialog method

Listing 2 (p. 1628) calls the static **showDialog** method of the **JColorChooser** class.

Figure 5 (p. 1629) provides information for the **showDialog** method.

The showDialog method.

```
public static Color showDialog(
                      Component component,
                      String title,
                      Color initialColor)
                        throws HeadlessException

Shows a modal color-chooser dialog and blocks
until the dialog is hidden. If the user presses
the "OK" button, then this method hides/disposes
the dialog and returns the selected color. If
the user presses the "Cancel" button or closes
the dialog without pressing "OK", then this
method hides/disposes the dialog and returns null.

Parameters:
  component - the parent Component for the dialog
  title - the String containing the dialog's title
  initialColor - the initial Color set when the
    color-chooser is shown
Returns:
  the selected color or null if the user opted out
```

Figure 6.42: The showDialog method.

A straightforward explanation

The behavior of the dialog is explained in Figure 5 (p. 1629) and shouldn't require further explanation.

Parameters to the showDialog method

A comparison of Listing 2 (p. 1628) and Figure 5 (p. 1629) shows how the color chooser is integrated into the program.

Perhaps the most important aspects of that integration are the *third parameter* and the *return value* .

The third parameter

The third parameter is an anonymous **Color** object that matches the color specified by the text fields in the GUI, which determine the color of the swatch.

Thus, the initial color for the color chooser matches the color of the swatch in the GUI when the color chooser first appears.

The return value

The return value from the color chooser is a reference to an object of type **Color** , which is saved in a variable named **selColor** .

Processing the return value

Returning to the code in Listing 1 (p. 1628) ,

- if the return value is not null,
- the color components of the return value are converted to strings and
- stored in the red, green, and blue text fields.

What happens next?

You learned in an earlier module that if the values stored in any of the three text fields changes for any reason, **DocumentListener** objects registered on the three text fields cause the color of the swatch to change to match the new values of the color components.

Thus, the color that is chosen by the user from the color chooser modifies the values in the text fields causing the color of the swatch in the GUI to match the chosen color.

Darkening and brightening the color

Listing 3 (p. 1630) shows an **ActionListener** object being registered on the button labeled **Darker** in Figure 1 (p. 1625) .

Listing 6.122: Darkening the color.

```
    darkerButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      Color color = new Color(
           redInt,greenInt,blueInt).darker();
      redField.setText("" + color.getRed());
      greenField.setText("" + color.getGreen());
      blueField.setText("" + color.getBlue());
    }//end action performed
  }//end newActionListener
);//end addActionListener
```

The darkening action listener

This code creates a new **Color** object that matches the color values in the three text fields.

Then it calls the **darker** method on that **Color** object causing its color to be darkened.

Then it extracts the color components from the darker **Color** object and stores those values in the text fields.

This, in turn, causes the color swatch to change to match the new color values.

A brightening event handler

A similar event handler is registered on the button labeled **Brighter** in Figure 1 (p. 1625) .

This code, which you can view in Listing 4 (p. 1631) calls the **brighter** method instead of the **darker** method on the **Color** object.

End of story

That concludes the explanation of material that is new and different in this program.

6.3.3.4.5 Run the program

I encourage you to copy the code from Listing 4 (p. 1631) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.3.4.6 Summary

In this module, you will learned how to use a **JColorChooser** object to specify a color in any one of five different ways.

You also learned how to create brighter and darker shades of a given color.

6.3.3.4.7 What's next?

In the next module, you will learn how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

6.3.3.4.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Using a JColorChooser object
- File: Java3128.htm
- Published: 09/11/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.3.4.9 Complete program listing

A complete listing of the source code discussed in this module is shown in Listing 4 (p. 1631) .

Listing 6.123: Complete program listing.

```
/*File Prob14 Copyright 2012 R.G.Baldwin

Old material:
This program services document events on the contents of
text fields containing color values.

The values are used to create a color swatch that displays
the color indicated by the color values in the red, green,
and blue text fields.

New material:
Demonstrates how to create and use a JColorChooser dialog.
Also demonstrates use of the darker and brighter methods.
Also demonstrates how to cause JTextField objects to be
non-editable.
**********************************************************/
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;
import javax.swing.JColorChooser;

public class Prob14{
  public static void main(String[] args){
    new Prob14Runner();
  }//end main method
}//end class Prob14
//=======================================================//

class Prob14Runner extends JFrame{

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new JPanel();

  private JTextField redField = new JTextField("000000");
  private JTextField greenField =
                                 new JTextField("000000");
  private JTextField blueField = new JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  private JButton chooseButton =
                             new JButton("Choose Color");
  private JButton brighterButton =
                               new JButton("Brighter");
  private JButton darkerButton = new JButton("Darker");
  //-----------------------------------------------------//

  public Prob14Runner(){//constructor

    setDefaultCloseOperation(
                         WindowConstants.EXIT_ON_CLOSE);
```

```
controlPanel.setLayout(new GridLayout(2,1));
controlPanel.add(colorPanel);
controlPanel.add(buttonPanel);

colorPanel.setBackground(Color.GREEN);
colorPanel.add(new JLabel("Red = "));
colorPanel.add(redField);
colorPanel.add(new JLabel(" Green = "));
colorPanel.add(greenField);
colorPanel.add(new JLabel(" Blue = "));
colorPanel.add(blueField);
colorPanel.add(colorIndicatorPanel);

redField.setEditable(false);
greenField.setEditable(false);
blueField.setEditable(false);

colorIndicatorPanel.setBorder(
                      new LineBorder(Color.black,1));
colorIndicatorPanel.setPreferredSize(
                                new Dimension(20,20));

buttonPanel.setBackground(Color.BLUE);
buttonPanel.add(chooseButton);
buttonPanel.add(brighterButton);
buttonPanel.add(darkerButton);

//Color the swatch for the first time.
paintColorSwatch();

//Add the controlPanel to the content pane, adjust to
// the correct size, and set the title.
getContentPane().add(controlPanel);
pack();
setTitle("Dick Baldwin");

//Make the GUI visible
setVisible(true);

//-------------------------------------------------//
//Register listeners on the user input components.
//-------------------------------------------------//
chooseButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      Color selColor = JColorChooser.showDialog(
                chooseButton,"Choose color",new Color(
                   redInt,greenInt,blueInt));
      if(selColor != null){
        //Don't change the color if the user cancels
        // out.
```

```
          redField.setText("" + selColor.getRed());
          greenField.setText(
                              "" + selColor.getGreen());
          blueField.setText("" + selColor.getBlue());
        }//end if

      }//end action performed
    }//end new ActionListener
  );//end addActionListener
  //-------------------------------------------------//

  darkerButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){
        Color color = new Color(
             redInt,greenInt,blueInt).darker();
        redField.setText("" + color.getRed());
        greenField.setText("" + color.getGreen());
        blueField.setText("" + color.getBlue());
      }//end action performed
    }//end newActionListener
  );//end addActionListener
  //-------------------------------------------------//

  brighterButton.addActionListener(
    new ActionListener(){
      public void actionPerformed(ActionEvent e){
        Color color = new Color(
           redInt,greenInt,blueInt).brighter();
        redField.setText("" + color.getRed());
        greenField.setText("" + color.getGreen());
        blueField.setText("" + color.getBlue());
      }//end action performed
    }//end newActionListener
  );//end addActionListener
  //-------------------------------------------------//

  //Register a document listener on the red text field.
  // This listener will respond when the contents of
  // the text field are modified either by the program
  // or by the user.
  redField.getDocument().addDocumentListener(
    new DocumentListener(){
      public void changedUpdate(DocumentEvent e){
        //Empty method - not needed
      }//end changedUpdate

      public void removeUpdate(DocumentEvent e){
        try{
          redInt = Integer.parseInt(
                                redField.getText());
```

```
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end removeUpdate

    public void insertUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                              redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end insertUpdate

  }//end new DocumentListener
);//end addDocumentListener
//-----------------------------------------------//

//Register a document listener on the green text
// field. Essentially the same as the above.
greenField.getDocument().addDocumentListener(
  new DocumentListener(){
    public void changedUpdate(DocumentEvent e){}

    public void removeUpdate(DocumentEvent e){
      try{
        greenInt = Integer.parseInt(
                             greenField.getText());
        if((greenInt >= 0) && (greenInt <= 255))
        {
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end removeUpdate

    public void insertUpdate(DocumentEvent e){
      try{
        greenInt = Integer.parseInt(
                             greenField.getText());
        if((greenInt >= 0) && (greenInt <= 255))
        {
          paintColorSwatch();
        }//end if
```

```
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-----------------------------------------------//

  //Register a document listener on the blue text
  // field. Essentially the same as the above.
  blueField.getDocument().addDocumentListener(
    new DocumentListener(){
      public void changedUpdate(DocumentEvent e){}

      public void removeUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                              blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end removeUpdate

      public void insertUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                              blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-----------------------------------------------//
}//end constructor
//-----------------------------------------------//

//The purpose of this method is to color a swatch
// located next to the RGB color values.
private void paintColorSwatch(){
  colorIndicatorPanel.setBackground(
                    new Color(redInt,greenInt,blueInt));
}//end paintColorSwatch
```

```
  //-----------------------------------------------//

  //The purpose of this method is to absorb any exceptions
  // that may be thrown by the parseInt method in order
  // to avoid having the program abort. In the event that
  // an exception is thrown, this method simply returns an
  // int value of 0;
  private int goParseInt(String string){
    int result = 0;
    try{
      result = Integer.parseInt(string);
    }catch(Exception e){
      result = 0;
    }//end catch
    return result;
  }//end goParseInt

  //-----------------------------------------------//


}//end class Prob14Runner
```

-end-

6.3.3.5 Java OOP: A Pixel Color Editor[70]

6.3.3.5.1 Table of Contents

6.3.3.5.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [71] .

[70] This content is available online at <http://cnx.org/content/m44926/1.2/>.
[71] http://cnx.org/content/m44148/latest/

6.3.3.5.2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

6.3.3.5.2.1.1 Figures

6.3.3.5.2.1.2 Listings

6.3.3.5.3 Preview

In this module, you will learn how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

Such a program could be useful, for example, for manually correcting "red eye" problems in digital images.

Program output at startup

Figure 1 (p. 1639) shows the program output at startup.

The output consists of

- an Ericson **PictureExplorer** object displaying an image of a penguin, and
- a color manipulation GUI similar to those discussed in earlier modules.

Program output at startup.

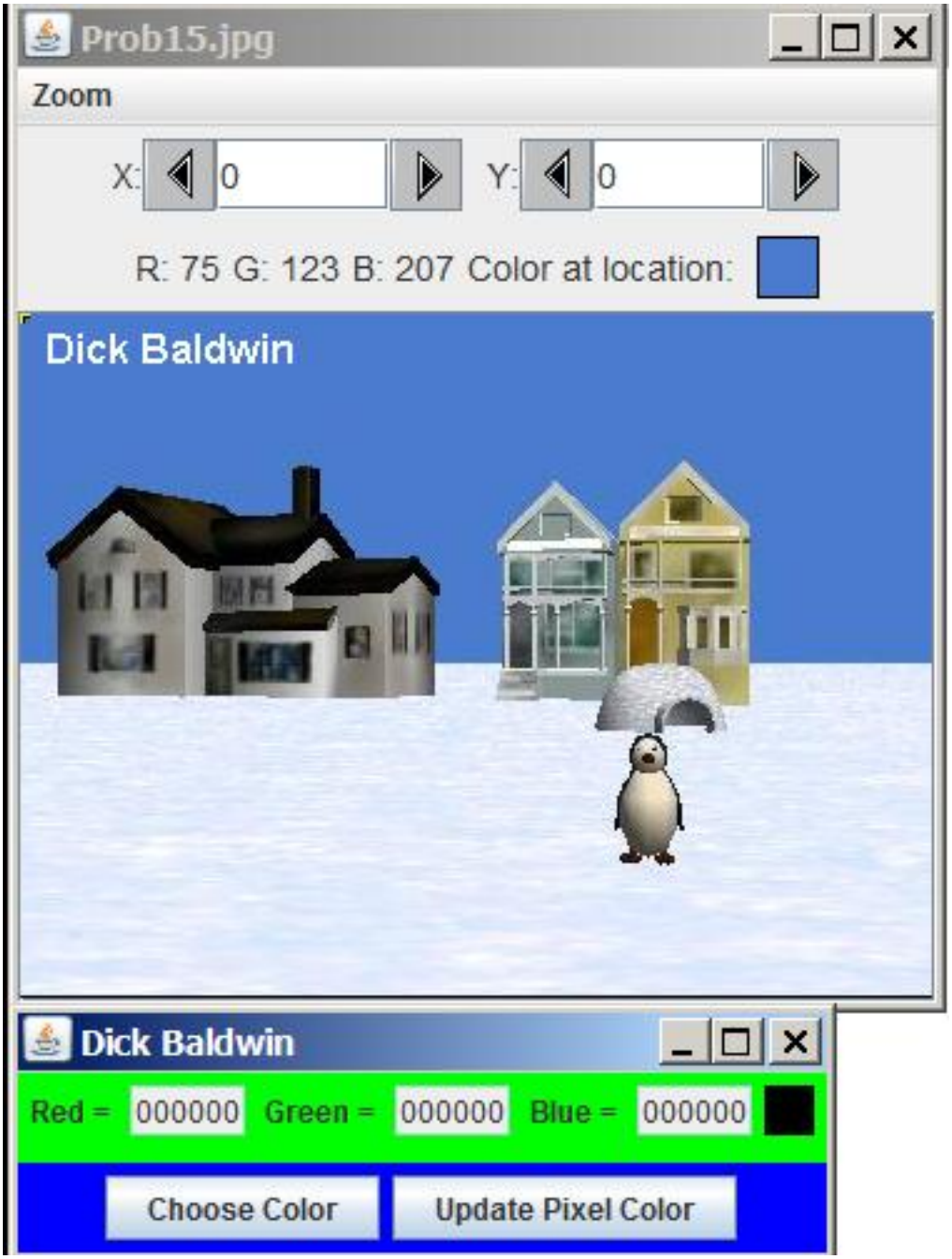


Figure 6.43: Program output at startup.

Program output after zooming the image

Figure 2 (p. 1641) shows the program output after placing the **PictureExplorer** cursor on the penguin's nose, and zooming the **PictureExplorer** image by 500%

If you look carefully, you should be able to see the crosshair cursor on the penguin's nose.

Program output after zooming the image.

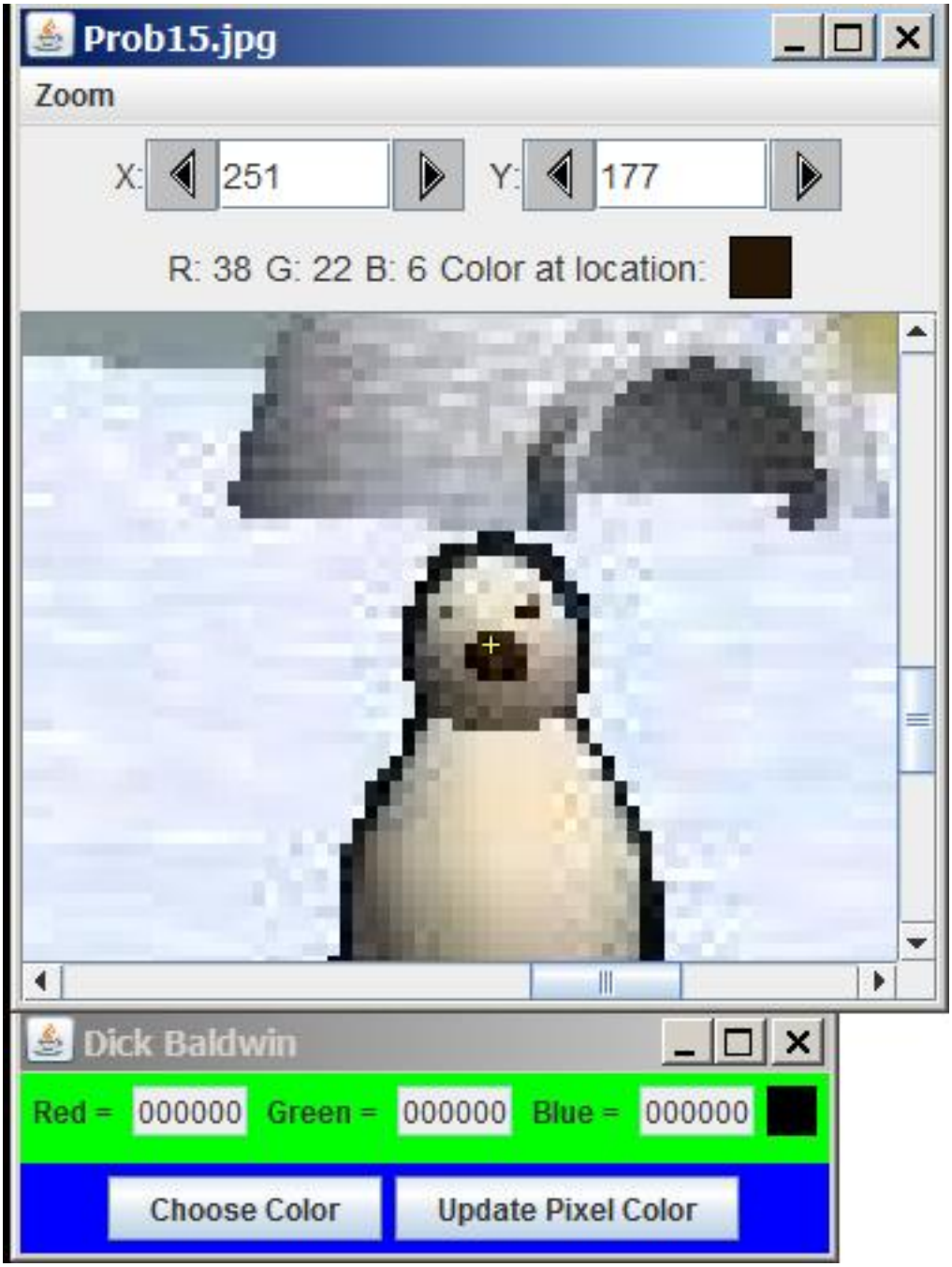


Figure 6.44: Program output after zooming the image.

Program output after changing the pixel color to yellow

Figure 3 (p. 1643) shows the result of selecting the color yellow with the GUI, and clicking the button labeled **Update Pixel Color**

If you look carefully, you should see that the pixel next to the crosshair cursor on the penguin's nose has changed from black to yellow.

Program output after changing the pixel color to yellow.

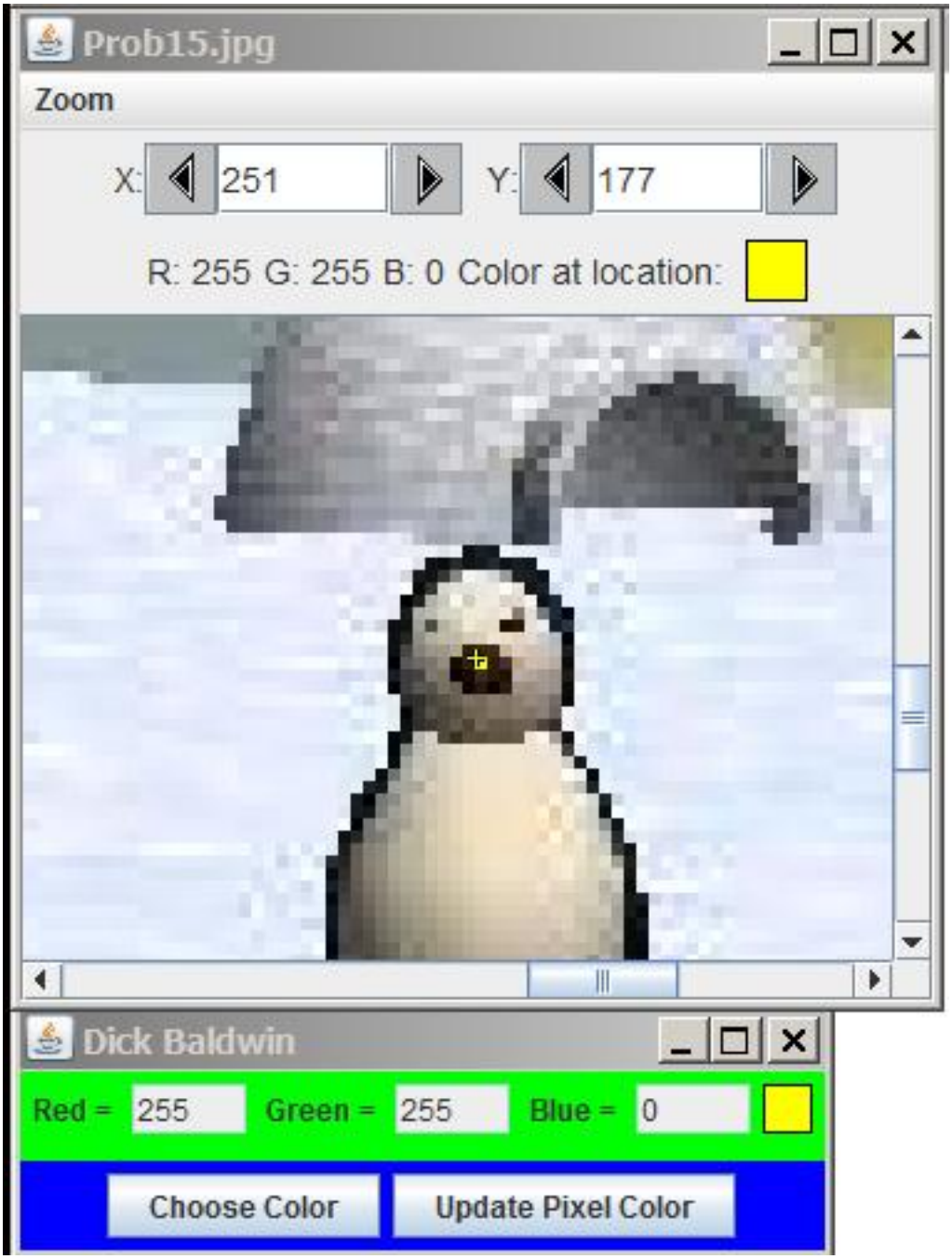


Figure 6.45: Program output after changing the pixel color to yellow.

6.3.3.5.4 Discussion and sample code

This program requires modifications to Ericson's **PictureExplorer** class. I will deal with that issue first.

A complete listing of the modified version of the **PictureExplorer** class is shown in Listing 10 (p. 1655) .

Changes to Ericson's PictureExplorer class

The **PictureExplorer** class was modified to add *getter* methods to cause the following values to be accessible from outside the object:

- int xIndex
- int yIndex
- double zoomFactor
- JFrame pictureFrame

Also the call to the **setDefaultCloseOperation** method was disabled.

Code for methods that were added

Listing 1 (p. 1644) shows the code for the methods that were added to the **PictureExplorer** class.

Listing 6.124: Code for methods that were added.

```
    //Method to get the xIndex value.
public int getXIndex(){
  return xIndex;
}//end getXIndex

//Method to get the yIndex value.
public int getYIndex(){
  return yIndex;
}//end getYIndex

//Method to get the zoomFactor value.
public double getZoomFactor(){
  return zoomFactor;
}//end getZoomFactor

//Method to get a reference to the frame
public JFrame getFrame(){
  return pictureFrame;
}//end getFrame()
```

Modified setDefaultCloseOperation method

Listing 2 (p. 1644) shows the modified version of the **setDefaultCloseOperation** method.

Note the parameter that is passed to the method when it is called.

Listing 6.125: Modified setDefaultCloseOperation method.

```
explorerFrame.setDefaultCloseOperation(
          WindowConstants.DO_NOTHING_ON_CLOSE);
```

Similar to previous code

Much of the code in this program is similar to code that I explained in earlier modules, and I won't repeat those explanations.

I will skip down to the event handler registered on the button labeled **Update Pixel Color** in Figure 1 (p. 1639) . That code begins in Listing 3 (p. 1645) .

Listing 6.126: Beginning of ActionListener registered on update button.

```
  updateButton.addActionListener(
new ActionListener(){
  public void actionPerformed(ActionEvent e){

    //Get properties of PictureExplorer object.
    xIndex = explorer.getXIndex();
    yIndex = explorer.getYIndex();
    zoomFactor = explorer.getZoomFactor();
```

Important to preserve properties

The only thing that changes when you click the **update** button is the color of the pixel at the crosshair cursor.

Therefore, the program must preserve the state of the explorer object including such items as the zoom factor and the location of crosshair cursor.

Listing 3 (p. 1645) gets and saves the property values that determine the state of the explorer object.

Convert zoom factor to String

The **getZoomFactor** method returns the zoom factor as type **double** . However, we need the zoom factor as type **String** . The code in Listing 4 (p. 1645) converts the zoom factor to type **String** .

Listing 6.127: Convert zoom factor to type String.

```
    //Save zoom factor as a string.
  String zoomString = "100%";
  if(zoomFactor == 0.25){
    zoomString = "25%";
  }else if(zoomFactor == 0.50){
    zoomString = "50%";
  }else if(zoomFactor == 0.75){
    zoomString = "75%";
  }else if(zoomFactor == 1.0){
    zoomString = "100%";
  }else if(zoomFactor == 1.5){
    zoomString = "150%";
  }else if(zoomFactor == 2.0){
    zoomString = "200%";
  }else if(zoomFactor == 5.0){
    zoomString = "500%";
  }else{
    zoomString = "100%";//in case no match
  }//end else
```

Set pixel color in original image

The code in Listing 5 (p. 1646)

- creates a new **Color** object based on the values in the text fields of Figure 3 (p. 1643) , and
- changes the color of the corresponding pixel in the original image to the new color.

At this point, the color of the pixel in the original image has been modified.

Listing 6.128: Set pixel color in original image.

```
    Color newColor = new Color(
                        redInt,greenInt,blueInt);
pix.getPixel(xIndex,yIndex).setColor(newColor);
```

Create and populate a new PictureExplorer object

Listing 6 (p. 1646) begins by disposing of the original **PictureExplorer** object. Then Listing 6 (p. 1646) creates a new **PictureExplorer** object containing the modified image.

Finally Listing 6 (p. 1646) calls the **setDefaultCloseOperation** on the **JFrame** that houses the **PictureExplorer** object disabling the X-button in the upper-right corner.

Listing 6.129: Create and populate a new PictureExplorer object.

```
    //Dispose of the existing explorer and create a
// new one.
explorerFrame.dispose();

explorer = new PictureExplorer(pix);

//Get reference to the new frame
explorerFrame = explorer.getFrame();
explorerFrame.setDefaultCloseOperation(
           WindowConstants.DO_NOTHING_ON_CLOSE);
```

Set the state of the PictureExplorer object

You will probably need to study the event handling code in the **PictureExplorer** class to fully understand the remaining code in this discussion.

When you click the image in the **PictureExplorer** object, a **MouseListener** object registered on the **PictureExplorer** object calls the **mousePressed** method belonging to that object

Simulate a physical click

We can simulate a physical click on that image by calling the same method from outside the **PictureExplorer** object.

Call the mousePressed method

Listing 7 (p. 1646) calls the **mousePressed** method on the **PictureExplorer** object passing the crosshair coordinates as parameters. Other parameters are also passed to satisfy the requirements of the **mousePressed** method.

(I will leave it as an exercise for the student to investigate those other parameters.)

Listing 6.130: Call the mousePressed method.

```
    //Now set the state of the new explorer.

//Simulate a mouse pressed event in the picture
// to set the cursor and the text in the
// coordinate fields.
explorer.mousePressed(new MouseEvent(
                new JButton("dummy component"),
                MouseEvent.MOUSE_PRESSED,
                (long)0,
```

```
                    0,
                    xIndex,
                    yIndex,
                    0,
                    false));
```

PictureExplorer object handles crosshair cursor

The code in Ericson's **mousePressed** method takes care of all the requirements that result from setting the location of the crosshair cursor, such as setting the coordinate values in the **PictureExplorer** object's text fields shown in Figure 3 (p. 1643) .

Set the zoom state

Manually selecting a zoom level from the **Zoom** menu in the **PictureExplorer** object causes the **actionPerformed** method belonging to an **ActionListener** object registered on the **PictureExplorer** object to be executed.

As before, we can simulate the manual selection of a **Zoom** menu value by calling that **actionPerformed** method from outside the object.

Set the zoom state

Listing 8 (p. 1647) calls Ericson's **actionPerformed** method, passing the **String** representation of the zoom factor as a parameter.

(I will leave it as an exercise for the student to investigate the other parameters.)

Listing 6.131: Set the zoom state.

```
        //Simulate an action event on the zoom menu to
      // set the zoom.
      explorer.actionPerformed(new ActionEvent(
                       explorer,
                       ActionEvent.ACTION_PERFORMED,
                       zoomString));


    }//end actionPerformed
  }//end newActionListener
);//end addActionListener
```

Conclusion of discussion

That concludes the discussion of the **ActionListener** object registered on the button labeled **Update Pixel Color** in Figure 1 (p. 1639) .

It also concludes the discussion of the program.

6.3.3.5.5 Run the program

I encourage you to copy the code from Listing 9 (p. 1648) and Listing 10 (p. 1655) and download the image file named Prob15.jpg [72] . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.3.3.5.6 Summary

In this module, you learned how to write an editor program that you can use to modify the colors in an image on a pixel-by-pixel basis.

[72] http://cnx.org/content/m44926/latest/Prob15.jpg

6.3.3.5.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: A Pixel Color Editor
- File: Java3130.htm
- Published: 09/11/12

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.3.3.5.8 Complete program listing

Complete listings of the source code discussed in this module are shown in Listing 9 (p. 1648) and Listing 10 (p. 1655) below.

Listing 6.132: Complete listing of Prob15.

```
/*File Prob15 Copyright 2012 R.G.Baldwin

The purpose of this program is demonstrate one way to
change the color of a pixel in a Picture object that is
encapsulated in a PictureExplorer object.

The pixel to be modified is selected by placing the
cursor in the PictureExplorer object.
*******************************************************/
import java.awt.event.MouseEvent;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.WindowConstants;
import javax.swing.event.DocumentListener;
```

```
import javax.swing.event.DocumentEvent;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Dimension;
import javax.swing.JColorChooser;

public class Prob15{
  public static void main(String[] args){
    new Prob15Runner();
  }//end main method
}//end class Prob15
//=======================================================//

class Prob15Runner extends JFrame{
  private Prob15Runner jFrameObj = null;
  private PictureExplorer explorer = null;
  private Picture pix;
  private JFrame explorerFrame;

  private JPanel controlPanel = new JPanel();
  private JPanel colorPanel = new JPanel();
  private JPanel buttonPanel = new JPanel();
  private JPanel colorIndicatorPanel = new JPanel();

  private JTextField redField = new JTextField("000000");
  private JTextField greenField =
                                 new JTextField("000000");
  private JTextField blueField = new JTextField("000000");

  private int redInt = 0;
  private int greenInt = 0;
  private int blueInt = 0;

  //Copies of properties of the PictureExplorer object
  private int xIndex = 0;
  private int yIndex = 0;
  private double zoomFactor = 0;

  private JButton chooseButton =
                              new JButton("Choose Color");
  private JButton updateButton =
                        new JButton("Update Pixel Color");
  //-------------------------------------------------------//

  public Prob15Runner(){//constructor

    pix = new Picture("Prob15.jpg");
    pix.addMessage("Dick Baldwin",10,20);
    explorer = new PictureExplorer(pix);
    explorerFrame = explorer.getFrame();
```

```
    explorerFrame.setDefaultCloseOperation(
               WindowConstants.DO_NOTHING_ON_CLOSE);

    //Set the location for the control GUI
    // immediately below the PictureExplorer object,
    // and set its default close operation.
    setLocation(0,pix.getHeight() + 128);

    setDefaultCloseOperation(
                          WindowConstants.EXIT_ON_CLOSE);

    controlPanel.setLayout(new GridLayout(2,1));
    controlPanel.add(colorPanel);
    controlPanel.add(buttonPanel);

    colorPanel.setBackground(Color.GREEN);
    colorPanel.add(new JLabel("Red = "));
    colorPanel.add(redField);
    colorPanel.add(new JLabel(" Green = "));
    colorPanel.add(greenField);
    colorPanel.add(new JLabel(" Blue = "));
    colorPanel.add(blueField);
    colorPanel.add(colorIndicatorPanel);

    redField.setEditable(false);
    greenField.setEditable(false);
    blueField.setEditable(false);

    colorIndicatorPanel.setBorder(
                          new LineBorder(Color.black,1));
    colorIndicatorPanel.setPreferredSize(
                                   new Dimension(20,20));

    buttonPanel.setBackground(Color.BLUE);
    buttonPanel.add(chooseButton);
    buttonPanel.add(updateButton);
//    buttonPanel.add(darkerButton);

    //Color the swatch for the first time.
    paintColorSwatch();

    //Add the controlPanel to the content pane, adjust to
    // the correct size, and set the tiele.
    getContentPane().add(controlPanel);
    pack();
    setTitle("Dick Baldwin");

    //Make the GUI visible
    setVisible(true);

    //-----------------------------------------------//
```

```
//Register listeners on the user input components.
//-------------------------------------------------//
chooseButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){
      Color selColor = JColorChooser.showDialog(
                chooseButton,"Choose color",new Color(
                   redInt,greenInt,blueInt));
      if(selColor != null){
        //Don't change the color if the user cancels
        // out.
        redField.setText("" + selColor.getRed());
        greenField.setText(
                            "" + selColor.getGreen());
        blueField.setText("" + selColor.getBlue());
      }//end if

    }//end action performed
  }//end new ActionListener
);//end addActionListener
//-------------------------------------------------//

updateButton.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent e){

      //Get properties of PictureExplorer object.
      xIndex = explorer.getXIndex();
      yIndex = explorer.getYIndex();
      zoomFactor = explorer.getZoomFactor();

      //Save zoom factor as a string.
      String zoomString = "100%";
      if(zoomFactor == 0.25){
        zoomString = "25%";
      }else if(zoomFactor == 0.50){
        zoomString = "50%";
      }else if(zoomFactor == 0.75){
        zoomString = "75%";
      }else if(zoomFactor == 1.0){
        zoomString = "100%";
      }else if(zoomFactor == 1.5){
        zoomString = "150%";
      }else if(zoomFactor == 2.0){
        zoomString = "200%";
      }else if(zoomFactor == 5.0){
        zoomString = "500%";
      }else{
        zoomString = "100%";//in case no match
      }//end else
```

```
        Color newColor = new Color(
                              redInt,greenInt,blueInt);
        pix.getPixel(xIndex,yIndex).setColor(newColor);

        //Dispose of the existing explorer and create a
        // new one.
        explorerFrame.dispose();

        explorer = new PictureExplorer(pix);
        //Get reference to the new frame
        explorerFrame = explorer.getFrame();
        explorerFrame.setDefaultCloseOperation(
                  WindowConstants.DO_NOTHING_ON_CLOSE);

        //Now set the state of the new explorer.
        //Simulate a mouse pressed event in the picture
        // to set the cursor and the text in the
        // coordinate fields.
        explorer.mousePressed(new MouseEvent(
                        new JButton("dummy component"),
                        MouseEvent.MOUSE_PRESSED,
                        (long)0,
                        0,
                        xIndex,
                        yIndex,
                        0,
                        false));

        //Simulate an action event on the zoom menu to
        // set the zoom.
        explorer.actionPerformed(new ActionEvent(
                          explorer,
                          ActionEvent.ACTION_PERFORMED,
                          zoomString));


      }//end actionPerformed
    }//end newActionListener
  );//end addActionListener
  //-------------------------------------------------//

  //Register a document listener on the red text field.
  // This listener will respond when the contents of
  // the text field are modified either by the program
  // or by the user.
  redField.getDocument().addDocumentListener(
    new DocumentListener(){
      public void changedUpdate(DocumentEvent e){
        //Empty method - not needed
      }//end changedUpdate
```

```
    public void removeUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                               redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end removeUpdate

    public void insertUpdate(DocumentEvent e){
      try{
        redInt = Integer.parseInt(
                               redField.getText());
        if((redInt >= 0) && (redInt <= 255)){
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end insertUpdate

  }//end new DocumentListener
);//end addDocumentListener
//---------------------------------------------------//

//Register a document listener on the green text
// field. Essentially the same as the above.
greenField.getDocument().addDocumentListener(
  new DocumentListener(){
    public void changedUpdate(DocumentEvent e){}

    public void removeUpdate(DocumentEvent e){
      try{
        greenInt = Integer.parseInt(
                              greenField.getText());
        if((greenInt >= 0) && (greenInt <= 255))
        {
          paintColorSwatch();
        }//end if
      }catch(Exception ex){
        //do nothing on exception
      }//end catch
    }//end removeUpdate

    public void insertUpdate(DocumentEvent e){
      try{
        greenInt = Integer.parseInt(
                              greenField.getText());
```

```
          if((greenInt >= 0) && (greenInt <= 255))
          {
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-----------------------------------------------//

  //Register a document listener on the blue text
  // field. Essentially the same as the above.
  blueField.getDocument().addDocumentListener(
    new DocumentListener(){
      public void changedUpdate(DocumentEvent e){}

      public void removeUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                              blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end removeUpdate

      public void insertUpdate(DocumentEvent e){
        try{
          blueInt = Integer.parseInt(
                              blueField.getText());
          if((blueInt >= 0) && (blueInt <= 255)){
            paintColorSwatch();
          }//end if
        }catch(Exception ex){
          //do nothing on exception
        }//end catch
      }//end insertUpdate

    }//end new DocumentListener
  );//end addDocumentListener
  //-----------------------------------------------//
}//end constructor
//-----------------------------------------------//

//The purpose of this method is to color a swatch
// located next to the RGB color values.
```

```
  private void paintColorSwatch(){
    colorIndicatorPanel.setBackground(
                      new Color(redInt,greenInt,blueInt));
  }//end paintColorSwatch

  //-----------------------------------------------------//

  //The purpose of this method is to absorb any exceptions
  // that may be thrown by the parseInt method in order
  // to avoid having the program abort. In the event that
  // an exception is thrown, this method simply returns an
  // int value of 0;
  private int goParseInt(String string){
    int result = 0;
    try{
      result = Integer.parseInt(string);
    }catch(Exception e){
      result = 0;
    }//end catch
    return result;
  }//end goParseInt

  //-----------------------------------------------------//


}//end class Prob15Runner
```

Listing 6.133: Complete listing of modified PictureExplorer class.

```
    import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import javax.swing.border.*;

/*
05/16/12 Modified by Baldwin to add getter methods to
 cause the following values to be accessible from
 outside the object:

int xIndex
int yIndex
double zoomFactor
JFrame pictureFrame

Also disabled the call to setDefaultCloseOperation

*/

/**
```

```
 * Displays a picture and lets you explore the picture by displaying the x, y, red,
 * green, and blue values of the pixel at the cursor when you click a mouse button or
 * press and hold a mouse button while moving the cursor.  It also lets you zoom in or
 * out.  You can also type in a x and y value to see the color at that location.
 *
 * Originally created for the Jython Environment for Students (JES).
 * Modified to work with DrJava by Barbara Ericson
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Keith McDermottt, gte047w@cc.gatech.edu
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class PictureExplorer implements MouseMotionListener, ActionListener, MouseListener
{

 // current x and y index
 private int xIndex = 0;
 private int yIndex = 0;

 //Main gui variables
 private JFrame pictureFrame;
 private JScrollPane scrollPane;

 //information bar variables
 private JLabel xLabel;
 private JButton xPrevButton;
 private JButton yPrevButton;
 private JButton xNextButton;
 private JButton yNextButton;
 private JLabel yLabel;
 private JTextField xValue;
 private JTextField yValue;
 private JLabel rValue;
 private JLabel gValue;
 private JLabel bValue;
 private JLabel colorLabel;
 private JPanel colorPanel;

 // menu components
 private JMenuBar menuBar;
 private JMenu zoomMenu;
 private JMenuItem twentyFive;
 private JMenuItem fifty;
 private JMenuItem seventyFive;
 private JMenuItem hundred;
 private JMenuItem hundredFifty;
 private JMenuItem twoHundred;
 private JMenuItem fiveHundred;

 /** The picture being explored */
 private DigitalPicture picture;
```

```
 /** The image icon used to display the picture */
 private ImageIcon scrollImageIcon;

 /** The image display */
 private ImageDisplay imageDisplay;

 /** the zoom factor (amount to zoom) */
 private double zoomFactor;

 /** the number system to use, 0 means starting at 0, 1 means starting at 1 */
 private int numberBase=0;

 /**
  * Public constructor
  * @param picture the picture to explore
  */
 public PictureExplorer(DigitalPicture picture)
 {
   // set the fields
   this.picture=picture;
   zoomFactor=1;

   // create the window and set things up
   createWindow();
 }

//===Methods added by Baldwin on 05/16/12==============//

  //Method to get the xIndex value.
  public int getXIndex(){
    return xIndex;
  }//end getXIndex

  //Method to get the yIndex value.
  public int getYIndex(){
    return yIndex;
  }//end getYIndex

  //Method to get the zoomFactor value.
  public double getZoomFactor(){
    return zoomFactor;
  }//end getZoomFactor

  //Method to get a reference to the frame
  public JFrame getFrame(){
    return pictureFrame;
  }//end getFrame()

//===End methods added by Baldwin on 05/16/12===========//
```

```
/**
 * Changes the number system to start at one
 */
public void changeToBaseOne()
{
  numberBase=1;
}

/**
 * Set the title of the frame
 *@param title the title to use in the JFrame
 */
public void setTitle(String title)
{
  pictureFrame.setTitle(title);
}

/**
 * Method to create and initialize the picture frame
 */
private void createAndInitPictureFrame()
{
  pictureFrame = new JFrame(); // create the JFrame
  pictureFrame.setResizable(true);  // allow the user to resize it
  pictureFrame.getContentPane().setLayout(new BorderLayout()); // use border layout

  //Disabled by Baldwin on 05/16/12
  //pictureFrame.setDefaultCloseOperation(
  // JFrame.DISPOSE_ON_CLOSE);

  pictureFrame.setTitle(picture.getTitle());
  PictureExplorerFocusTraversalPolicy newPolicy = new PictureExplorerFocusTraversalPolicy();
  pictureFrame.setFocusTraversalPolicy(newPolicy);

}

/**
 * Method to create the menu bar, menus, and menu items
 */
private void setUpMenuBar()
{
  //create menu
  menuBar = new JMenuBar();
  zoomMenu = new JMenu("Zoom");
  twentyFive = new JMenuItem("25%");
  fifty = new JMenuItem("50%");
```

```
  seventyFive = new JMenuItem("75%");
  hundred = new JMenuItem("100%");
  hundred.setEnabled(false);
  hundredFifty = new JMenuItem("150%");
  twoHundred = new JMenuItem("200%");
  fiveHundred = new JMenuItem("500%");

  // add the action listeners
  twentyFive.addActionListener(this);
  fifty.addActionListener(this);
  seventyFive.addActionListener(this);
  hundred.addActionListener(this);
  hundredFifty.addActionListener(this);
  twoHundred.addActionListener(this);
  fiveHundred.addActionListener(this);

  // add the menu items to the menus
  zoomMenu.add(twentyFive);
  zoomMenu.add(fifty);
  zoomMenu.add(seventyFive);
  zoomMenu.add(hundred);
  zoomMenu.add(hundredFifty);
  zoomMenu.add(twoHundred);
  zoomMenu.add(fiveHundred);
  menuBar.add(zoomMenu);

  // set the menu bar to this menu
  pictureFrame.setJMenuBar(menuBar);
}

/**
 * Create and initialize the scrolling image
 */
private void createAndInitScrollingImage()
{
  scrollPane = new JScrollPane();

  BufferedImage bimg = picture.getBufferedImage();
  imageDisplay = new ImageDisplay(bimg);
  imageDisplay.addMouseMotionListener(this);
  imageDisplay.addMouseListener(this);
  imageDisplay.setToolTipText("Click a mouse button on a pixel to see the pixel information");
  scrollPane.setViewportView(imageDisplay);
  pictureFrame.getContentPane().add(scrollPane, BorderLayout.CENTER);
}

/**
 * Creates the JFrame and sets everything up
 */
private void createWindow()
{
```

```
  // create the picture frame and initialize it
  createAndInitPictureFrame();

  // set up the menu bar
  setUpMenuBar();

  //create the information panel
  createInfoPanel();

  //creates the scrollpane for the picture
  createAndInitScrollingImage();

  // show the picture in the frame at the size it needs to be
  pictureFrame.pack();
  pictureFrame.setVisible(true);
}

/**
 * Method to set up the next and previous buttons for the
 * pixel location information
 */
private void setUpNextAndPreviousButtons()
{
  // create the image icons for the buttons
  Icon prevIcon = new ImageIcon(SoundExplorer.class.getResource("leftArrow.gif"),
                                "previous index");
  Icon nextIcon = new ImageIcon(SoundExplorer.class.getResource("rightArrow.gif"),
                                "next index");
  // create the arrow buttons
  xPrevButton = new JButton(prevIcon);
  xNextButton = new JButton(nextIcon);
  yPrevButton = new JButton(prevIcon);
  yNextButton = new JButton(nextIcon);

  // set the tool tip text
  xNextButton.setToolTipText("Click to go to the next x value");
  xPrevButton.setToolTipText("Click to go to the previous x value");
  yNextButton.setToolTipText("Click to go to the next y value");
  yPrevButton.setToolTipText("Click to go to the previous y value");

  // set the sizes of the buttons
  int prevWidth = prevIcon.getIconWidth() + 2;
  int nextWidth = nextIcon.getIconWidth() + 2;
  int prevHeight = prevIcon.getIconHeight() + 2;
  int nextHeight = nextIcon.getIconHeight() + 2;
  Dimension prevDimension = new Dimension(prevWidth,prevHeight);
  Dimension nextDimension = new Dimension(nextWidth, nextHeight);
  xPrevButton.setPreferredSize(prevDimension);
  yPrevButton.setPreferredSize(prevDimension);
  xNextButton.setPreferredSize(nextDimension);
  yNextButton.setPreferredSize(nextDimension);
```

```
  // handle previous x button press
  xPrevButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      xIndex--;
      if (xIndex < 0)
        xIndex = 0;
      displayPixelInformation(xIndex,yIndex);
    }
  });

  // handle previous y button press
  yPrevButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      yIndex--;
      if (yIndex < 0)
        yIndex = 0;
      displayPixelInformation(xIndex,yIndex);
    }
  });

  // handle next x button press
  xNextButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      xIndex++;
      if (xIndex >= picture.getWidth())
        xIndex = picture.getWidth() - 1;
      displayPixelInformation(xIndex,yIndex);
    }
  });

  // handle next y button press
  yNextButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      yIndex++;
      if (yIndex >= picture.getHeight())
        yIndex = picture.getHeight() - 1;
      displayPixelInformation(xIndex,yIndex);
    }
  });
}

/**
 * Create the pixel location panel
 * @param labelFont the font for the labels
 * @return the location panel
 */
public JPanel createLocationPanel(Font labelFont) {

  // create a location panel
  JPanel locationPanel = new JPanel();
```

```
  locationPanel.setLayout(new FlowLayout());
  Box hBox = Box.createHorizontalBox();

  // create the labels
  xLabel = new JLabel("X:");
  yLabel = new JLabel("Y:");

  // create the text fields
  xValue = new JTextField(Integer.toString(xIndex + numberBase),6);
  xValue.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      displayPixelInformation(xValue.getText(),yValue.getText());
    }
  });
  yValue = new JTextField(Integer.toString(yIndex + numberBase),6);
  yValue.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     displayPixelInformation(xValue.getText(),yValue.getText());
    }
  });

  // set up the next and previous buttons
  setUpNextAndPreviousButtons();

  // set up the font for the labels
  xLabel.setFont(labelFont);
  yLabel.setFont(labelFont);
  xValue.setFont(labelFont);
  yValue.setFont(labelFont);

  // add the items to the vertical box and the box to the panel
  hBox.add(Box.createHorizontalGlue());
  hBox.add(xLabel);
  hBox.add(xPrevButton);
  hBox.add(xValue);
  hBox.add(xNextButton);
  hBox.add(Box.createHorizontalStrut(10));
  hBox.add(yLabel);
  hBox.add(yPrevButton);
  hBox.add(yValue);
  hBox.add(yNextButton);
  locationPanel.add(hBox);
  hBox.add(Box.createHorizontalGlue());

  return locationPanel;
}

/**
 * Create the color information panel
 * @param labelFont the font to use for labels
 * @return the color information panel
```

```
 */
private JPanel createColorInfoPanel(Font labelFont)
{
  // create a color info panel
  JPanel colorInfoPanel = new JPanel();
  colorInfoPanel.setLayout(new FlowLayout());

  // get the pixel at the x and y
  Pixel pixel = new Pixel(picture,xIndex,yIndex);

  // create the labels
  rValue = new JLabel("R: " + pixel.getRed());
  gValue = new JLabel("G: " + pixel.getGreen());
  bValue = new JLabel("B: " + pixel.getBlue());

  // create the sample color panel and label
  colorLabel = new JLabel("Color at location: ");
  colorPanel = new JPanel();
  colorPanel.setBorder(new LineBorder(Color.black,1));

  // set the color sample to the pixel color
  colorPanel.setBackground(pixel.getColor());

  // set the font
  rValue.setFont(labelFont);
  gValue.setFont(labelFont);
  bValue.setFont(labelFont);
  colorLabel.setFont(labelFont);
  colorPanel.setPreferredSize(new Dimension(25,25));

  // add items to the color information panel
  colorInfoPanel.add(rValue);
  colorInfoPanel.add(gValue);
  colorInfoPanel.add(bValue);
  colorInfoPanel.add(colorLabel);
  colorInfoPanel.add(colorPanel);

  return colorInfoPanel;
}

/**
 * Creates the North JPanel with all the pixel location
 * and color information
 */
private void createInfoPanel()
{
  // create the info panel and set the layout
  JPanel infoPanel = new JPanel();
  infoPanel.setLayout(new BorderLayout());

  // create the font
```

```
  Font largerFont = new Font(infoPanel.getFont().getName(),
                             infoPanel.getFont().getStyle(),14);

  // create the pixel location panel
  JPanel locationPanel = createLocationPanel(largerFont);

  // create the color informaiton panel
  JPanel colorInfoPanel = createColorInfoPanel(largerFont);

  // add the panels to the info panel
  infoPanel.add(BorderLayout.NORTH,locationPanel);
  infoPanel.add(BorderLayout.SOUTH,colorInfoPanel);

  // add the info panel
  pictureFrame.getContentPane().add(BorderLayout.NORTH,infoPanel);
}

/**
 * Method to check that the current position is in the viewing area and if
 * not scroll to center the current position if possible
 */
public void checkScroll()
{
  // get the x and y position in pixels
  int xPos = (int) (xIndex * zoomFactor);
  int yPos = (int) (yIndex * zoomFactor);

  // only do this if the image is larger than normal
  if (zoomFactor > 1) {

    // get the rectangle that defines the current view
    JViewport viewport = scrollPane.getViewport();
    Rectangle rect = viewport.getViewRect();
    int rectMinX = (int) rect.getX();
    int rectWidth = (int) rect.getWidth();
    int rectMaxX = rectMinX + rectWidth - 1;
    int rectMinY = (int) rect.getY();
    int rectHeight = (int) rect.getHeight();
    int rectMaxY = rectMinY + rectHeight - 1;

    // get the maximum possible x and y index
    int maxIndexX = (int) (picture.getWidth() * zoomFactor) - rectWidth - 1;
    int maxIndexY = (int) (picture.getHeight() * zoomFactor) - rectHeight - 1;

    // calculate how to position the current position in the middle of the viewing
    // area
    int viewX = xPos - (int) (rectWidth / 2);
    int viewY = yPos - (int) (rectHeight / 2);

    // reposition the viewX and viewY if outside allowed values
    if (viewX < 0)
```

```
      viewX = 0;
    else if (viewX > maxIndexX)
      viewX = maxIndexX;
    if (viewY < 0)
      viewY = 0;
    else if (viewY > maxIndexY)
      viewY = maxIndexY;

    // move the viewport upper left point
    viewport.scrollRectToVisible(new Rectangle(viewX,viewY,rectWidth,rectHeight));
  }
}

/**
 * Zooms in the on picture by scaling the image.
 * It is extremely memory intensive.
 * @param factor the amount to zoom by
 */
public void zoom(double factor)
{
  // save the current zoom factor
  zoomFactor = factor;

  // calculate the new width and height and get an image that size
  int width = (int) (picture.getWidth()*zoomFactor);
  int height = (int) (picture.getHeight()*zoomFactor);
  BufferedImage bimg = picture.getBufferedImage();

  // set the scroll image icon to the new image
  imageDisplay.setImage(bimg.getScaledInstance(width, height, Image.SCALE_DEFAULT));
  imageDisplay.setCurrentX((int) (xIndex * zoomFactor));
  imageDisplay.setCurrentY((int) (yIndex * zoomFactor));
  imageDisplay.revalidate();
  checkScroll();  // check if need to reposition scroll
}

/**
 * Repaints the image on the scrollpane.
 */
public void repaint()
{
  pictureFrame.repaint();
}

//**************************************//
//               Event Listeners          //
//**************************************//

/**
 * Called when the mouse is dragged (button held down and moved)
 * @param e the mouse event
```

```
 */
public void mouseDragged(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method to check if the given x and y are in the picture
 * @param x the horiztonal value
 * @param y the vertical value
 * @return true if the x and y are in the picture and false otherwise
 */
private boolean isLocationInPicture(int x, int y)
{
  boolean result = false; // the default is false
  if (x >= 0 && x < picture.getWidth() &&
      y >= 0 && y < picture.getHeight())
    result = true;

  return result;
}

/**
 * Method to display the pixel information from the passed x and y but
 * also converts x and y from strings
 * @param xString the x value as a string from the user
 * @param yString the y value as a string from the user
 */
public void displayPixelInformation(String xString, String yString)
{
  int x = -1;
  int y = -1;
  try {
    x = Integer.parseInt(xString);
    x = x - numberBase;
    y = Integer.parseInt(yString);
    y = y - numberBase;
  } catch (Exception ex) {
  }

  if (x >= 0 && y >= 0) {
    displayPixelInformation(x,y);
  }
}

/**
 * Method to display pixel information for the passed x and y
 * @param pictureX the x value in the picture
 * @param pictureY the y value in the picture
 */
private void displayPixelInformation(int pictureX, int pictureY)
```

```
{
  // check that this x and y is in range
  if (isLocationInPicture(pictureX, pictureY))
  {
    // save the current x and y index
    xIndex = pictureX;
    yIndex = pictureY;

    // get the pixel at the x and y
    Pixel pixel = new Pixel(picture,xIndex,yIndex);

    // set the values based on the pixel
    xValue.setText(Integer.toString(xIndex  + numberBase));
    yValue.setText(Integer.toString(yIndex + numberBase));
    rValue.setText("R: " + pixel.getRed());
    gValue.setText("G: " + pixel.getGreen());
    bValue.setText("B: " + pixel.getBlue());
    colorPanel.setBackground(new Color(pixel.getRed(), pixel.getGreen(), pixel.getBlue()));

  }
  else
  {
    clearInformation();
  }

  // notify the image display of the current x and y
  imageDisplay.setCurrentX((int) (xIndex * zoomFactor));
  imageDisplay.setCurrentY((int) (yIndex * zoomFactor));
}

/**
 * Method to display pixel information based on a mouse event
 * @param e a mouse event
 */
private void displayPixelInformation(MouseEvent e)
{

  // get the cursor x and y
  int cursorX = e.getX();
  int cursorY = e.getY();

  // get the x and y in the original (not scaled image)
  int pictureX = (int) (cursorX / zoomFactor + numberBase);
  int pictureY = (int) (cursorY / zoomFactor + numberBase);

  // display the information for this x and y
  displayPixelInformation(pictureX,pictureY);

}

/**
```

```
 * Method to clear the labels and current color and reset the
 * current index to -1
 */
private void clearInformation()
{
  xValue.setText("N/A");
  yValue.setText("N/A");
  rValue.setText("R: N/A");
  gValue.setText("G: N/A");
  bValue.setText("B: N/A");
  colorPanel.setBackground(Color.black);
  xIndex = -1;
  yIndex = -1;
}

/**
 * Method called when the mouse is moved with no buttons down
 * @param e the mouse event
 */
public void mouseMoved(MouseEvent e)
{}

/**
 * Method called when the mouse is clicked
 * @param e the mouse event
 */
public void mouseClicked(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method called when the mouse button is pushed down
 * @param e the mouse event
 */
public void mousePressed(MouseEvent e)
{
  displayPixelInformation(e);
}

/**
 * Method called when the mouse button is released
 * @param e the mouse event
 */
public void mouseReleased(MouseEvent e)
{
}

/**
 * Method called when the component is entered (mouse moves over it)
 * @param e the mouse event
```

```
 */
public void mouseEntered(MouseEvent e)
{
}

/**
 * Method called when the mouse moves over the component
 * @param e the mouse event
 */
public void mouseExited(MouseEvent e)
{
}

/**
 * Method to enable all menu commands
 */
private void enableZoomItems()
{
  twentyFive.setEnabled(true);
  fifty.setEnabled(true);
  seventyFive.setEnabled(true);
  hundred.setEnabled(true);
  hundredFifty.setEnabled(true);
  twoHundred.setEnabled(true);
  fiveHundred.setEnabled(true);
}

/**
 * Controls the zoom menu bar
 *
 * @param a the ActionEvent
 */
public void actionPerformed(ActionEvent a)
{

  if(a.getActionCommand().equals("Update"))
  {
    this.repaint();
  }

  if(a.getActionCommand().equals("25%"))
  {
    this.zoom(.25);
    enableZoomItems();
    twentyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("50%"))
  {
    this.zoom(.50);
    enableZoomItems();
```

```
    fifty.setEnabled(false);
  }

  if(a.getActionCommand().equals("75%"))
  {
    this.zoom(.75);
    enableZoomItems();
    seventyFive.setEnabled(false);
  }

  if(a.getActionCommand().equals("100%"))
  {
    this.zoom(1.0);
    enableZoomItems();
    hundred.setEnabled(false);
  }

  if(a.getActionCommand().equals("150%"))
  {
    this.zoom(1.5);
    enableZoomItems();
    hundredFifty.setEnabled(false);
  }

  if(a.getActionCommand().equals("200%"))
  {
    this.zoom(2.0);
    enableZoomItems();
    twoHundred.setEnabled(false);
  }

  if(a.getActionCommand().equals("500%"))
  {
    this.zoom(5.0);
    enableZoomItems();
    fiveHundred.setEnabled(false);
  }
}

/**
 * Test Main.  It will ask you to pick a file and then show it
 */
public static void main( String args[])
{
  Picture p = new Picture(FileChooser.pickAFile());
  PictureExplorer test = new PictureExplorer(p);

}

/**
 * Class for establishing the focus for the textfields
```

```
 */
private class PictureExplorerFocusTraversalPolicy
               extends FocusTraversalPolicy {

       /**
        * Method to get the next component for focus
        */
       public Component getComponentAfter(Container focusCycleRoot,
                                          Component aComponent) {
           if (aComponent.equals(xValue))
             return yValue;
           else
             return xValue;
       }

       /**
        * Method to get the previous component for focus
        */
        public Component getComponentBefore(Container focusCycleRoot,
                                    Component aComponent) {
           if (aComponent.equals(xValue))
             return yValue;
           else
             return xValue;
        }

        public Component getDefaultComponent(Container focusCycleRoot) {
           return xValue;
       }

       public Component getLastComponent(Container focusCycleRoot) {
           return yValue;
       }

       public Component getFirstComponent(Container focusCycleRoot) {
           return xValue;
       }
   }

}
```

-end-

6.4 Practice Tests

6.4.1 Java OOP: ITSE 2317 Practice Test 1[73]

6.4.1.1 ITSE2317 - Java Programming (Intermediate) - Practice Test 1

6.4.1.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [74] .

6.4.1.1.2 Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

6.4.1.1.3 Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [75] .

6.4.1.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

6.4.1.1.5 New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX when such class definitions are provided.

[73] This content is available online at <http://cnx.org/content/m44264/1.2/>.
[74] http://cnx.org/content/m44148/latest/
[75] http://cnx.org/content/m44148/latest/

6.4.1.1.6 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

6.4.1.1.7 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [76] .

6.4.1.1.8 Program Specifications

6.4.1.1.8.1 Program 1

Listing 6.134: Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image file named Prob01.jpg to produce the graphic output image shown in Figure 1 (p. 1675) below.

Click here [77] to download a zip file containing the required image file along with the source code for a solution.

Contrary to the general instructions given above, you may not define any new classes to cause your program to behave as required.

You must copy and modify (if necessary) the media classes named World.java, Turtle.java, and SimpleTurtle.java to cause your program to produce the required output. Don't forget to compile these classes after you modify them.

In addition to the output image, your program must produce the following output on the command- line screen, and must substitute your name for mine wherever my name appears both in the image and on the command-line screen:

```
    Dick Baldwin
Picture, filename Prob01.jpg height 274 width 365
Dick Baldwin
Dick Baldwin
Dick Baldwin
Dick Baldwin
**********************************************************/
```

[76] http://cnx.org/content/m44148/latest/
[77] http://cnx.org/content/m44264/latest/Prob01solution.zip

```
public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    World mars = new World(200,250);
    Turtle joe = new Turtle(mars);
    joe.forward();
    Turtle bill = new Turtle(mars);
    bill.moveTo(50,125);
    Turtle sue = new Turtle(mars);
    sue.moveTo(150,125);
    Turtle tom = new Turtle(mars);
    tom.moveTo(100,225);
  }//end main method
}//end class Prob01
//End program specifications.
```

Required output images for Prob01.

Figure 6.46: Required output images for Prob01.

6.4.1.1.8.2 Program 2

Listing 6.135: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses the class definition shown below and Ericson's media library to produce the graphic output image shown in Figure 2 (p. 1677) below.

Click here [78] to download a zip file containing source code for a solution.

Contrary to the general instructions given above, you may not define any new classes to cause your program to behave as required.

You must copy and modify (if necessary) the media classes named Turtle.java, and SimpleTurtle.java to cause your program to produce the required output. Don't forget to compile these classes after you modify them.

In addition to the output image, your program must produce the following output on the command- line screen, and must substitute your name for mine wherever my name appears both in the image and on the command-line screen:

```
    Dick Baldwin
My name is Joe the turtle.
*************************************************************/
import java.awt.Color;

public class Prob02{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    World mars = new World(200,300);
    Turtle joe = new Turtle(mars,"Joe");
    joe.moveTo(20,280);
    joe.setInfoColor(Color.WHITE);
    joe.setShowInfo(true);
    System.out.println(joe);
  }//end main method
}//end class Prob02
//End program specifications.
```

[78] http://cnx.org/content/m44264/latest/Prob02solution.zip

Required output images for Prob02.

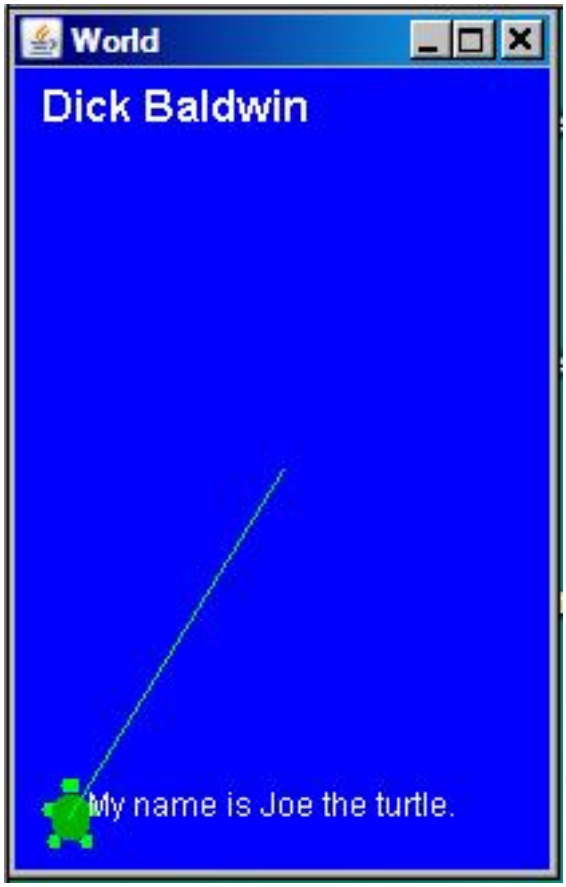


Figure 6.47: Required output images for Prob02.

6.4.1.1.8.3 Program 3

Listing 6.136: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library to produce the graphic output images shown in Figure 3 (p. 1678) and Figure 4 (p. 1679) below.

Click here [79] to download a zip file containing the source code for a solution.

The image shown in Figure 3 (p. 1678) is the image that appears on the screen when the program starts running. The image shown in Figure 4 (p. 1679) is what you should see when you click the button at the bottom of the world.

You must copy and modify (if necessary) the media class named World.java, to cause your program to produce the required output with the required behavior. Don't forget to compile that class after you modify it.

[79] http://cnx.org/content/m44264/latest/Prob03solution.zip

This program adds a JButton object to the SOUTH location of the World object as shown in Figure 3 (p. 1678) .

The program initially displays an empty white world. When the user clicks the button, the world's background color changes to blue, a turtle appears in the center of the World, and the student's name appears near the top of the world.

The first of two required output images for Prob03.

Figure 6.48: The first of two required output images for Prob03.

The second of two required output images for Prob03.

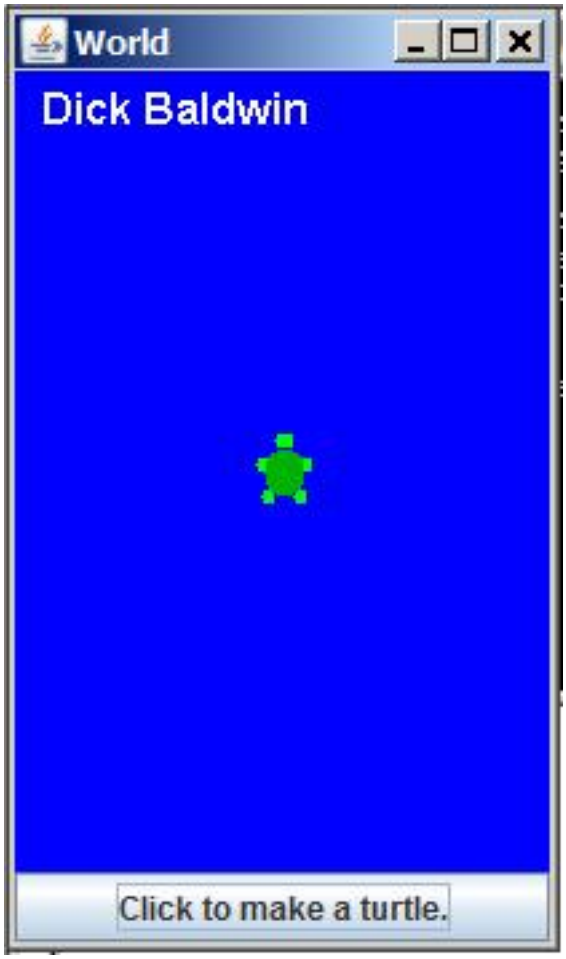


Figure 6.49: The second of two required output images for Prob03.

6.4.1.1.8.4 Program 4

Listing 6.137: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses Ericson's media library to produce the graphic output images shown in Figure 5 (p. 1680) , Figure 6 (p. 1681) , and Figure 7 (p. 1682) below.

Click here [80] to download a zip file containing the source code for a solution.

Figure 5 (p. 1680) shows the image that appears on the screen when the program starts running. Figure 6 (p. 1681) shows what you should see after you have clicked the button at the bottom of the world one time. Figure 6 (p. 1681) shows what you should see after you have clicked the button sixteen times.

[80] http://cnx.org/content/m44264/latest/Prob04solution.zip

You must copy and modify the media classes named World.java and SimplePicture.java, to cause your program to produce the required output with the required behavior. Don't forget to compile those classes after you modify them.

This program adds a JButton object to the SOUTH location of the World object as shown in Figure 5 (p. 1680) .

The following description is intended to guide you in writing your program. However, you must run my version of the program and replicate it exactly. I recommend that you run the two programs side-by-side and compare their appearance and behavior each time you click both programs.

The program initially displays an empty white world with a button at the bottom. When the user clicks the button, the world's background color changes to green, a turtle appears in the bottom right of the World, and the student's name appears near the top of the world in blue. The turtle has a blue body and a red shell.

When you click the button again, the background changes to yellow, the student's name changes to red, and the turtle changes to a red body with a blue shell. The turtle turns 90 degrees left and moves forward 100 pixels plus the value of a click counter. As a result, the turtle leaves a blue trail.

On the next click, the colors revert to the same as before, the turtle turns 90 degrees left and moves forward 100 pixels plus the value of the click counter leaving a red trail.

This cycle repeats on each click with the turtle's trail drawing a square spiral of increasing size with red lines on the top and bottom of the spiral and blue lines on the right and left of the spiral.

Required output image for Prob04.

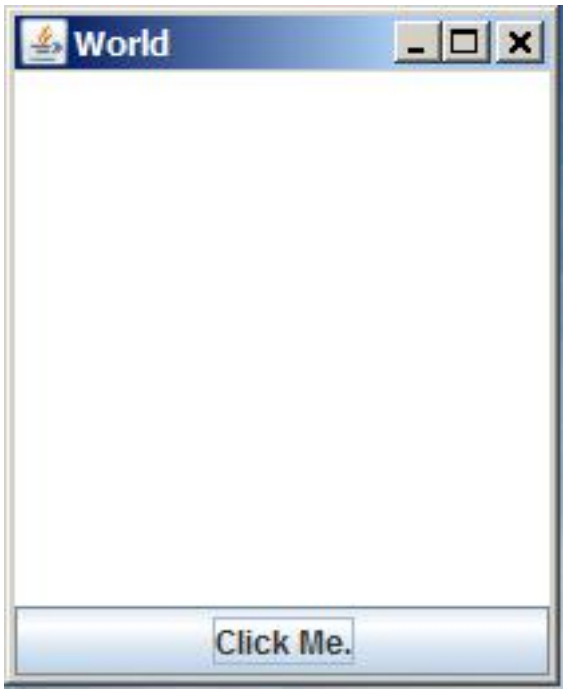


Figure 6.50: Required output image for Prob04.

Required output image for Prob04.

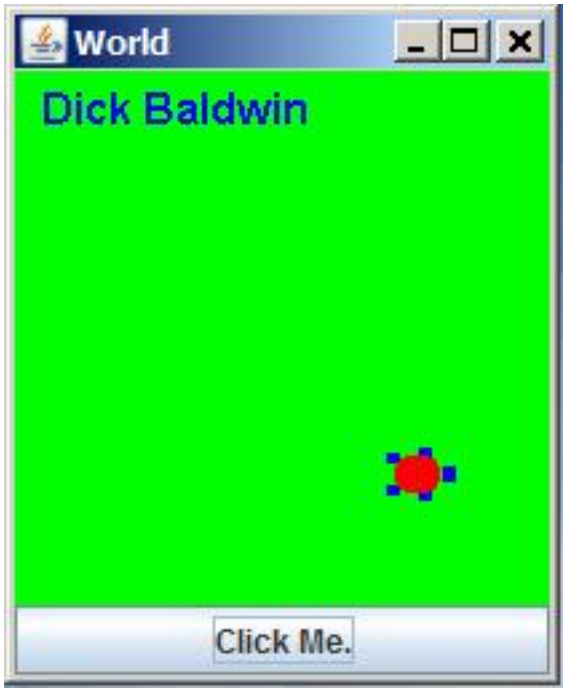


Figure 6.51: Required output image for Prob04.

Required output image for Prob04.

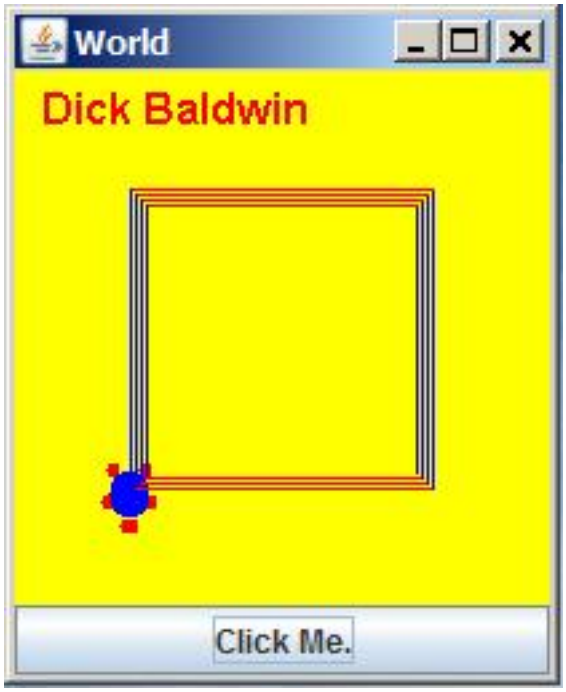


Figure 6.52: Required output image for Prob04.

6.4.1.1.8.5 Program 5

Listing 6.138: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library to produce the graphic output image shown in Figure 8 (p. 1683) , Figure 9 (p. 1684) , and Figure 10 (p. 1685) below.

Click here [81] to download a zip file containing the source code for a solution.

Figure 8 (p. 1683) shows the image that appears on the screen when the program starts running. Figure 9 (p. 1684) shows what you should see after you have entered numeric values into the angle and distance fields and have clicked the Move button. Figure 10 (p. 1685) is similar to what you should see after doing the above several times for different numeric values.

You must copy and modify the media class named World.java to cause your program to produce the required output with the required behavior. Don't forget to compile World.java after you modify it.

This program adds two buttons, two labels, and two text fields to form a GUI at the bottom of the World object.

If you enter numeric values into the angle and distance fields and then click the Move button, the turtle will turn by that angle in degrees and move by that distance in pixels.

[81] http://cnx.org/content/m44264/latest/Prob05solution.zip

The program must terminate and return control to the operating system when you click the Quit button.

Note that the GUI at the bottom of the World object is comprised of AWT components instead of Swing components.

Required output image for Prob05.

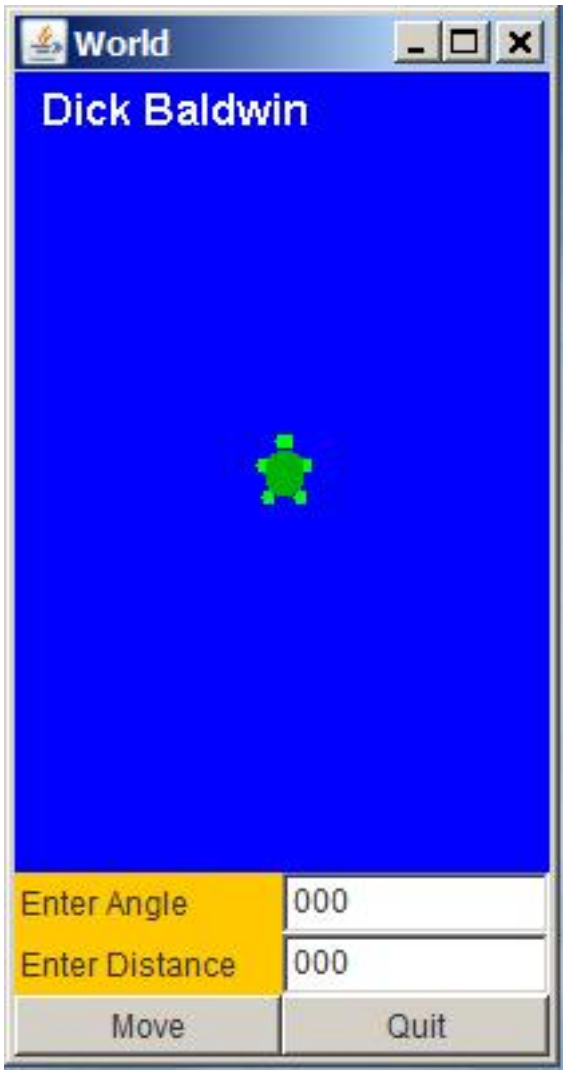


Figure 6.53: Required output image for Prob05.

Required output image for Prob05.

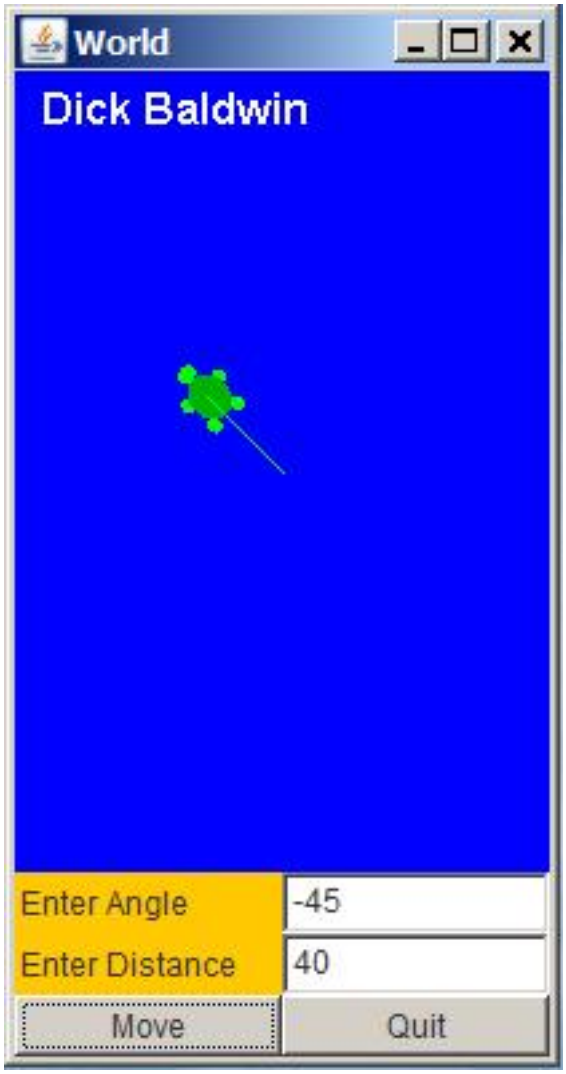


Figure 6.54: Required output image for Prob05.

Required output image for Prob05.

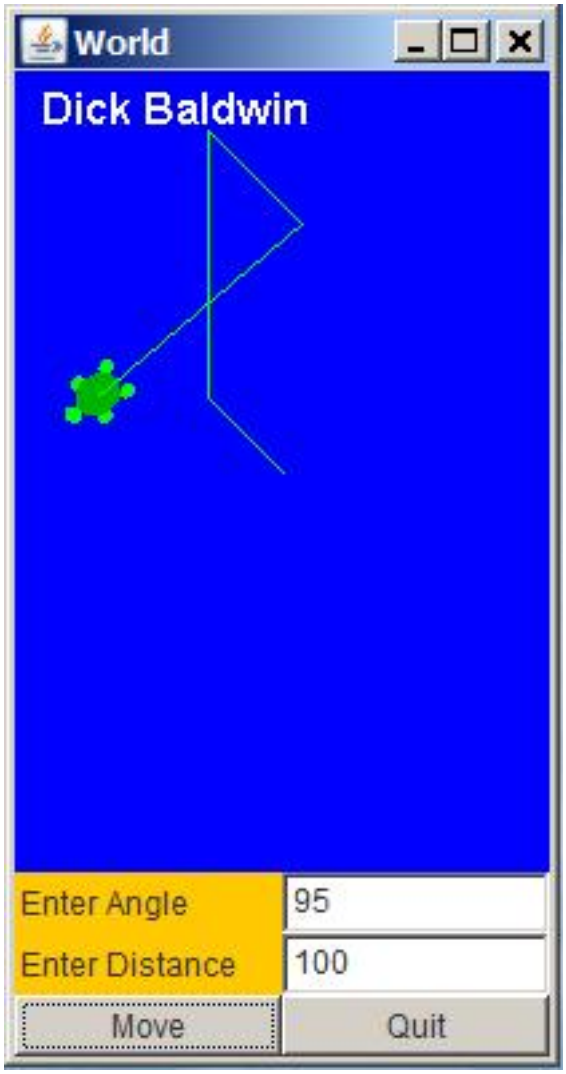


Figure 6.55: Required output image for Prob05.

6.4.1.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2317 Practice Test 1
- File: PracticeTest01.htm
- Published: August 9, 2012

- Revised: –

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.4.2 Java OOP: ITSE 2317 Practice Test 2[82]

6.4.2.1 ITSE2317 - Java Programming (Intermediate) - Practice Test 2

6.4.2.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [83] .

6.4.2.1.2 Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

6.4.2.1.3 Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [84] .

6.4.2.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

6.4.2.1.5 New Classes

You may define new classes and add import directives as needed to cause your programs to behave as required, but you may not modify the class definitions for the given classes named ProbXX when such class definitions are provided.

[82] This content is available online at <http://cnx.org/content/m44265/1.3/>.
[83] http://cnx.org/content/m44148/latest/
[84] http://cnx.org/content/m44148/latest/

6.4.2.1.6 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

6.4.2.1.7 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [85] .

6.4.2.1.8 Program Specifications

6.4.2.1.8.1 Program 1

Listing 6.139: Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses the class definition shown below and Ericson's media library along with the image files named Prob01a.jpg and Prob01b.jpg to produce the graphic output images shown in Figure 1 (p. 1690) , Figure 2 (p. 1691) , and Figure 3 (p. 1692) below.

Click here [86] to download a zip file containing the required image files along with the source code for a solution.

Just in case you haven't noticed it, the image in Figure 3 (p. 1692) contains a partially transparent image of a butterfly superimposed and centered on the beach image.

In order to write this program, you must modify the class from Ericson's media library named SimplePicture. Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through. The degree of transparency can range from being completely transparent at one extreme to being totally opaque at the other extreme. In this case, the butterfly image is about 37-percent opaque. Don't forget to compile the SimplePicture class after you modify it.

You will probably need to do some outside research in order to write this program. For example, you will need to learn about the following topics and probably some other topics as well:

- Alpha transparency
- BufferedImage objects of TYPE_INT_ARGB
- The representation of a pixel as type int.
- Bit manipulation of pixels.

[85] http://cnx.org/content/m44148/latest/
[86] http://cnx.org/content/m44265/latest/Prob01solution.zip

- The drawImage method of the Graphics class.

In addition to the output images described above, your program must produce the following output on the command-line screen, and must substitute your name for mine wherever my name appears both in the images and on the command-line screen:

```
    Dick Baldwin.
Dick Baldwin
Picture, filename Prob01a.jpg height 118 width 100
Picture, filename Prob01b.jpg height 240 width 320
Picture, filename None height 101 width 77
*******************************************************/
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    Picture[] pictures = new Prob01Runner().run();
    System.out.println(pictures[0]);
    System.out.println(pictures[1]);
    System.out.println(pictures[2]);
  }//end main method
}//end class Prob01
//End program specifications.
```

The first of three required output images for Prob01.

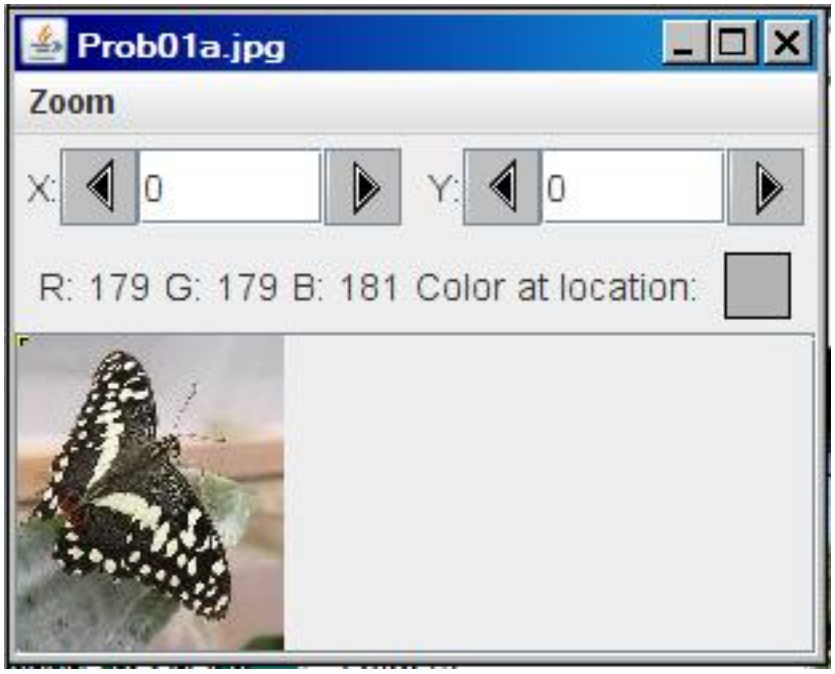


Figure 6.56: The first of three required output images for Prob01.

The second of three required output images for Prob01.

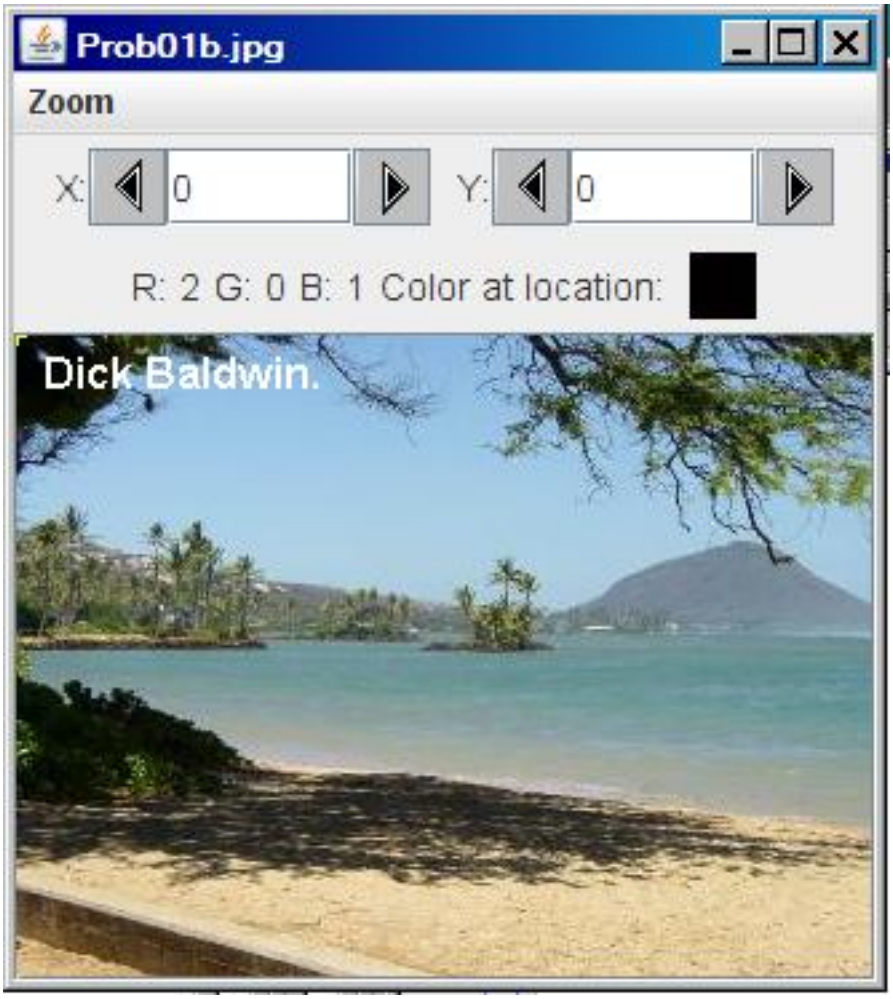


Figure 6.57: The second of three required output images for Prob01.

The third of three required output images for Prob01.

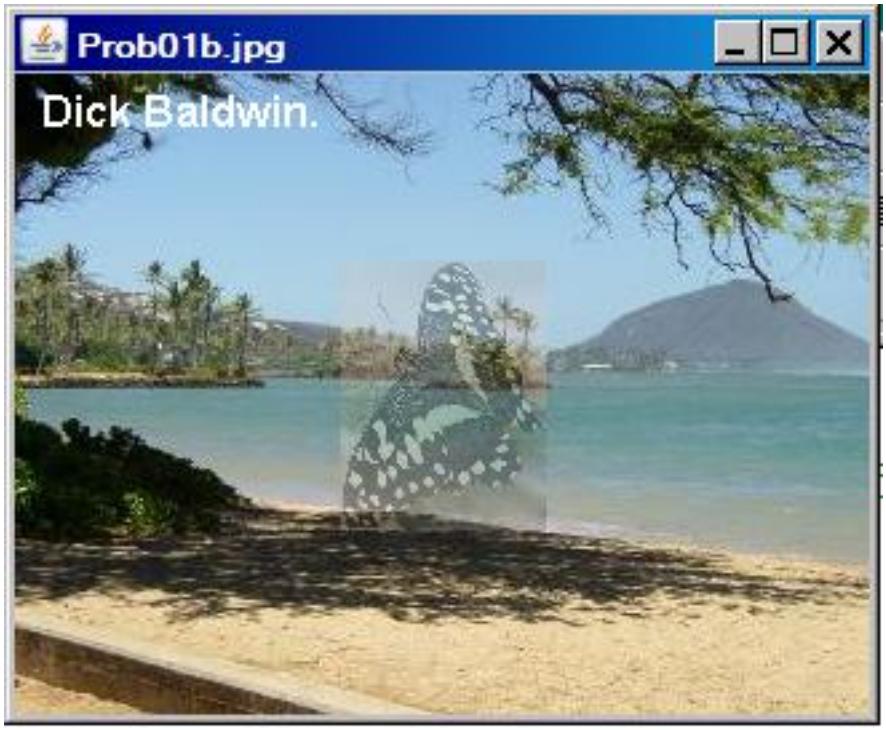


Figure 6.58: The third of three required output images for Prob01.

6.4.2.1.8.2 Program 2

Listing 6.140: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses Ericson's media library along with the image files named Prob02a.jpg and Prob02b.jpg to produce the graphic output images shown in Figure 4 (p. 1693) below.

Click here [87] to download a zip file containing the required image files along with the source code for a solution.

The top image shown in Figure 4 (p. 1693) is a beach scene with a partially opaque butterfly superimposed on the beach scene. The bottom image is a slider that is used to control the percent opacity of the butterfly image.

At startup, the slider is positioned at the 50-percent mark and the opacity of the butterfly is 50 percent.

As you move the slider to the right, the butterfly becomes more opaque, becoming totally opaque when the slider is positioned at 100 percent. As you move the slider to the left, the butterfly becomes less opaque, becoming totally transparent when the slider is positioned at 0 percent.

[87] http://cnx.org/content/m44265/latest/Prob02solution.zip

In order to write this program, you must modify the class from Ericson's media library named SimplePicture. Your modifications must make it possible for you to display a partially transparent image on top of another image with the background image showing through.

Your modification must also make it possible to display your name in the dark blue banner at the top of the image of the beach scene.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

In order to improve the responsiveness and memory utilization of the program, you should instantiate all of the Picture objects that the program needs at startup, and should not instantiate additional Picture objects when handling events fired by the slider.

Required output images for Prob02.

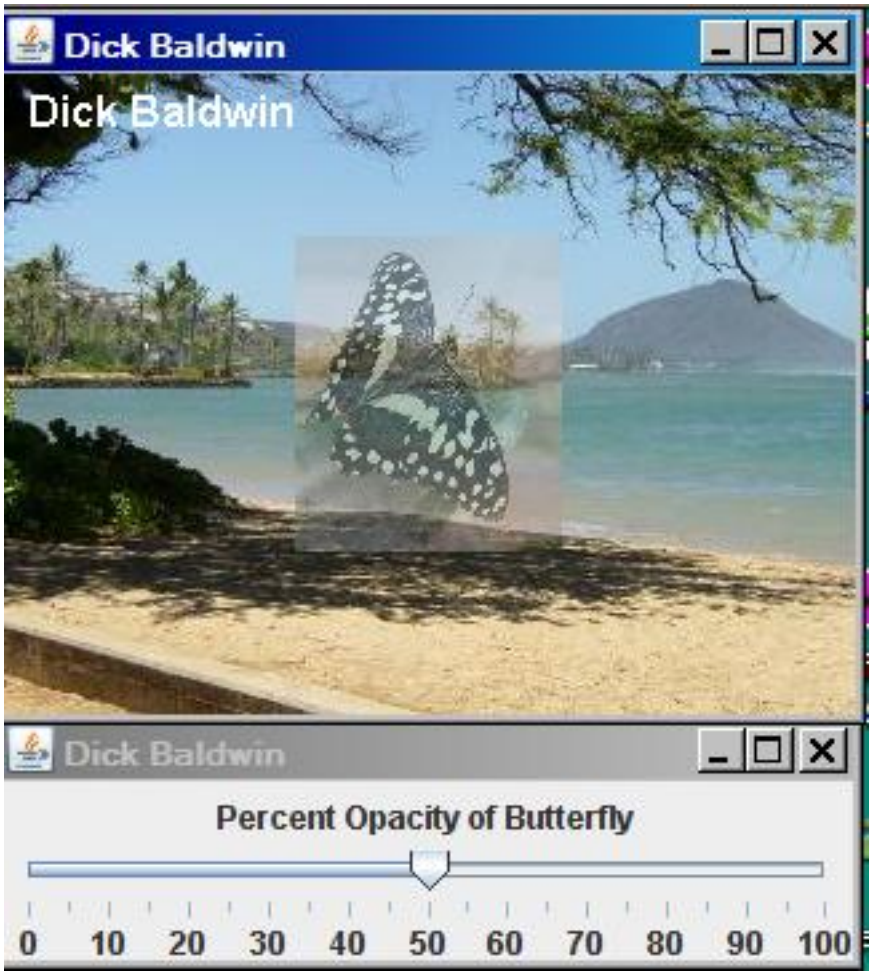


Figure 6.59: Required output images for Prob02.

6.4.2.1.8.3 Program 3

Listing 6.141: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library along with the image file named Prob3.jpg to produce the graphic output images shown in Figure 5 (p. 1695) below.

Click here [88] to download a zip file containing the required image file along with the source code for a solution.

The top image shown in Figure 5 (p. 1695) is an image of a butterfly to which an edge detection algorithm has been applied. The bottom image is a slider that is used to control the edge-detection threshold.

The edge-detection algorithm performs edge detection on a Picture object by rows and also by columns. All edges that are detected by processing adjacent pixels on a row are marked in red. All edges that are detected by processing adjacent pixels on a column are marked in black. If a pixel is determined to be on an edge using both approaches, it ends up being black. If an edge is not detected, the corresponding pixel is marked in white.

At startup, the slider is positioned at the 50-percent mark and the image has been edge-detected using a threshold value of 50. As you move the slider to the right, the threshold increases up to a value of 100, which in turn causes the amount of white area in the image to increase. As you move the slider to the left, the threshold decreases down to a value of zero, which in turn causes the amount of white area in the image to decrease.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

[88] http://cnx.org/content/m44265/latest/Prob03solution.zip

Required output images for Prob03.

Figure 6.60: Required output images for Prob03.

6.4.2.1.8.4 Program 4

Listing 6.142: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses Ericson's media library along with the image files named Prob04a.jpg and Prob04b.jpg to produce the graphic output images shown in Figure 6 (p. 1697) and Figure 7 (p. 1698) below.

Click here [89] to download a zip file containing the required image files along with the source code for a solution.

[89] http://cnx.org/content/m44265/latest/Prob04solution.zip

The top image shown in Figure 6 (p. 1697) is a butterfly image. The image immediately below that one is a slider that is used to control a scale factor that is applied to an image of a beach.

At startup, the slider is positioned at the zero-percent mark *(at the far left)* and the beach image is too small to be seen in the upper-left corner of the butterfly image.

As you move the slider to the right, an image of a beach emerges from the upper-left corner covering the image of the butterfly.

Figure 7 (p. 1698) shows the result of moving the slider to the 50-percent mark.

The size of the beach image increases and decreases smoothly as you move the slider back and forth. The upper- left corner of the beach image is always in the upper- left corner of the butterfly image. The butterfly becomes completely covered by the beach image when the slider is positioned at 100 percent *(the far right)* .

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

Required output images for Prob04.

Figure 6.61: Required output images for Prob04.

Required output image for Prob04.

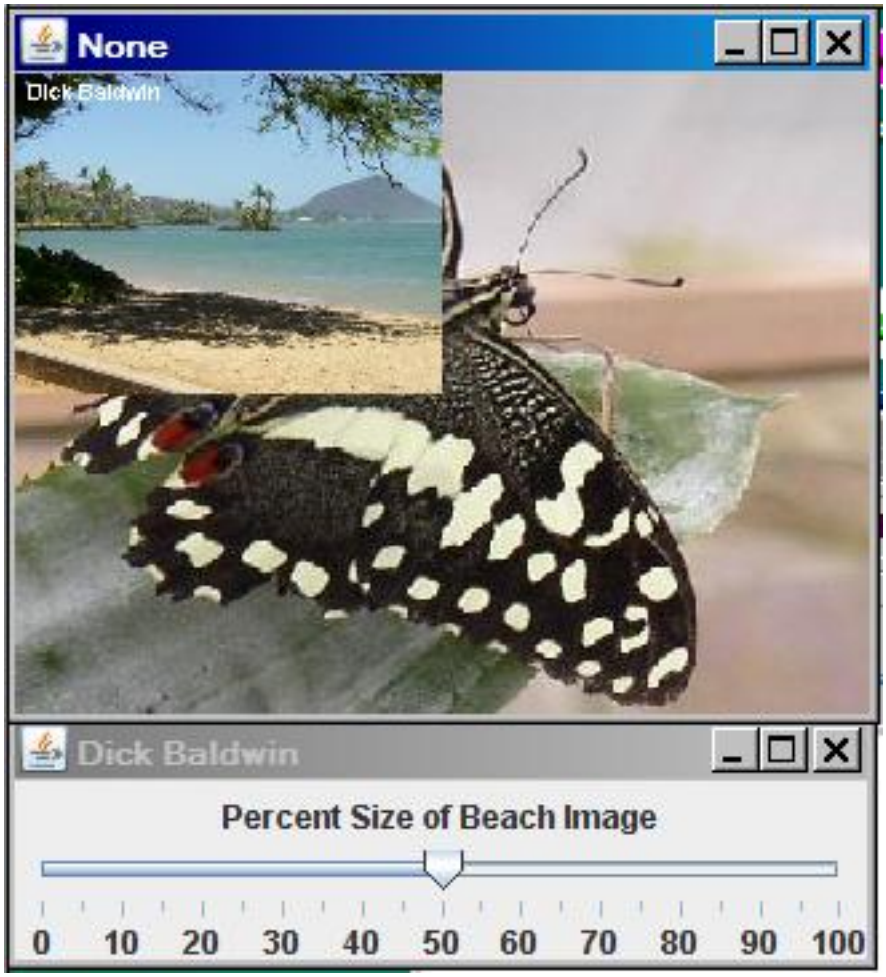


Figure 6.62: Required output image for Prob04.

6.4.2.1.8.5 Program 5

Listing 6.143: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library along with the image file named Prob05.jpg to produce the graphic output images shown in Figure 8 (p. 1700) and Figure 9 (p. 1701) below.

Click here [90] to download a zip file containing the required image file along with the source code for a solution.

[90] http://cnx.org/content/m44265/latest/Prob05solution.zip

The top image in Figure 8 (p. 1700) is a butterfly image. The image of the butterfly can be rotated in its picture by any angle ranging from -360 degrees to +360 degrees.

The image immediately below that one is a slider that is used to control the rotation angle that is applied to the butterfly image.

At startup, the slider is positioned at the zero-degrees mark *(in the center)* and the butterfly image is displayed with no rotation. As you move the slider to the right, the butterfly image rotates clockwise around its center through an angle that can be as large as 360 degrees.

Figure 9 (p. 1701) shows the result of moving the slider to the +120-degree mark.

As you move the slider to the left, the butterfly image rotates counter-clockwise around its center through an angle that can be as large as -360 degrees.

The butterfly image rotates smoothly around its center as you move the slider back and forth.

The program must terminate and return control to the operating system when you click the large X in the upper- right corner of the GUI containing the slider.

Required output image for Prob05.

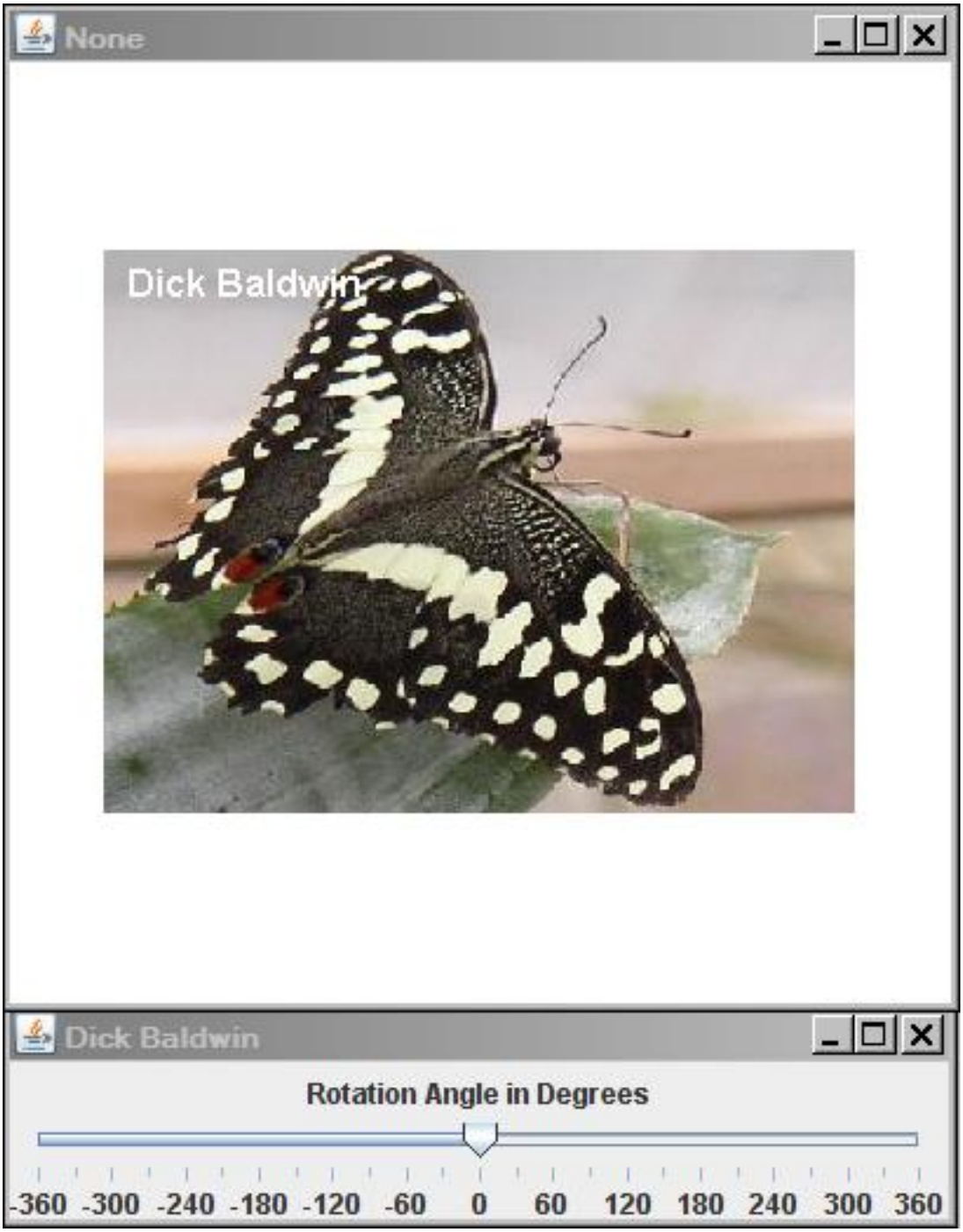


Figure 6.63: Required output image for Prob05.

Required output image for Prob05.

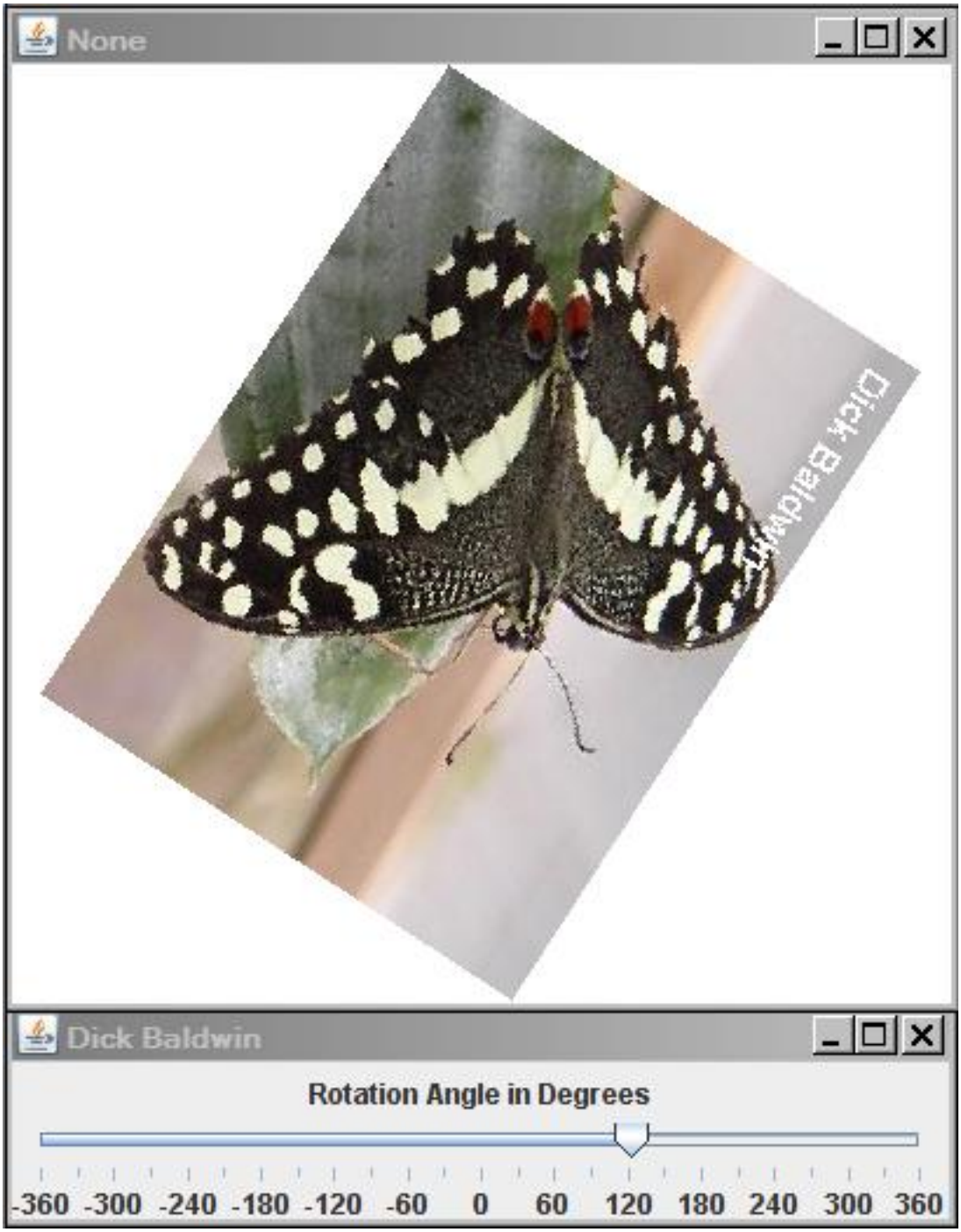


Figure 6.64: Required output image for Prob05.

6.4.2.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2317 Practice Test 2
- File: PracticeTest02.htm
- Published: August 10, 2012
- Revised: –

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

6.4.3 Java OOP: ITSE 2317 Practice Test 3[91]

6.4.3.1 ITSE2317 - Java Programming (Intermediate) - Practice Test 3

6.4.3.1.1 Java and Media Library Version Requirements

Your programs must be compatible with Sun's Standard Edition JDK Version 1.7 or later.

Some of the programs on this test require you to use the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [92] .

6.4.3.1.2 Input Image Files

Links are provided within the individual program specifications for downloading zip files that contain any image files that may be required to write, compile, and test your programs.

6.4.3.1.3 Solution source code files

The downloadable zip files mentioned above also contains source code files for the programming solutions. You can compile and execute those programs using procedures described in Java OOP: The Guzdial-Ericson Multimedia Class Library [93] .

6.4.3.1.4 Output Images

Your output image(s) must match my output image(s) in every respect including color, size, position, etc. Don't forget to display your name in the output image(s) as shown.

6.4.3.1.5 Hints

For some of the programs, you may first need to deduce the algorithm used to transform the input image into the output image, and then write a working program that implements that algorithm. In some cases, you may need to compare numeric color values for corresponding pixels in the input and output images in order to deduce the algorithm.

You can obtain those color values using the following procedure:

[91] This content is available online at <http://cnx.org/content/m44262/1.2/>.
[92] http://cnx.org/content/m44148/latest/
[93] http://cnx.org/content/m44148/latest/

1. Click the download link for the zip files that contain input image files and solution source code files. Use the capabilities of your browser to download and save the contents of those zip files.
2. If necessary, replace calls to the **show** method in my source code with calls to the **explore** method to force the program to display the output images in a **PictureExplorer** window.
3. Compile and run the source code.
4. Write, compile, and run a simple Java program that will display each input image file in a **PictureExplorer** window.
5. Use the input and output **PictureExplorer** windows to compare the input and output color values on a pixel by pixel basis.

You may find other useful hints in my online tutorials and slides for this course.

6.4.3.1.6 Testing Your Programs

You can compile and execute your program by following the instructions given at Java OOP: The Guzdial-Ericson Multimedia Class Library [94] .

6.4.3.1.7 Program Specifications

6.4.3.1.7.1 Program 1

Listing 6.144: Write the Java application described below.

/*File Prob01 Copyright 2012 R.G.Baldwin

Write a program named Prob01 that uses Ericson's media library along with the image file named Prob01a.jpg to produce the graphic output image shown in Figure 1 (p. 1705) below.

Click here [95] to download a zip file containing the required image file along with the source code for a solution.

When the program first starts running, only the GUI containing the text field is visible. When you type the file name into the text field and press Enter, the image of the penguin appears and the GUI moves to a position below the image of the penguin. The program terminates and returns control to the operating system when you click the X in the upper-right corner of the GUI.

[94] http://cnx.org/content/m44148/latest/
[95] http://cnx.org/content/m44262/latest/Prob01solution.zip

Required output images for Prob01.

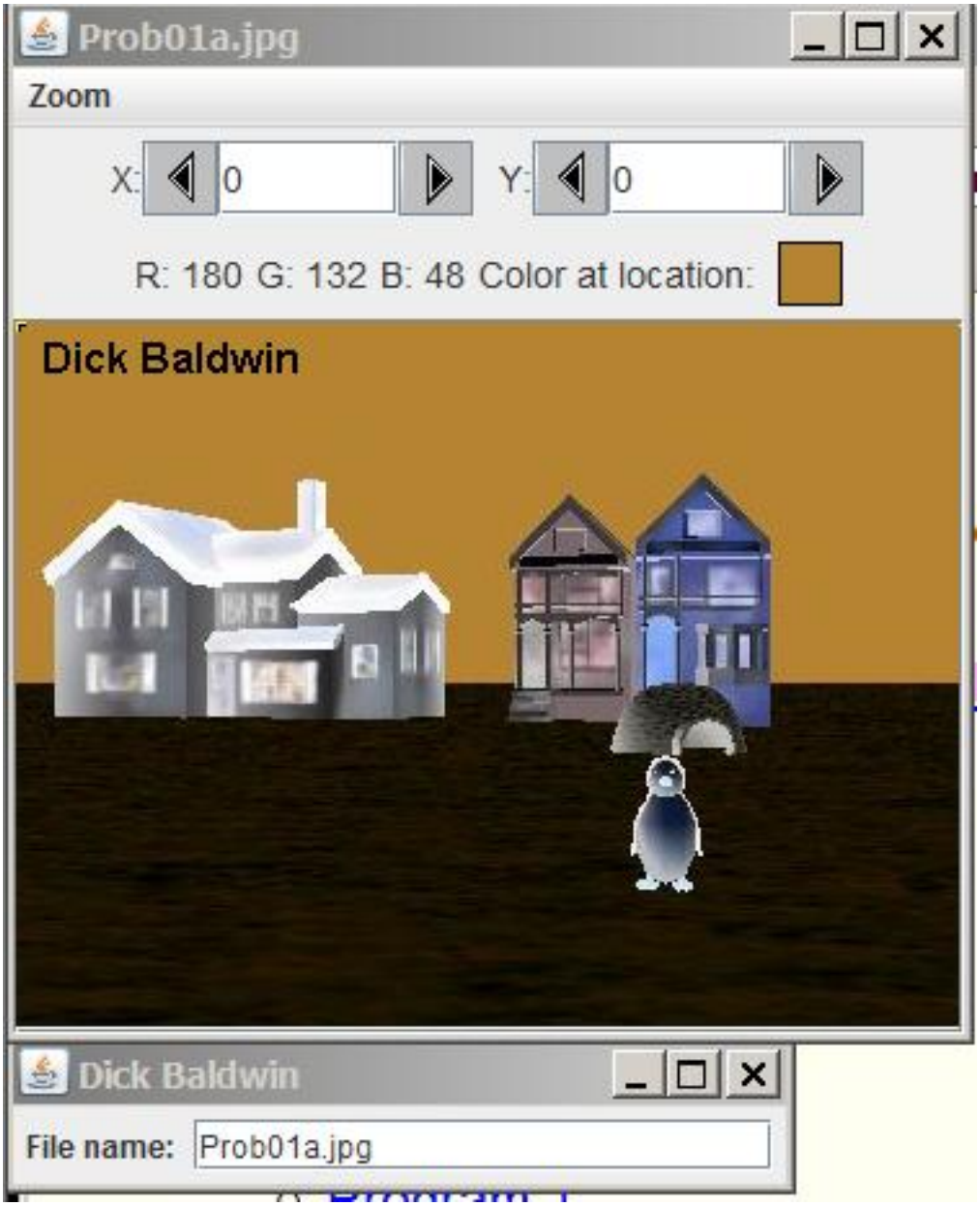


Figure 6.65: Required output images for Prob01.

6.4.3.1.7.2 Program 2

Listing 6.145: Write the Java application described below.

/*File Prob02 Copyright 2012 R.G.Baldwin

Write a program named Prob02 that uses Ericson's media library along with the image file named Prob02.jpg to produce the graphic output images shown in Figure 2 (p. 1707) below.

Click here [96] to download a zip file containing the required image file along with the source code for a solution.

Whenever you click the button in the GUI in the lower image, the green color component value at the location of the crosshair cursor in the upper image is displayed in the text field in the GUI.

[96] http://cnx.org/content/m44262/latest/Prob02solution.zip

Required output images for Prob02.

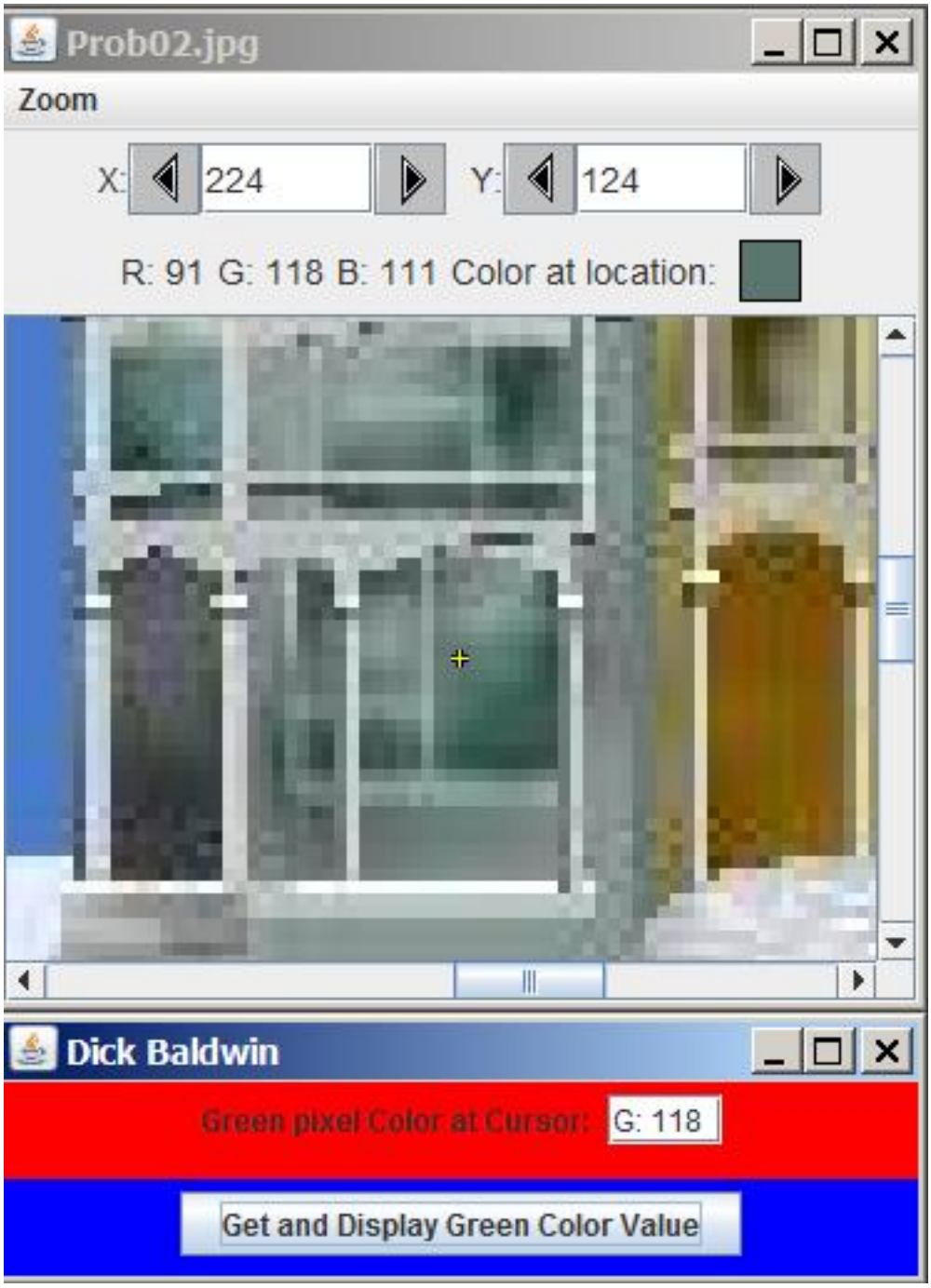


Figure 6.66: Required output images for Prob02.

6.4.3.1.7.3 Program 3

Listing 6.146: Write the Java application described below.

/*File Prob03 Copyright 2012 R.G.Baldwin

Write a program named Prob03 that uses Ericson's media library to produce the graphic output image shown in Figure 3 (p. 1708) below.

Click here [97] to download a zip file containing the source code for a solution.

Whenever you click the button in the GUI, the square in the upper right turns from black to yellow with a black border and the RGB color values for the color yellow appear in the three text fields.

Required output image for Prob03.

Figure 6.67: Required output image for Prob03.

6.4.3.1.7.4 Program 4

Listing 6.147: Write the Java application described below.

/*File Prob04 Copyright 2012 R.G.Baldwin

Write a program named Prob04 that uses Ericson's media library to produce the graphic output images shown in Figure 4 (p. 1709) below.

Click here [98] to download a zip file containing the source code for a solution.

Whenever you click the "Choose Color" button in the GUI in the upper image, the color chooser dialog shown in the lower image appears. When you select a color and click the OK button, the color chooser dialog disappears and that color appears in the square in the upper right portion of the GUI. In addition, the red, green, and blue color component values for that color appear in the corresponding text fields.

Whenever you click the "Brighter" and "Darker" buttons, the color displayed in the square becomes brighter or darker and the color values in the text fields change accordingly.

[97] http://cnx.org/content/m44262/latest/Prob03solution.zip
[98] http://cnx.org/content/m44262/latest/Prob04solution.zip

Required output images for Prob04.

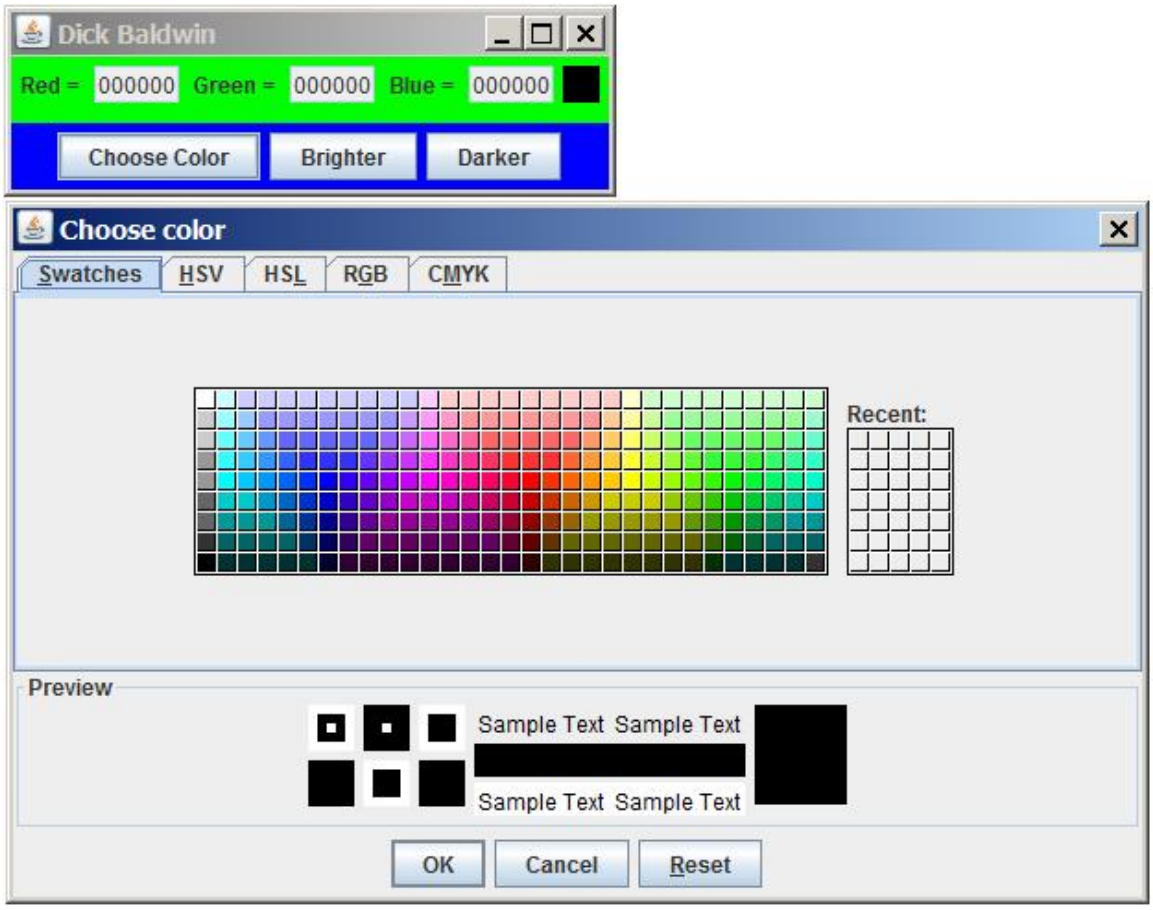


Figure 6.68: Required output images for Prob04.

6.4.3.1.7.5 Program 5

Listing 6.148: Write the Java application described below.

/*File Prob05 Copyright 2012 R.G.Baldwin

Write a program named Prob05 that uses Ericson's media library along with the image file named Prob05.jpg to produce the graphic output images shown in Figure 5 (p. 1711) below.

Click here [99] to download a zip file containing the required image file along with the source code for a solution.

The color in the square in the upper right of the GUI in the lower image is governed by the color values in the red, green, and blue text fields.

Whenever you click the button in the GUI, the pixel at the crosshair cursor in the upper image is changed to reflect the color in the square in the GUI. This can best be seen when the upper

[99] http://cnx.org/content/m44262/latest/Prob05solution.zip

image is zoomed to 500%.

Required output image for Prob05.

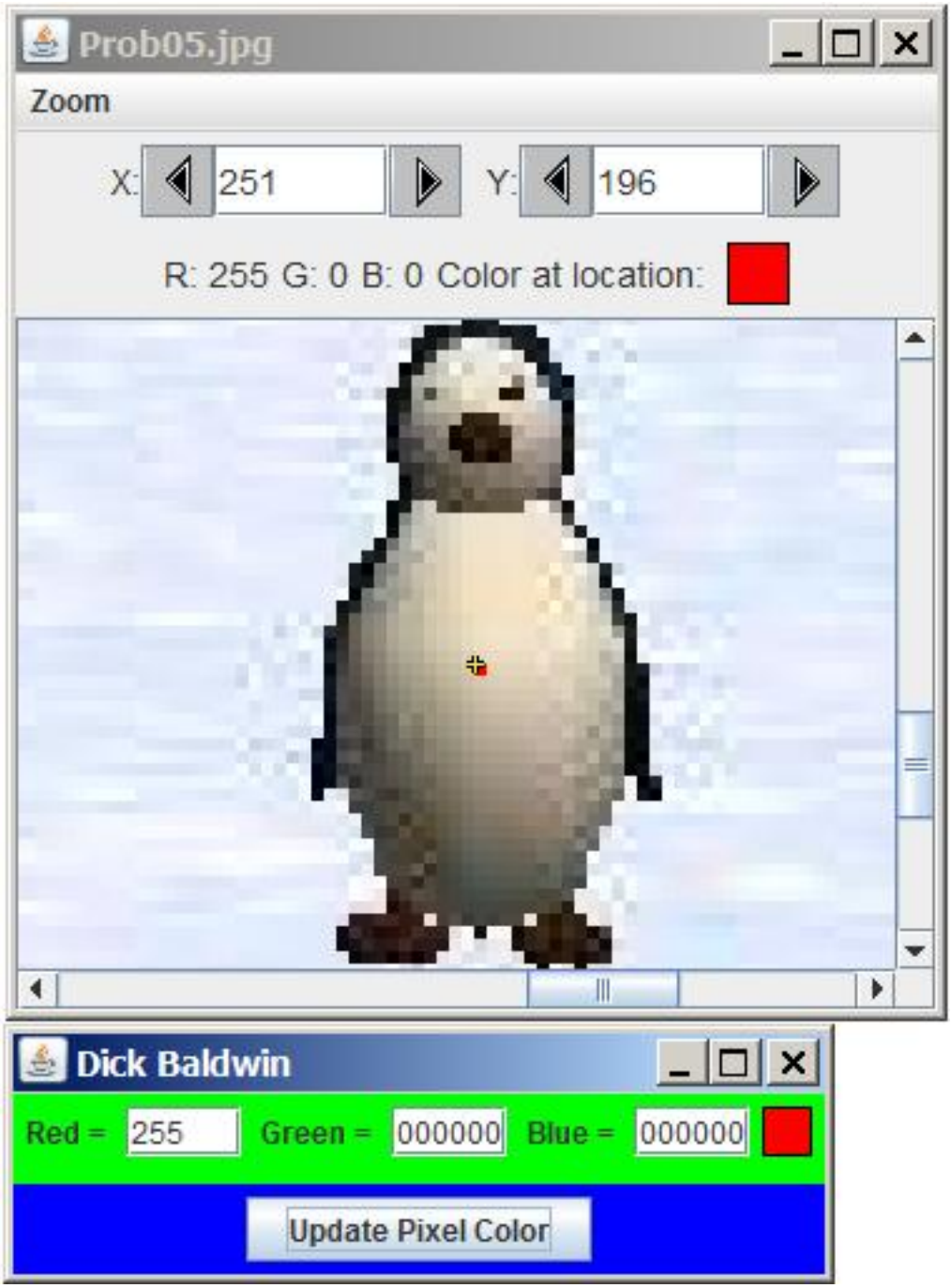


Figure 6.69: Required output image for Prob05.

6.4.3.2 Miscellaneous Information

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: ITSE 2317 Practice Test 3
- File: PracticeTest03.htm
- Published: August 9, 2012
- Revised: August 10, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 7

GAME 2302 - Mathematical Applications for Game Development

7.1 Jy0040: GAME2302: Mathematical Applications for Game Development[1]

7.1.1 Table of Contents

7.1.2 Welcome

Click the link to view the course material for GAME 2302 Mathematical Applications for Game Development [2] , which I teach at Austin Community College in Austin, TX.

Official information about the course

The college website for this course is: http://www.austincc.edu/baldwin/ [3] .

As of December 2012, the description for this course reads:

"GAME 2302 - Mathematical Applications for Game Development

Presents applications of mathematics and science in game and simulation programming. Includes the utilization of matrix and vector operations, kinematics, and Newtonian principles in games and simulations. Also covers code optimization."

7.1.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: GAME2302: Mathematical Applications for Game Development
- File: Jy0040.htm
- Published: 01/17/13

[1] This content is available online at <http://cnx.org/content/m45680/1.2/>.
[2] http://cnx.org/content/col11450
[3] http://www.austincc.edu/baldwin/

NOTE: **Disclaimers:: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 8

Anatomy of a Game Engine

8.1 Jy0060: Anatomy of a Game Engine[1]

8.1.1 Table of Contents

8.1.2 Welcome

Click the link to view the material for Anatomy of a Game Engine [2] , which I use in some of the courses that I teach at Austin Community College in Austin, TX.

This is a collection of modules designed to teach students about the anatomy of a typical game or simulation engine *(sometimes called a game or simulation framework)* .

The collection is built around the **Slick2D** Game library.

The Slick2D library

I chose to concentrate on the free game library named Slick2D [3] *,(which is written in Java)* for several reasons including the following:

- Java is the language with which I am the most comfortable. Hence, I can probably do a better job of explaining the anatomy of a game engine that uses Slick2D than would be the case for a game engine written in C++, C#, Python, or some other programming language.
- Java has proven in recent years to be a commercially successful game programming language. For example, I cite the commercial game named Minecraft [4] , written in Java, for which apparently millions of copies have been sold. Also, knowing Java is very beneficial for those who might want to develop apps for Android.
- Slick2D is free and the source code for Slick2D is readily available.
- The overall structure of a basic Slick2D game engine is very similar to Dark GDK and XNA, and is probably similar to other game engines as well.
- Java is platform independent.

Applicable to other environments as well

Although the modules in this collection concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to other game engines written in different programming languages.

[1] This content is available online at <http://cnx.org/content/m45747/1.1/>.
[2] http://cnx.org/content/col11489/latest/
[3] http://slick.cokeandcode.com/index.php
[4] http://minecraft.net/

8.1.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0060: Anatomy of a Game Engine
- File: Jy0060.htm
- Published: 02/05/13

NOTE: **Disclaimers:: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 9

Principles of Object-Oriented Programming

9.1 Jy0070-Principles of Object-Oriented Programming[1]

9.1.1 Table of Contents

9.1.2 Welcome

Click the following title to view Principles of Object-Oriented Programming [2] *(current as of 02/20/13)* :

- **Summary** : An objects-first with design patterns introductory course
- **Instructor** : Stephen Wong and Dung Nguyen
- **Institution** : Rice University
- **Course Number** : COMP201

One of the great things about cnx.org is the ability for an instructor like myself to incorporate course materials shared by others. This makes it possible for the courses that I teach to have more breadth than might otherwise be the case.

It may be beneficial for you to study this material, which you will probably find to be more formal and rigorous than the material that I have provided. Pay particular attention to the design pattern concept as well as the UML diagrams.

9.1.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0070-Principles of Object-Oriented Programming
- File: Jy0070.htm
- Published: 02/20/13

[1] This content is available online at <http://cnx.org/content/m45793/1.1/>.
[2] http://cnx.org/content/col10213/latest/

NOTE: **Disclaimers:: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 10

Programming Oldies But Goodies

10.1 Jy0050: Programming Oldies But Goodies[1]

10.1.1 Table of Contents

10.1.2 Welcome

Over the years, I have published a large number of tutorials in the areas of computer programming and DSP. As I have the time to do so, I am converting the more significant of those tutorials into cnxml code and re-publishing them at cnx.org. In the meantime, the collection titled Programming Oldies But Goodies [2] , which is a work in process, gathers many of the tutorials in their original HTML format into a common location to make them readily available for Connexions users.

10.1.3 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jy0050: Programming Oldies But Goodies
- File: Jy0050.htm
- Published: 01/17/13

NOTE: **Disclaimers:: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

[1] This content is available online at <http://cnx.org/content/m45681/1.1/>.
[2] http://cnx.org/content/col11478

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 11

Appendices

11.1 Java OOP: Java Documentation[1]

11.1.1 Table of Contents

11.1.2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

I cannot overemphasize the importance of Oracle's Java documentation [2] to aspiring Java programmers. The documentation package, which can be downloaded for installation and local access or accessed online, contains a wealth of information.

In my opinion, it is not possible to write Java programs of any substance without frequent reference to the documentation. No one can memorize everything that they need to know to be a successful Java programmer.

(Note that each time Oracle releases a new version of the Java Development Kit (JDK), they also release a new version of the documentation. Therefore, as new versions are released, the links provided in this document may become outdated and may not take you to the latest version of the documentation. However, you should be able to find the latest documentation via an online search.)

11.1.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the figures while you are reading about them.

[1] This content is available online at <http://cnx.org/content/m45117/1.1/>.
[2] http://docs.oracle.com/javase/7/docs/

11.1.2.1.1 Figures

- Figure 1 (p. 1723) . Screen shot of the API specification at startup.
- Figure 2 (p. 1725) . Screen shot of the API documentation after selecting the JButton class.

11.1.3 Discussion

Small core language, large class library

The *Java Platform Standard Edition* programming environment consists of a small core programming language and a large class library.

(As of the date of this writing, you can view The Java Language Specification [3] *online. This specification will tell you just about everything that most programmers need to know about the core language.)*

The size of the class library grows with the release of each new version of the JDK, because each new version provides capabilities that didn't exist in the previous version. New capabilities are added through the addition of new classes and new interfaces to the library.

The true power of Java resides in the class libraries

Once you understand how OOP is implemented in Java and you get beyond **while** loops, **if** statements, and other fundamental programming concepts, virtually all the power of Java resides in:

- Classes, interfaces, and methods in the various class libraries that you use directly.
- Classes, interfaces, and methods in the various class libraries that you extend.
- New classes, interfaces, and methods that you and others define.

You will always use material from Oracle's standard class libraries. You will often use material from other libraries that you create yourself, or that you obtain from sources outside of Oracle *(such as Barb Ericson's multimedia library* [4] *, for example)* .

Top-level online documentation

As of the date of this writing, you can view the top-level page of the documentation for *Java Platform Standard Edition 7* at http://docs.oracle.com/javase/7/docs/ [5] .

A complicated page

When you first view that documentation page, it may appear to be very complicated. Of all the material on the page, the most important is probably the *Application Programming Interface (API)* , which is currently available via a link on the right side of the page labeled Java SE API [6] .

However, you should not ignore the links to other information available on the top-level page [7] . Those links will often contain information that you need, and that information can make you more efficient in using the API documentation.

Search

If you follow the Search link at the top of the page, you will arrive at a search engine page [8] that gives you the ability to do a keyword search on the online documentation.

Downloading and installing Java

Of particular interest to most newcomers should be the links titled Installation Instructions [9] and Java SE Downloads [10] .

The Java Tutorials

[3] http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
[4] http://coweb.cc.gatech.edu/mediaComp-plan/101
[5] http://docs.oracle.com/javase/7/docs/
[6] http://docs.oracle.com/javase/7/docs/api/index.html
[7] http://docs.oracle.com/javase/7/docs/
[8] http://docs.oracle.com/javase/search.html
[9] http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
[10] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Also very important is the link to The Java Tutorials [11] . Once you reach those tutorials, of particular importance is the Path and CLASSPATH [12] tutorial. *(Many students trip on the path, the classpath, and the difference between the two.)*

11.1.4 Java Platform, Standard Edition 7 API Specification

The *API Specification* [13] is probably the most frequently used section of the entire documentation package.

A screen shot

Figure 1 (p. 1723) shows a partial screen shot of what you should see when you first load the API Specification into your browser. *(Note that this screen shot was cropped to force it to fit into this publication format.)*

Screen shot of the API specification at startup.

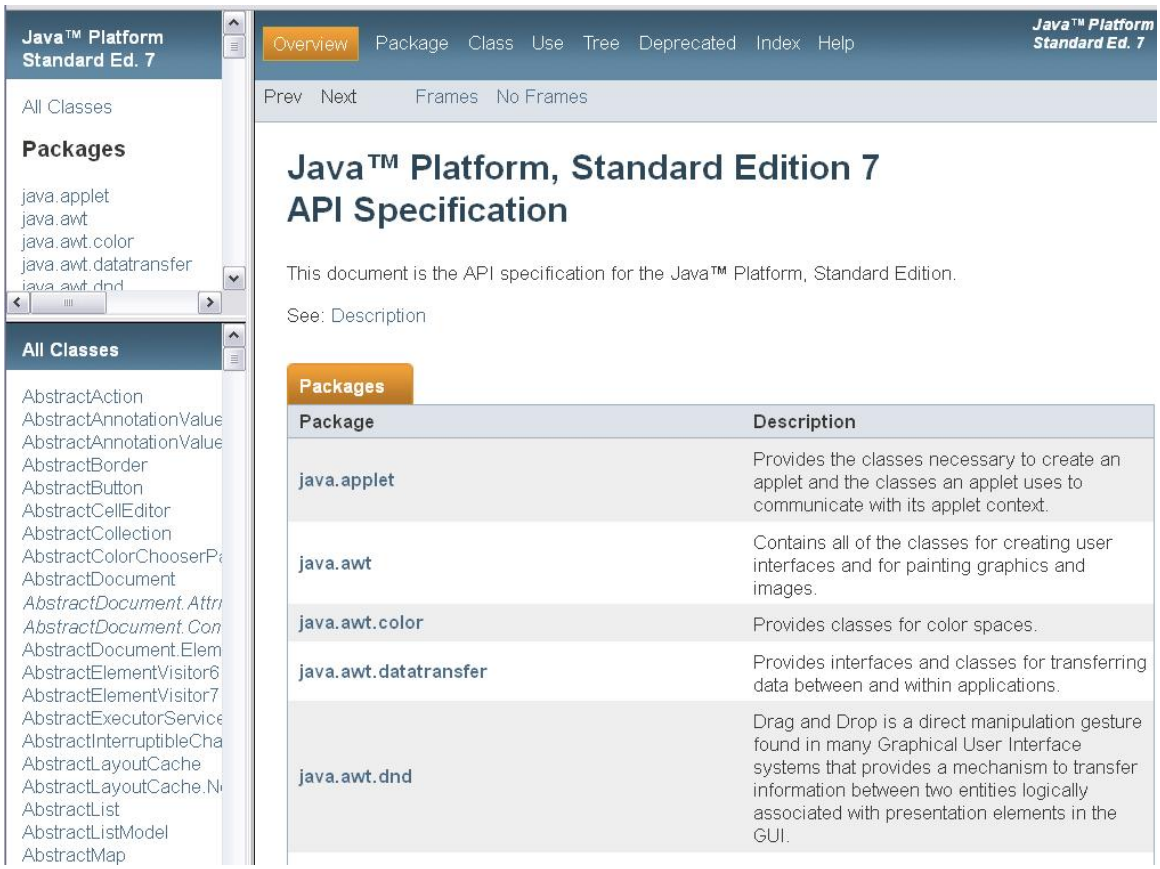


Figure 11.1: Screen shot of the API specification at startup.

---

[11] http://docs.oracle.com/javase/tutorial/
[12] http://docs.oracle.com/javase/tutorial/essential/environment/paths.html
[13] http://docs.oracle.com/javase/7/docs/api/index.html

Screen layout

The purpose of providing this screen shot is to give you an idea of the general layout of the material as it appears on your screen. As you can see, the layout consists of three frames when viewed in your HTML browser.

(A version without frames is also available by selecting the "No Frames" link at the top of the page. The version without frames is particularly useful for using the "page search" capability of your browser to find something on the page.)

The two leftmost frames

The upper-left frame contains a list of *packages* . The lower-left frame contains a list of the *classes* and *interfaces* that are contained in the package that is selected in the top-left frame.

(The default package selection in the upper-left frame is "All Classes".)

You make selections in these two frames *(or in the link bar at the top of the page)* to control the contents of the rightmost frame.

The rightmost frame

When you first access the API documentation *(and when you select* ***Overview*** *at the top of the page)*, the rightmost frame contains summary information about all of the packages.

When you select a class in the lower-left frame *(such as the* ***JButton*** *class for example)* , the rightmost frame contains hyperlinked information about that class, including:

- A visual class hierarchy diagram
- Interfaces implemented by the class
- Known subclasses of the class
- Text description of the class
- Summary of nested classes
- Summary of fields defined in the class
- Summary of fields inherited into the class
- Summary and detailed information on constructors defined in the class
- Summary and detailed information on methods defined in the class
- List of methods inherited into the class

Figure 2 (p. 1725) shows a screen shot of the browser window after selecting the class named **JButton** in the lower-left frame. If you pull down the thumb in the scroll bar on the right side of the browser window, you will expose all of the information in the above list.

Screen shot of the API documentation after selecting the JButton class.

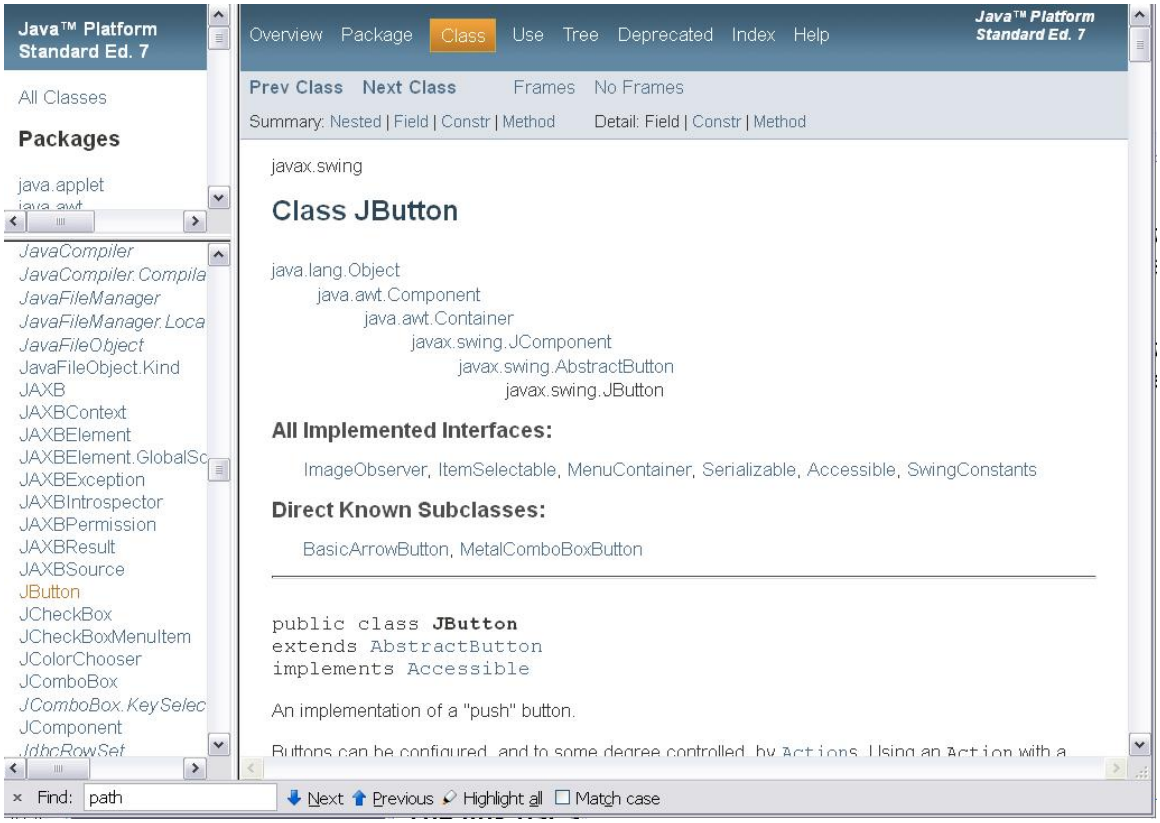


Figure 11.2: Screen shot of the API documentation after selecting the JButton class.

The link bar at the top of the page

It can sometimes be difficult to find what you are looking for in the documentation package. Referring back to Figure 2 (p. 1725) , you can see a link bar at the top of the page in the rightmost frame. This link bar contains the following hyperlinks:

- **Overview** - provides summary information about packages *(shown in Figure 1 (p. 1723)).*
- **Package** - provides a description of the package that contains the class selected in the lower-left frame.
- **Class** - provides a description of the class selected in the lower-left frame.
- **Use** - describes how the class selected in the lower-left frame is used in various packages.
- **Tree** - provides detailed inheritance hierarchy information for a selected package.
- **Deprecated** - provides a list of material that has been deprecated *(may not be supported in future versions)*
- **Index** - provides a hyperlinked alphabetized index of interfaces, classes, constructors, variables, and methods.
- **Help** - Describes how the API document is organized.

The various pages that are displayed by selecting these links can often help you to find what you are looking for. Probably the most useful and frequently consulted item in the above list is the *Index*.

The alphabetical index

The alphabetical index can be extremely useful if you know the full or partial spelling of what you are looking for, beginning with the first character.

For example, assume that you remember *(or you can surmise from what you know about Java properties)* that there is a method whose name begins with **getSystemLookAndFeel** , which returns the name of the *LookAndFeel* class that implements the native look and feel for a particular operating system. You can easily look this up under G in the alphabetical index.

What you will find when you look it up is a brief description of a method named **getSystemLookAndFeelClassName** that matches what you are looking for. The name of the method is also a hyperlink to the detailed description of the same method as it appears in the documentation for the class named **UIManager** .

11.1.5 A universal documentation format

The documentation for the API is created using a program named **javadoc.exe** that is contained in the JDK. Therefore, the API documentation for class libraries obtained from outside sources should have the same format as that described above *(assuming that the documentation was produced using javadoc.exe)* .

However, the program named **javadoc.exe** only controls the format of the documentation. It does not produce the explanatory content. It is the responsibility of the author of the class library to provide that content.

11.1.6 Typical usage of the API documentation

Typical usage of the API documentation consists of the following steps:

- Open the API documentation in your browser.
- Click the class of interest in the lower-left frame.
- Manually search the rightmost frame for summary information about fields, constructors, or methods of interest.
- Click the name of a field, constructor, or method in the summary section to open a detailed description of that field, constructor, or method.

11.1.7 Summary

There are several ways to search for useful information in the Oracle documentation. There is no single approach that will serve all of your needs to find information in the documentation. You simply need to become familiar with the different ways to search for information in the documentation and be prepared to use the approach that does the best job in each situation.

11.1.8 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Java Documentation
- File: Java3140.htm
- Published: 11/11/12

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Object-Oriented Programming (OOP) with Java*
Edited by: Richard Baldwin
URL: http://cnx.org/content/col11441/1.121/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jy0010: Preface to OOP with Java"
By: Richard Baldwin
URL: http://cnx.org/content/m45136/1.9/
Pages: 1-3
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Gf0100: Objects First with Greenfoot"
By: Richard Baldwin
URL: http://cnx.org/content/m45790/1.1/
Pages: 5-10
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0005: Preface to OOP Self-Assessment"
By: Richard Baldwin
URL: http://cnx.org/content/m45252/1.6/
Pages: 11-12
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0010: Self-assessment, Primitive Types"
By: Richard Baldwin
URL: http://cnx.org/content/m45284/1.4/
Pages: 12-36
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0020: Self-assessment, Assignment and Arithmetic Operators"
By: Richard Baldwin
URL: http://cnx.org/content/m45286/1.3/
Pages: 37-68
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures"
By: Richard Baldwin
URL: http://cnx.org/content/m45287/1.2/
Pages: 69-93
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method"
By: Richard Baldwin
URL: http://cnx.org/content/m45260/1.4/
Pages: 93-112

Module: "Ap0050: Self-assessment, Escape Character Sequences and Arrays"
By: Richard Baldwin
URL: http://cnx.org/content/m45280/1.4/
Pages: 112-140

Module: "Ap0060: Self-assessment, More on Arrays"
By: Richard Baldwin
URL: http://cnx.org/content/m45264/1.4/
Pages: 140-167

Module: "Ap0070: Self-assessment, Method Overloading"
By: Richard Baldwin
URL: http://cnx.org/content/m45276/1.4/
Pages: 167-183

Module: "Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods"
By: Richard Baldwin
URL: http://cnx.org/content/m45279/1.4/
Pages: 183-202

Module: "Ap0090: Self-assessment, the super keyword, final keyword, and static methods"
By: Richard Baldwin
URL: http://cnx.org/content/m45270/1.4/
Pages: 203-223

Module: "Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables"
By: Richard Baldwin
URL: http://cnx.org/content/m45296/1.3/
Pages: 223-245

Module: "Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior"
By: Richard Baldwin
URL: http://cnx.org/content/m45308/1.3/
Pages: 245-266
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0120: Self-assessment, Interfaces and polymorphic behavior"
By: Richard Baldwin
URL: http://cnx.org/content/m45303/1.3/
Pages: 266-299
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions"
By: Richard Baldwin
URL: http://cnx.org/content/m45310/1.3/
Pages: 300-320
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references"
By: Richard Baldwin
URL: http://cnx.org/content/m45302/1.3/
Pages: 320-340
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0103 Preface to Programming Fundamentals"
By: Richard Baldwin
URL: http://cnx.org/content/m45179/1.6/
Pages: 341-342
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0105: Java OOP: Similarities and Differences between Java and C++"
By: Richard Baldwin
URL: http://cnx.org/content/m45142/1.2/
Pages: 342-346
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0110: Java OOP: Programming Fundamentals, Getting Started"
By: Richard Baldwin
URL: http://cnx.org/content/m45137/1.3/
Pages: 346-351
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0110r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45162/1.5/
Pages: 352-356

Module: "Jb0115: Java OOP: First Program"
By: Richard Baldwin
URL: http://cnx.org/content/m45220/1.2/
Pages: 356-361

Module: "Jb0120: Java OOP: A Gentle Introduction to Java Programming"
By: Richard Baldwin
URL: http://cnx.org/content/m45138/1.2/
Pages: 361-367

Module: "Jb0120r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45164/1.4/
Pages: 368-372

Module: "Jb0130: Java OOP: A Gentle Introduction to Methods in Java"
By: Richard Baldwin
URL: http://cnx.org/content/m45139/1.2/
Pages: 372-380

Module: "Jb0130r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45165/1.4/
Pages: 381-386

Module: "Jb0140: Java OOP: Java comments"
By: Richard Baldwin
URL: http://cnx.org/content/m45140/1.3/
Pages: 386-391

Module: "Jb0140r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45169/1.4/
Pages: 392-395

Module: "Jb0150: Java OOP: A Gentle Introduction to Java Data Types"
By: Richard Baldwin
URL: http://cnx.org/content/m45141/1.2/
Pages: 396-409
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0150r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45168/1.4/
Pages: 410-418
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0160: Java OOP: Hello World"
By: Richard Baldwin
URL: http://cnx.org/content/m45143/1.2/
Pages: 418-424
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0160r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45159/1.7/
Pages: 425-431
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0170: Java OOP: A little more information about classes."
By: Richard Baldwin
URL: http://cnx.org/content/m45144/1.2/
Pages: 431-433
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0170r: Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45177/1.4/
Pages: 434-437
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0180: Java OOP: The main method."
By: Richard Baldwin
URL: http://cnx.org/content/m45145/1.2/
Pages: 437-440
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0180r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45171/1.4/
Pages: 441-444
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0190: Java OOP: Using the System and PrintStream Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m45148/1.2/
Pages: 445-448

Module: "Jb0190r: Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45175/1.4/
Pages: 449-455

Module: "Jb0200: Java OOP: Variables"
By: Richard Baldwin
URL: http://cnx.org/content/m45150/1.2/
Pages: 455-470

Module: "Jb0200r: Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45173/1.6/
Pages: 471-484

Module: "Jb0210: Java OOP: Operators"
By: Richard Baldwin
URL: http://cnx.org/content/m45195/1.4/
Pages: 484-493

Module: "Jb0210r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45186/1.4/
Pages: 494-511

Module: "Jb0220: Java OOP: Statements and Expressions"
By: Richard Baldwin
URL: http://cnx.org/content/m45192/1.3/
Pages: 511-513

Module: "Jb0220r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45189/1.4/
Pages: 514-518

Module: "Jb0230: Java OOP: Flow of Control"
By: Richard Baldwin
URL: http://cnx.org/content/m45196/1.4/
Pages: 518-536
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0230r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45218/1.4/
Pages: 537-545
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0240: Java OOP: Arrays and Strings"
By: Richard Baldwin
URL: http://cnx.org/content/m45214/1.4/
Pages: 545-558
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0240r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45208/1.4/
Pages: 559-568
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0250: Java OOP: Brief Introduction to Exceptions"
By: Richard Baldwin
URL: http://cnx.org/content/m45211/1.3/
Pages: 568-571
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0260: Java OOP: Command-Line Arguments"
By: Richard Baldwin
URL: http://cnx.org/content/m45246/1.4/
Pages: 571-574
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0260r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45244/1.5/
Pages: 575-579
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0270: Java OOP: Packages"
By: Richard Baldwin
URL: http://cnx.org/content/m45229/1.4/
Pages: 580-588
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jb0280: Java OOP: String and StringBuffer"
By: Richard Baldwin
URL: http://cnx.org/content/m45237/1.3/
Pages: 588-598

Module: "Jb0280r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45241/1.4/
Pages: 599-608

Module: "Jb0290: The end of Programming Fundamentals"
By: Richard Baldwin
URL: http://cnx.org/content/m45257/1.3/
Page: 608

Module: "Jy0020: Java OOP: Preface to ITSE 2321"
By: Richard Baldwin
URL: http://cnx.org/content/m45222/1.8/
Pages: 609-612

Module: "Java1600: Objects and Encapsulation"
By: Richard Baldwin
URL: http://cnx.org/content/m44153/1.3/
Pages: 612-619

Module: "Java1602: Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44150/1.3/
Pages: 619-631

Module: "Java1604: Inheritance, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m44193/1.3/
Pages: 631-641

Module: "Java1606: Inheritance, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m44156/1.3/
Pages: 641-649

Module: "Java1608: Polymorphism Based on Overloaded Methods"
By: Richard Baldwin
URL: http://cnx.org/content/m44182/1.3/
Pages: 649-656
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1610: Polymorphism, Type Conversion, Casting, etc."
By: Richard Baldwin
URL: http://cnx.org/content/m44168/1.3/
Pages: 656-667
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1612: Runtime Polymorphism through Inheritance"
By: Richard Baldwin
URL: http://cnx.org/content/m44177/1.3/
Pages: 667-676
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1614: Polymorphism and the Object Class"
By: Richard Baldwin
URL: http://cnx.org/content/m44190/1.3/
Pages: 676-685
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1616: Polymorphism and Interfaces, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m44195/1.3/
Pages: 685-696
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1618: Polymorphism and Interfaces, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m44196/1.3/
Pages: 696-709
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1620: Static Members"
By: Richard Baldwin
URL: http://cnx.org/content/m44197/1.4/
Pages: 709-727
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1622: Array Objects, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m44198/1.3/
Pages: 727-739
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1624: Array Objects, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m44199/1.3/
Pages: 739-762
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1626: Array Objects, Part 3"
By: Richard Baldwin
URL: http://cnx.org/content/m44200/1.4/
Pages: 762-775
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1628: The this and super Keywords"
By: Richard Baldwin
URL: http://cnx.org/content/m44201/1.4/
Pages: 775-790
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java1630: Exception Handling"
By: Richard Baldwin
URL: http://cnx.org/content/m44202/1.4/
Pages: 790-815
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3000: The Guzdial-Ericson Multimedia Class Library"
By: Richard Baldwin
URL: http://cnx.org/content/m44148/1.7/
Pages: 815-823
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3000r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45761/1.4/
Pages: 824-831
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3000s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45620/1.3/
Pages: 831-832
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3002: Creating and Manipulating Turtles and Pictures in a World Object"
By: Richard Baldwin
URL: http://cnx.org/content/m44149/1.7/
Pages: 832-853
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3002r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45762/1.3/
Pages: 854-871
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3002s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45621/1.5/
Pages: 871-872
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3004: Image Processing Algorithms, Image Inversion, and PictureExplorer Objects"
By: Richard Baldwin
URL: http://cnx.org/content/m44203/1.6/
Pages: 872-885
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3004r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45763/1.4/
Pages: 886-895
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3004s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45622/1.4/
Pages: 895-896
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3006: Implementing a space-wise linear color-modification algorithm."
By: Richard Baldwin
URL: http://cnx.org/content/m44204/1.6/
Pages: 896-908
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3006r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45768/1.2/
Pages: 909-916
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3006s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45623/1.3/
Pages: 916-917
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3008: Abstract Methods, Abstract Classes, and Overridden Methods"
By: Richard Baldwin
URL: http://cnx.org/content/m44205/1.5/
Pages: 917-926

Module: "Java3008r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45773/1.1/
Pages: 927-933

Module: "Java3008s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45624/1.3/
Pages: 933-934

Module: "Java3010: Indirection, Array Objects, and Casting"
By: Richard Baldwin
URL: http://cnx.org/content/m44206/1.5/
Pages: 934-944

Module: "Java3010r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45774/1.1/
Pages: 945-954

Module: "Java3010s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45625/1.2/
Pages: 954-955

Module: "Java3012: Using Nested Loops to Process Pixels"
By: Richard Baldwin
URL: http://cnx.org/content/m44207/1.6/
Pages: 955-966

Module: "Java3012r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45775/1.1/
Pages: 967-974

Module: "Java3012s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45626/1.2/
Pages: 974-975
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3014: Cropping, Flipping, and Combining Pictures"
By: Richard Baldwin
URL: http://cnx.org/content/m44238/1.7/
Pages: 975-993
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3014r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45778/1.1/
Pages: 994-1002
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3014s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45628/1.3/
Pages: 1002-1003
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3016: Green-Screen Processing"
By: Richard Baldwin
URL: http://cnx.org/content/m44210/1.7/
Pages: 1003-1020
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3016r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45781/1.1/
Pages: 1021-1031
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3016s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45637/1.2/
Page: 1031
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3018: Darkening, Brightening, and Tinting the Colors in a Picture"
By: Richard Baldwin
URL: http://cnx.org/content/m44234/1.5/
Pages: 1031-1050
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3018r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45782/1.1/
Pages: 1051-1060

Module: "Java3018s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45636/1.2/
Pages: 1060-1061

Module: "Java3020: Interfaces, Object Arrays, etc."
By: Richard Baldwin
URL: http://cnx.org/content/m44214/1.5/
Pages: 1061-1075

Module: "Java3020s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45632/1.2/
Page: 1075

Module: "Java3022: Scaling, Rotating, and Translating Images using Affine Transforms"
By: Richard Baldwin
URL: http://cnx.org/content/m44223/1.6/
Pages: 1075-1091

Module: "Java3022r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45783/1.1/
Pages: 1092-1099

Module: "Java3022s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45639/1.2/
Pages: 1099-1100

Module: "Java3024: Mirroring Images"
By: Richard Baldwin
URL: http://cnx.org/content/m44228/1.6/
Pages: 1100-1114

Module: "Java3024r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45784/1.1/
Pages: 1115-1122
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3024s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45640/1.2/
Page: 1122
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3026: GradientPaint and other Java2D Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44242/1.6/
Pages: 1122-1139
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3026r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45785/1.1/
Pages: 1140-1146
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3026s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45643/1.2/
Pages: 1146-1147
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3028: Clipping Images"
By: Richard Baldwin
URL: http://cnx.org/content/m44246/1.6/
Pages: 1147-1158
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3028r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45786/1.1/
Pages: 1159-1166
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3028s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45646/1.2/
Page: 1166
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java3030: Merging Pictures"
By: Richard Baldwin
URL: http://cnx.org/content/m44247/1.6/
Pages: 1166-1178

Module: "Java3030r Review"
By: Richard Baldwin
URL: http://cnx.org/content/m45787/1.1/
Pages: 1179-1187

Module: "Java3030s Slides"
By: Richard Baldwin
URL: http://cnx.org/content/m45648/1.2/
Page: 1187

Module: "Java4010: Getting Started with Java Collections"
By: Richard Baldwin
URL: http://cnx.org/content/m46135/1.3/
Pages: 1188-1198

Module: "Java4020: What is a Collection"
By: Richard Baldwin
URL: http://cnx.org/content/m46136/1.2/
Pages: 1198-1203

Module: "Java4030: Purpose of Framework Interfaces"
By: Richard Baldwin
URL: http://cnx.org/content/m46140/1.2/
Pages: 1203-1208

Module: "Java4040: Purpose of Framework Implementations and Algorithms"
By: Richard Baldwin
URL: http://cnx.org/content/m46137/1.3/
Pages: 1208-1218

Module: "Java4050: Core Collection Interfaces"
By: Richard Baldwin
URL: http://cnx.org/content/m46138/1.2/
Pages: 1218-1227

Module: "Java4060: Duplicate Elements, Ordered Collections, Sorted Collections, and Interface Specialization"
By: Richard Baldwin
URL: http://cnx.org/content/m46141/1.2/
Pages: 1227-1233
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4070: The Comparable Interface, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m46142/1.2/
Pages: 1233-1246
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4080: The Comparable Interface, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m46143/1.1/
Pages: 1246-1257
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4090: The Comparator Interface, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m46189/1.1/
Pages: 1257-1268
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4100: The Comparator Interface, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m46190/1.1/
Pages: 1268-1277
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4110: The Comparator Interface, Part 3"
By: Richard Baldwin
URL: http://cnx.org/content/m46191/1.1/
Pages: 1277-1284
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4120: The Comparator Interface, Part 4"
By: Richard Baldwin
URL: http://cnx.org/content/m46192/1.1/
Pages: 1284-1296
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4130: The Comparator Interface, Part 5"
By: Richard Baldwin
URL: http://cnx.org/content/m46193/1.1/
Pages: 1296-1304
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4140: The Comparator Interface, Part 6"
By: Richard Baldwin
URL: http://cnx.org/content/m46194/1.1/
Pages: 1304-1313
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4150: The toArray Method, Part 1"
By: Richard Baldwin
URL: http://cnx.org/content/m46197/1.1/
Pages: 1313-1325
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java4160: The toArray Method, Part 2"
By: Richard Baldwin
URL: http://cnx.org/content/m46198/1.1/
Pages: 1325-1337
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: ITSE 2321 Practice Group 1"
By: Richard Baldwin
URL: http://cnx.org/content/m44252/1.7/
Pages: 1339-1353
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: ITSE 2321 Practice Group 2"
By: Richard Baldwin
URL: http://cnx.org/content/m44254/1.6/
Pages: 1354-1366
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: ITSE 2321 Practice Group 3"
By: Richard Baldwin
URL: http://cnx.org/content/m44255/1.4/
Pages: 1367-1376
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Jy0030: Java OOP: Preface to ITSE 2317"
By: Richard Baldwin
URL: http://cnx.org/content/m45258/1.1/
Pages: 1377-1378
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: The AWT and Swing, A Preview"
By: Richard Baldwin
URL: http://cnx.org/content/m44331/1.2/
Pages: 1378-1381
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Callbacks - I"
By: Richard Baldwin
URL: http://cnx.org/content/m44333/1.2/
Pages: 1381-1400
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Event Handling in JDK 1.1, A First Look, Delegation Event Model"
By: Richard Baldwin
URL: http://cnx.org/content/m44340/1.1/
Page: 1400
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Swing and the Delegation Event Model"
By: Richard Baldwin
URL: http://cnx.org/content/m44336/1.1/
Page: 1400
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Member Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44347/1.1/
Page: 1400
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Local Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44346/1.1/
Pages: 1400-1401
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Anonymous Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44342/1.1/
Page: 1401
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Modifying the World and SimpleTurtle Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44330/1.1/
Pages: 1401-1427
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Modifications to the Turtle and SimpleTurtle Classes"
By: Richard Baldwin
URL: http://cnx.org/content/m44348/1.1/
Pages: 1427-1449
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Incorporating GUI Components into a World Object"
By: Richard Baldwin
URL: http://cnx.org/content/m44350/1.1/
Pages: 1449-1464
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Background Color, Text Color, Mouse Clicks, etc."
By: Richard Baldwin
URL: http://cnx.org/content/m44351/1.1/
Pages: 1464-1497
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Panels, Labels, Text Fields, and Buttons"
By: Richard Baldwin
URL: http://cnx.org/content/m44352/1.2/
Pages: 1498-1510
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Using Alpha Transparency with Ericson's Media Library"
By: Richard Baldwin
URL: http://cnx.org/content/m44911/1.1/
Pages: 1510-1523
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Controlling Opacity with a Slider"
By: Richard Baldwin
URL: http://cnx.org/content/m44912/1.3/
Pages: 1523-1538
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Controlling an Edge Detector with a Slider"
By: Richard Baldwin
URL: http://cnx.org/content/m44913/1.2/
Pages: 1538-1549
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Controlling an Image-Scaling Program with a Slider"
By: Richard Baldwin
URL: http://cnx.org/content/m44914/1.2/
Pages: 1549-1559
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Controlling Image Rotation with a Slider and Affine Transforms"
By: Richard Baldwin
URL: http://cnx.org/content/m44915/1.3/
Pages: 1559-1575
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Opening an Image File in a PictureExplorer Object"
By: Richard Baldwin
URL: http://cnx.org/content/m44916/1.2/
Pages: 1575-1582
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Extracting pixel color data from a PictureExplorer object"
By: Richard Baldwin
URL: http://cnx.org/content/m44917/1.2/
Pages: 1582-1610
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Handling document events on a text field and creating a color swatch"
By: Richard Baldwin
URL: http://cnx.org/content/m44921/1.2/
Pages: 1610-1623
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: Using a JColorChooser object"
By: Richard Baldwin
URL: http://cnx.org/content/m44923/1.2/
Pages: 1623-1637
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: A Pixel Color Editor"
By: Richard Baldwin
URL: http://cnx.org/content/m44926/1.2/
Pages: 1637-1671
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: ITSE 2317 Practice Test 1"
By: Richard Baldwin
URL: http://cnx.org/content/m44264/1.2/
Pages: 1673-1686
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java OOP: ITSE 2317 Practice Test 2"
By: Richard Baldwin
URL: http://cnx.org/content/m44265/1.3/
Pages: 1687-1702
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Object-Oriented Programming (OOP) with Java
Teaching material for various Object-Oriented Programming (OOP) courses at Austin Community College in Austin, TX.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.