Java Fundamentals I and II

Based on *Java How to Program, Seventh Edition*
http://www.deitel.com/books/jhtp7/

Paul J. Deitel, CEO
Deitel® & Associates, Inc.
Sun Certified Java Programmer
Sun Certified Java Developer
Sun Java Champion

Java™ Champions

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data On file

Visit us on the Web: www.informit.com/ph

Corporate and Government Sales

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States please contact: International Sales, international@pearsoned.com.

Warning and Disclaimer

This book and video product is designed to provide information about Java programming. Every effort has been made to make it as complete and as accurate as possible, but no warranty or fitness is implied. The information is provided on an “as is” basis. The author and Prentice Hall shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the disc or programs that may accompany it. The opinions expressed in this LiveLesson belong to the author and are not necessarily those of Prentice Hall.

Feedback Information

At Pearson, our goal is to create in-depth technical books of the highest quality and value. Each product is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community. Readers’ feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise alter it to better suit your needs, you can contact us through e-mail at mylivelessons@pearsoned.com. Please make sure to include the title and ISBN in your message.

We greatly appreciate your assistance.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Neither Prentice Hall nor Pearson Education, Inc., can attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

ISBN-13: 978-0-13-713113-6
ISBN-10: 0-13-713113-5

Text printed in the United States at R.R. Donnelley in Crawfordsville, Indiana.
First printing, March 2008

Publisher
Paul Boger

Editor-in-Chief
Mark L. Taub

Editorial Assistant
Noreen Regina

Managing Editor
John Fuller

Project Editor
Julie B. Nahil

Copy Editor
Joe Ruddick

Multimedia Developer
Eric Strom, Pearson Video Production Services

Designer
Gary Adair

Contents

Preface

Welcome to the *Java Fundamentals LiveLessons*. This two-part sequence presents object-oriented programming in Java using Java Standard Edition (Java SE) 6. After studying the fundamental topics presented here, you'll have the foundation you need to learn more about Java programming on the Java Micro Edition (Java ME), Java Standard Edition (Java SE) and Java Enterprise Edition (Java EE) platforms.

What You Will Learn

- Features of Java Standard Edition (Java SE) 6.
- To build, compile and run applications with the Java Development Kit (JDK), as well as the popular Eclipse and Netbeans integrated development environments (IDEs).
- How the Java virtual machine (JVM) makes Java applications portable.
- To use the classes and interfaces in the Java Applications Programming Interface (API) and to use the Java online documentation to locate the features you need in the Java API.
- To use formatted input and output capabilities.
- Arithmetic, increment, decrement, assignment, relational, equality and logical operators.
- Control statements.
- Primitive types and their corresponding Java API classes.
- Methods, method overloading and methods with variable-length argument lists.
- Arrays and strings, and how they are manipulated as Java objects.
- Object-oriented programming concepts including classes, objects, encapsulation, interfaces, inheritance, polymorphism, `abstract` classes and `abstract` methods.
- To use and create your own `static` methods and `static` variables.
- To package your own classes to create reusable class libraries.
- The fundamentals of event-driven graphical user interface (GUI) programming and Swing GUI components including windows, buttons, labels, comboboxes, text fields and panels.
- To use exception handling to make your programs more robust.
- The classes, interfaces and algorithms of the Java Collections Framework (Java's data structures and algorithms for manipulating them).

Who Should Use These LiveLessons?

The *Java Fundamentals LiveLessons* are intended for students and professionals who are already familiar with programming fundamentals in a high-level programming language such as C, C++, C# or Visual Basic. Object-oriented programming experience is not required—this is a key focus of these *LiveLessons*. *Java Fundamentals: Part 1* introduces object-oriented programming fundamentals in Java. *Java Fundamentals: Part 2* continues with in-depth discussions of object-oriented programming and introduces GUIs, exception handling and the Java Collections Framework.

Teaching Approach

In the *Java Fundamentals LiveLessons*, author Paul Deitel concentrates on the principles of good software engineering and stresses program clarity, teaching by example. Paul is a professional trainer who presents leading-edge courses to government, industry, the military and academia.

Live-Code Approach. These *LiveLessons* are loaded with "live-code" examples—that is, each new concept is presented in the context of a complete working Java application. In addition to discussing the new concepts in each application, I execute the application so you can see the concepts "in action." This style exemplifies the way we teach and write about programming at Deitel & Associates; we call this the "live-code" approach.

Object-Oriented Throughout. After learning some Java fundamentals in *Lesson 1* of *Java Fundamentals: Part 1*, you'll create your first customized classes and objects in *Lesson 2*. Presenting objects and classes early gets you "thinking about objects" immediately and mastering these concepts more thoroughly. We then use these concepts throughout both *LiveLessons*.

Online Documentation. Throughout both *LiveLessons* we show you how to use Java's online documentation. This will help you avoid "reinventing the wheel" by locating features that are already defined in the Java API and that you need in your applications. It will also help you learn the relationships among many key classes and interfaces in the Java API. Learning these relationships is essential to taking full advantage of the Java API's capabilities.

How These LiveLessons Are Organized

These *LiveLessons* are based on portions of Paul's best-selling computer science textbook and professional book *Java How to Program, 7/e* (`www.deitel.com/books/jhtp7/`) and his *Dive-Into® Series* corporate training courses (`www.deitel.com/training/`), which he presents to organizations worldwide. Feel free to email Paul at `deitel@deitel.com`.

Java Fundamentals: Part 1 (40 examples)

- **Lesson 1, Introduction to Java Applications,** introduces Java application programming. You'll learn formatted input and output capabilities, and how to compile and run Java applications using an IDE and using the JDK command-line tools. You'll also begin using the packages of reusable classes in the Java class libraries.
- **Lesson 2, Introduction to Classes and Objects,** introduces object-oriented programming principles and constructs, and begins our case study on developing a grade-book class that instructors can use to maintain student test scores. This case study is enhanced over the next several lessons, culminating with the versions presented in Lesson 6, Arrays. The last example in this lesson uses a bank account class to introduce data validation concepts. In this lesson, you'll learn what classes, objects, methods and instance variables are; how to declare a class and use it to create an object; how to declare methods in a class to implement the class's behaviors; how to declare instance variables in a class to implement the class's attributes; how to call an object's methods to make them perform their tasks; the differences between instance variables of a class and local variables of a method; to use a constructor to ensure that an object's data is initialized when the object is created; and the differences between primitive and reference types.
- **Lesson 3, Control Statements: Part 1,** continues enhancing the `GradeBook` case study with additional functionality. You'll learn Java's `if`, `if...else` and `while` control statements, and the increment and decrement operators.
- **Lesson 4, Control Statements: Part 2,** introduces Java's `for` and `do...while` repetition statements, and the `switch` multiple-selection statement. A portion of this lesson expands

the `GradeBook` class presented in Lessons 2–3 by using a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric grades entered by the user.

- **Lesson 5, Methods: A Deeper Look,** discusses other details of method definitions. You'll also learn about `static` methods and fields of a class; Java's eight primitive types and the implicit type promotion rules between them; some common packages in Java; random-number generation; how to create and use named constants; the scope of identifiers; and what method overloading is and how to create overloaded methods.
- **Lesson 6, Arrays,** introduces Java's implementation arrays. You'll learn how to declare, initialize and manipulate arrays; to iterate through arrays with the enhanced `for` statement; to pass arrays to methods; to declare and manipulate multidimensional arrays; to create methods with variable-length argument lists; and to read a program's command-line arguments. We'll also enhance the `GradeBook` case study using arrays to maintain a set of grades in memory and analyze student grades from multiple exams in a semester.
- **Lesson 7, Classes and Objects: A Deeper Look,** takes a deeper look at building classes, controlling access to class members and creating constructors. The examples teach encapsulation and data hiding; composition; how to use keyword `this` to refer to an object's members; how to create `static` variables and methods; how to import `static` class members; how to use the `enum` type to create sets of named constants that can be initialized with arguments; and how to organize classes into your own packages for reusability.

Java Fundamentals: Part 2 (29 examples)

- **Lesson 1, Object-Oriented Programming: Inheritance,** discusses the object-oriented programming (OOP) concept of inheritance—a form of software reuse in which a new class absorbs an existing class's members and embellishes them with new or modified capabilities. You'll learn how inheritance promotes software reusability; the notions of superclasses and subclasses; to use keyword `extends` to create a class that inherits from another class; to use access modifier `protected` to give subclass methods access to superclass members; to access superclass members with keyword `super`; how constructors are used in inheritance hierarchies; and the methods of class `Object`—the direct or indirect superclass of all classes in Java.
- **Lesson 2, Object-Oriented Programming: Polymorphism,** introduces the OOP concept of polymorphism, which enables programs to process objects which share the same superclass in a class hierarchy as if they are all objects of the superclass. You'll learn the concept of polymorphism; how to use overridden methods to effect polymorphism; to distinguish between `abstract` and concrete classes; to declare `abstract` methods; how polymorphism makes systems extensible and maintainable; to determine an object's type at execution time; and to declare and implement interfaces—objects of classes that implement the same interface can respond polymorphically to the same method calls.
- **Lesson 3, Introduction to Graphical User Interfaces (GUIs) and Event Handling,** shows how to build Swing GUIs and respond to user interactions. You'll also learn to create and use nested classes and anonymous inner classes; the packages containing GUI components, event-handling classes and interfaces; to create and manipulate several types of GUI components; and to handle mouse events. As you'll see, the polymorphism and interface concepts presented in Lesson 2 are used frequently in programs with GUIs.

- **Lesson 4, Exception Handling,** introduces features that enable you to write robust and fault-tolerant programs. You'll learn to use `try`, `throw` and `catch` to detect, indicate and handle exceptions, respectively; to use the `finally` block to release resources; how stack unwinding enables exceptions not caught in one scope to be caught in another scope; how stack traces help in debugging; to use the online documentation to determine the exceptions thrown by a method; to use the exception class hierarchy to distinguish between checked and unchecked exceptions; and to create chained exceptions.
- **Lesson 5, The Java Collections Framework,** discusses Java's prepackaged data structures, interfaces and algorithms. You'll learn what collections are; the common array manipulations of class `Arrays`; to use the collections framework implementations; to use the collections framework algorithms to manipulate collections; to use the collections framework interfaces to polymorphically manipulate collections; and to use iterators to "walk through" a collection (with methods of iterator objects and with the enhanced `for` statement). You'll also learn about the synchronization and modifiability wrappers for collections.

Playing the DVD

All the example programs presented in these LiveLessons are included on the DVD in the folder `extras` at the DVD's root level. The DVD will run on both Windows and Mac OS/X systems.

- If your Windows system is configured for AutoPlay, the video will start playing when you insert the DVD. If AutoPlay is off, you'll need to insert the DVD, launch Windows Explorer, navigate to the DVD's root folder and double-click the file `Start_livelesson.exe`.
- On the Mac, the *LiveLesson* application icon will appear in a Finder window when you insert the DVD. Simply click the icon to launch the *LiveLesson*.

This *LiveLessons* product is designed to run at a screen resolution of 1280 × 800 or higher. Please adjust your screen resolution for the best playback experience. The following are the system requirements for this DVD.

- Operating system: Windows 2000, XP, Vista, or Mac OS/X
- Computer: 500MHz or higher, 128MB RAM or more
- Multimedia: DVD drive, 1280 × 1024 or higher display, and sound card with speakers

About the Author

Paul J. Deitel, CEO, Chief Technical Officer and co-founder of Deitel & Associates, Inc., is a graduate of MIT's Sloan School of Management, where he studied Information Technology. He holds the Sun Certified Java Programmer and Sun Certified Java Developer certifications, and has been designated by Sun Microsystems as a Sun Java Champion. Through Deitel & Associates, Inc., he has delivered Java, C, C++, C#, Visual Basic and web-application development courses to industry clients, including Cisco, IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He has also lectured on Java and C++ for the Boston Chapter of the Association for Computing Machinery. He and his father, Dr. Harvey M. Deitel, are the world's best-selling programming language textbook authors.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized corporate training and content-creation organization specializing in computer programming languages, Internet and web software technology, object technology education and Internet business development through its Web 2.0 Internet Business Initiative. The company provides instructor-led courses on major programming languages and platforms, such as C++, Java, C, C#, Visual C++, Visual Basic, XML, object technology and Internet and web programming. The founders of Deitel & Associates, Inc., Paul J. Deitel and Dr. Harvey M. Deitel. The company's clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. Through its 31-year publishing partnership with the Prentice Hall and Addison Wesley imprints of Pearson Education, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive multimedia *Cyber Classrooms*, Web-based training courses, online and offline video courses, and e-content for the popular course management systems WebCT, Blackboard and Pearson's CourseCompass. Deitel & Associates, Inc., and Paul Deitel can be reached via e-mail at:

`deitel@deitel.com`

To learn more about Deitel & Associates, Inc., its publications and its worldwide *Dive Into*® Series Corporate Training curriculum, visit:

`www.deitel.com`

and subscribe to the free *Deitel*® *Buzz Online* e-mail newsletter at:

`www.deitel.com/newsletter/subscribe.html`

Check out the growing list of Java and related online Deitel Resource Centers at:

`www.deitel.com/ResourceCenters.html`

Individuals wishing to purchase Deitel publications can do so at:

`www.deitel.com/books/index.html`

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Prentice Hall. For more information, visit

`www.prenhall.com/mischtm/support.html#order`

lesson 1

Introduction to Java Applications

Based on Chapter 2 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`).

Learning Objectives

- Write simple Java applications.
- Use command-line input and output statements.
- Use output formatting capabilities.
- Understand the fundamentals of compiling and running Java applications from an Integrated Development Environment (IDE) and from the command line.
- Begin using the packages in the Java class libraries—also called the Java APIs (application programming interfaces).

Figures

```java
// Fig. 2.1: Welcome1.java
// Text-printing program.

public class Welcome1
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      System.out.println( "Welcome to Java Programming!" );

   } // end method main

} // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program.

```java
// Fig. 2.15: Comparisona.java
// Compare integers using if statements, relational operators
// and equality operators.
import java.util.Scanner; // program uses class Scanner

public class Comparison
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int number1; // first number to compare
      int number2; // second number to compare

      System.out.print( "Enter first integer: " ); // prompt
      number1 = input.nextInt(); // read first number from user

      System.out.print( "Enter second integer: " ); // prompt
      number2 = input.nextInt(); // read second number from user

      if ( number1 == number2 )
         System.out.printf( "%d == %d\n", number1, number2 );

         ( number1 != number2 )
         System.out.printf( "%d != %d\n", number1, number2 );

         ( number1 < number2 )
         System.out.printf( "%d < %d\n", number1, number2 );

         ( number1 > number2 )
         System.out.printf( "%d > %d\n", number1, number2 );

         ( number1 <= number2 )
         System.out.printf( "%d <= %d\n", number1, number2 );

         ( number1 >= number2 )
         System.out.printf( "%d >= %d\n", number1, number2 );

   } // end method main

} // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

Fig. 2.15 Equality and relational operators. (Part 1 of 2.)

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Equality and relational operators. (Part 2 of 2.)

2 lesson

Introduction to Classes and Objects

Based on Chapter 3 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`).

Learning Objectives

- Understand what classes, objects, methods and instance variables are.
- Declare a class and use it to create an object.
- Declare methods in a class to implement the class's behaviors.
- Declare instance variables in a class to implement the class's attributes.
- Call an object's methods to make those methods perform their tasks.
- Understand the differences between instance variables of a class and local variables of a method.
- Use a constructor to ensure that an object's data is initialized when the object is created.
- Understand the differences between primitive and reference types.

Figures

```
// Fig 3.1: GradeBook.java
// Class declaration with one method.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      System.out.println( "Welcome to the Grade Book!" );
   } // end method displayMessage

} // end class GradeBook
```

Fig. 3.1 | Class declaration with one method.

```java
// Fig 3.2: GradeBookTest.java
// Create a GradeBook object and call its displayMessage method.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String args[] )
   {
      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // call myGradeBook's displayMessage method
      myGradeBook.displayMessage();
   } // end main

} // end class GradeBookTest
```

```
Welcome to the Grade Book!
```

Fig. 3.2 | Creating an object of class `GradeBook` and calling its `displayMessage` method.

```java
// Fig 3.4: GradeBook.java
// Class declaration with a method that has a parameter.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage( String courseName )
   {
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         courseName );
   } // end method displayMessage

} // end class GradeBook
```

Fig. 3.4 | Class declaration with one method that has a parameter.

```java
// Fig 3.5: GradeBookTest.java
// Create GradeBook object and pass a String to
// its displayMessage method.
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String args[] )
   {
```

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 1 of 2.)

```
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // prompt for and input course name
      System.out.println( "Please enter the course name:" );
      String nameOfCourse = input.nextLine(); // read a line of text
      System.out.println(); // outputs a blank line

      // call myGradeBook's displayMessage method
      // and pass nameOfCourse as an argument
      myGradeBook.displayMessage( nameOfCourse );
   } // end main

} // end class GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 2 of 2.)

```
// Fig 3.7: GradeBook.java
// GradeBook class that contains a courseName instance variable
// and methods to set and get its value.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
```

Fig. 3.7 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value. (Part 1 of 2.)

```java
        // this statement calls getCourseName to get the
        // name of the course this GradeBook represents
        System.out.printf( "Welcome to the grade book for\n%s!\n",
           getCourseName() );
     } // end method displayMessage

  } // end class GradeBook
```

Fig. 3.7 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value. (Part 2 of 2.)

```java
  // Fig 3.8: GradeBookTest.java
  // Create and manipulate a GradeBook object.
  import java.util.Scanner; // program uses Scanner

  public class GradeBookTest
  {
     // main method begins program execution
     public static void main( String args[] )
     {
        // create Scanner to obtain input from command window
        Scanner input = new Scanner( System.in );

        // create a GradeBook object and assign it to myGradeBook
        GradeBook myGradeBook = new GradeBook();

        // display initial value of courseName
        System.out.printf( "Initial course name is: %s\n\n",
           myGradeBook.getCourseName() );

        // prompt for and read course name
        System.out.println( "Please enter the course name:" );
        String theName = input.nextLine(); // read a line of text
        myGradeBook.setCourseName( theName ); // set the course name
        System.out.println(); // outputs a blank line

        // display welcome message after specifying course name
        myGradeBook.displayMessage();
     } // end main

  } // end class GradeBookTest
```

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object.

```
// Fig 3.10: GradeBook.java
// GradeBook class wsith a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // constructor initializes courseName with String supplied as argument
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // this statement calls getCourseName to get the
      // name of the course this GradeBook represents
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         getCourseName() );
   } // end method displayMessage

} // end class GradeBook
```

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name.

```
// Fig 3.11: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String args[] )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
```

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created. (Part 1 of 2.)

```java
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main

} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created. (Part 2 of 2.)

```java
// Fig 3.13: Account.java
// Account class with a constructor to
// initialize instance variable balance.

public class Account
{
   private double balance; // instance variable that stores the balance

   // constructor
   public Account( double initialBalance )
   {
      // validate that initialBalance is greater than 0.0;
      // if it is not, balance is initialized to the default value 0.0
      if ( initialBalance > 0.0 )
         balance = initialBalance;
   } // end Account constructor

   // credit (add) an amount to the account
   public void credit( double amount )
   {
      balance = balance + amount; // add amount to balance
   } // end method credit

   // return the account balance
   public double getBalance()
   {
      return balance; // gives the value of balance to the calling method
   } // end method getBalance

} // end class Account
```

Fig. 3.13 | `Account` class with an instance variable of type `double`.

```java
// Fig 3.14: AccountTest.java
// Inputting and outputting floating-point numbers with Account objects.
import java.util.Scanner;

public class AccountTest
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      Account account1 = new Account( 50.00 ); // create Account object
      Account account2 = new Account( -7.53 ); // create Account object

      // display initial balance of each object
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );
      double depositAmount; // deposit amount read from user

      System.out.print( "Enter deposit amount for account1: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account1 balance\n\n",
         depositAmount );
      account1.credit( depositAmount ); // add to account1 balance

      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      System.out.print( "Enter deposit amount for account2: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account2 balance\n\n",
         depositAmount );
      account2.credit( depositAmount ); // add to account2 balance

      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n",
         account2.getBalance() );
   } // end main

} // end class AccountTest
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 2.)

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 2.)

3 lesson

Control Statements: Part 1

Based on Chapter 4 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`).

Learning Objectives

- Continue enhancing the `GradeBook` case study with additional functionality.
- Use Java's `if`, `if…else` and `while` control statements.
- Use the prefix-increment, prefix-decrement, postfix-increment and postfix-decrement operators.

Figures

```java
// Fig. 4.6: GradeBook.java
// GradeBook class that solves class-average problem using
// counter-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName
```

Fig. 4.6 | Counter-controlled repetition: Class-average problem. (Part 1 of 2.)

```java

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine class average based on 10 grades entered by user
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades entered by user
      int gradeCounter; // number of the grade to be entered next
      int grade; // grade value entered by user
      int average; // average of grades

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 1; // initialize loop counter

      // processing phase
      while ( gradeCounter <= 10 ) // loop 10 times
      {
         System.out.print( "Enter grade: " ); // prompt
         grade = input.nextInt(); // input next grade
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter by 1
      } // end while

      // termination phase
      average = total / 10; // integer division yields integer result

      // display total and average of grades
      System.out.printf( "\nTotal of all 10 grades is %d\n", total );
      System.out.printf( "Class average is %d\n", average );
   } // end method determineClassAverage

} // end class GradeBook
```

Fig. 4.6 | Counter-controlled repetition: Class-average problem. (Part 2 of 2.)

```java
// Fig. 4.7: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
```

Fig. 4.7 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method. (Part 1 of 2.)

```java
    public static void main( String args[] )
    {
       // create GradeBook object myGradeBook and
       // pass course name to constructor
       GradeBook myGradeBook = new GradeBook(
          "CS101 Introduction to Java Programming" );

       myGradeBook.displayMessage(); // display welcome message
       myGradeBook.determineClassAverage(); // find average of 10 grades
    } // end main

 } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 4.7 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method. (Part 2 of 2.)

```java
// Fig. 4.9: GradeBook.java
// GradeBook class that solves class-average program using
// sentinel-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
```

Fig. 4.9 | Sentinel-controlled repetition: Class-average problem. (Part 1 of 3.)

```
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine the average of an arbitrary number of grades
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades
      int gradeCounter; // number of grades entered
      int grade; // grade value
      double average; // number with decimal point for average

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 0; // initialize loop counter

      // processing phase
      // prompt for input and read grade from user
      System.out.print( "Enter grade or -1 to quit: " );
      grade = input.nextInt();

      // loop until sentinel value read from user
      while ( grade != -1 )
      {
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter

         // prompt for input and read next grade from user
         System.out.print( "Enter grade or -1 to quit: " );
         grade = input.nextInt();
      } // end while

      // termination phase
      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
```

Fig. 4.9 | Sentinel-controlled repetition: Class-average problem. (Part 2 of 3.)

```
      // calculate average of all grades entered
      average = (double) total / gradeCounter;

      // display total and average (with two digits of precision)
      System.out.printf( "\nTotal of the %d grades entered is %d\n",
         gradeCounter, total );
      System.out.printf( "Class average is %.2f\n", average );
   } // end if
   else // no grades were entered, so output appropriate message
      System.out.println( "No grades were entered" );
} // end method determineClassAverage

} // end class GradeBook
```

Fig. 4.9 | Sentinel-controlled repetition: Class-average problem. (Part 3 of 3.)

```
// Fig. 4.10: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
   public static void main( String args[] )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.determineClassAverage(); // find average of grades
   } // end main

} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 4.10 | `GradeBookTest` class creates an object of class `GradeBook` and invokes its `determineClassAverage` method.

```java
// Fig. 4.16: Increment.java
// Prefix increment and postfix increment operators.

public class Increment
{
   public static void main( String args[] )
   {
      int c;

      // demonstrate postfix increment operator
      c = 5; // assign 5 to c
      System.out.println( c );   // prints 5
      System.out.println( c++ ); // prints 5 then postincrements
      System.out.println( c );   // prints 6

      System.out.println(); // skip a line

      // demonstrate prefix increment operator
      c = 5; // assign 5 to c
      System.out.println( c );   // prints 5
      System.out.println( ++c ); // preincrements then prints 6
      System.out.println( c );   // prints 6

   } // end main

} // end class Increment
```

```
5
5
6

5
6
6
```

Fig. 4.16 | Preincrementing and postincrementing.

4 lesson

Control Statements: Part 2

Based on Chapter 5 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Use the `for` and `do…while` repetition statements.
- Use the `switch` multiple-selection statement.
- Use the end-of-file indicator to determine when user input is complete.
- Learn the differences between the conditional operators and the boolean logical operators that are used to form complex conditional expressions in control statements.

Figures

```java
// Fig. 5.6: Interest.java
// Compound-interest calculations with for.

public class Interest
{
   public static void main( String args[] )
   {
      double amount; // amount on deposit at end of each year
      double principal = 1000.0; // initial amount before interest
      double rate = 0.05; // interest rate

      // display headers
      System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );

      // calculate amount on deposit for each of ten years
      for ( int year = 1; year <= 10; year++ )
      {
         // calculate new amount for specified year
         amount = principal * Math.pow( 1.0 + rate, year );

         // display the year and the amount
         System.out.printf( "%4d%,20.2f\n", year, amount );
      } // end for
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
    } // end main
} // end class Interest
```

```
Year   Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

```
// Fig. 5.7: DoWhileTest.java
// do...while repetition statement.

public class DoWhileTest
{
   public static void main( String args[] )
   {
      int counter = 1; // initialize counter

      do
      {
         System.out.printf( "%d  ", counter );
         ++counter;
      } while ( counter <= 10 ); // end do...while

      System.out.println(); // outputs a newline
   } // end main
} // end class DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | `do...while` repetition statement.

```
// Fig. 5.9: GradeBook.java
// GradeBook class uses switch statement to count A, B, C, D and F grades.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents
   private int total; // sum of grades
   private int gradeCounter; // number of grades entered
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count A, B, C, D and F grades. (Part 1 of 4.)

```
   private int aCount; // count of A grades
   private int bCount; // count of B grades
   private int cCount; // count of C grades
   private int dCount; // count of D grades
   private int fCount; // count of F grades

   // constructor initializes courseName;
   // int instance variables are initialized to 0 by default
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // input arbitrary number of grades from user
   public void inputGrades()
   {
      Scanner input = new Scanner( System.in );

      int grade; // grade entered by user

      System.out.printf( "%s\n%s\n   %s\n   %s\n",
         "Enter the integer grades in the range 0-100.",
         "Type the end-of-file indicator to terminate input:",
         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
         "On Windows type <ctrl> z then press Enter" );

      // loop until user enters the end-of-file indicator
      while ( input.hasNext() )
      {
         grade = input.nextInt(); // read grade
         total += grade; // add grade to total
         ++gradeCounter; // increment number of grades
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count A, B, C, D and F grades. (Part 2 of 4.)

```

         // call method to increment appropriate counter
         incrementLetterGradeCounter( grade );
      } // end while
   } // end method inputGrades

   // add 1 to appropriate counter for specified grade
   public void incrementLetterGradeCounter( int Grade )
   {
      // determine which grade was entered
      switch ( grade / 10 )
      {
         case 9:  // grade was between 90
         case 10: // and 100
            ++aCount; // increment aCount
            break; // necessary to exit switch

         case 8: // grade was between 80 and 89
            ++bCount; // increment bCount
            break; // exit switch

         case 7: // grade was between 70 and 79
            ++cCount; // increment cCount
            break; // exit switch

         case 6: // grade was between 60 and 69
            ++dCount; // increment dCount
            break; // exit switch

         default: // grade was less than 60
            ++fCount; // increment fCount
            break; // optional; will exit switch anyway
      } // end switch
   } // end method incrementLetterGradeCounter

   // display a report based on the grades entered by user
   public void displayGradeReport()
   {
      System.out.println( "\nGrade Report:" );

      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
         // calculate average of all grades entered
         double average = (double) total / gradeCounter;

         // output summary of results
         System.out.printf( "Total of the %d grades entered is %d\n",
            gradeCounter, total );
         System.out.printf( "Class average is %.2f\n", average );
         System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
            "Number of students who received each grade:",
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count A, B, C, D and F grades. (Part 3 of 4.)

```
            "A: ", aCount,   // display number of A grades
            "B: ", bCount,   // display number of B grades
            "C: ", cCount,   // display number of C grades
            "D: ", dCount,   // display number of D grades
            "F: ", fCount ); // display number of F grades
      } // end if
      else // no grades were entered, so output appropriate message
         System.out.println( "No grades were entered" );
   } // end method displayGradeReport
} // end class GradeBook
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count A, B, C, D and F grades. (Part 4 of 4.)

```
// Fig. 5.10: GradeBookTest.java
// Create GradeBook object, input grades and display grade report.

public class GradeBookTest
{
   public static void main( String args[] )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.inputGrades(); // read grades from user
      myGradeBook.displayGradeReport(); // display report based on grades
   } // end main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
   On Windows type <ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z
```

Fig. 5.10 | `GradeBookTest` creates a `GradeBook` object and invokes its methods. (Part 1 of 2.)

```
Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.10 | `GradeBookTest` creates a `GradeBook` object and invokes its methods. (Part 2 of 2.)

```java
// Fig. 5.18: LogicalOperators.java
// Logical operators.

public class LogicalOperators
{
   public static void main( String args[] )
   {
      // create truth table for && (conditional AND) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Conditional AND (&&)", "false && false", ( false && false ),
         "false && true", ( false && true ),
         "true && false", ( true && false ),
         "true && true", ( true && true ) );

      // create truth table for || (conditional OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Conditional OR (||)", "false || false", ( false || false ),
         "false || true", ( false || true ),
         "true || false", ( true || false ),
         "true || true", ( true || true ) );

      // create truth table for & (boolean logical AND) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical AND (&)", "false & false", ( false & false ),
         "false & true", ( false & true ),
         "true & false", ( true & false ),
         "true & true", ( true & true ) );

      // create truth table for | (boolean logical inclusive OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical inclusive OR (|)",
         "false | false", ( false | false ),
         "false | true", ( false | true ),
         "true | false", ( true | false ),
         "true | true", ( true | true ) );

      // create truth table for ^ (boolean logical exclusive OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical exclusive OR (^)",
```

Fig. 5.18 | Logical operators. (Part 1 of 2.)

```
         "false ^ false", ( false ^ false ),
         "false ^ true", ( false ^ true ),
         "true ^ false", ( true ^ false ),
         "true ^ true", ( true ^ true ) );

      // create truth table for ! (logical negation) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
         "!false", ( !false ), "!true", ( !true ) );
   } // end main
} // end class LogicalOperators
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Fig. 5.18 | Logical operators. (Part 2 of 2.)

lesson 5

Methods: A Deeper Look

Based on Chapter 6 of *Java How to Program, 7/e* (http://www.deitel.com/books/jhtp7/)

Learning Objectives

- Understand how `static` methods and fields are associated with an entire class rather than specific instances of the class.
- Learn the eight primitive types in Java and the implicit type promotion rules between them.
- Learn about some common packages in Java.
- Use random-number generation to implement game-playing applications.
- Create and use the named constants of a simple `enum` type.
- Learn the scope of identifiers.
- Learn what method overloading is and how to create overloaded methods.

Figures

Type	Valid promotions
`double`	None
`float`	double
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. 6.5 | Promotions allowed for primitive types.

Package	Description
`java.applet`	The ***Java Applet Package*** contains a class and several interfaces required to create Java applets—programs that execute in Web browsers.
`java.awt`	The ***Java Abstract Window Toolkit Package*** contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead.
`java.awt.event`	The ***Java Abstract Window Toolkit Event Package*** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages.
`java.io`	The ***Java Input/Output Package*** contains classes and interfaces that enable programs to input and output data.
`java.lang`	The ***Java Language Package*** contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so you do not need to do so.
`java.net`	The ***Java Networking Package*** contains classes and interfaces that enable programs to communicate via computer networks like the Internet.
`java.text`	The ***Java Text Package*** contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The ***Java Utilities Package*** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`).
`javax.swing`	The ***Java Swing GUI Components Package*** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.
`javax.swing.event`	The ***Java Swing Event Package*** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`.

Fig. 6.6 | Java API packages (a subset).

```
// Fig. 6.9: Craps.java
// Craps class simulates the dice game craps.
import java.util.Random;

public class Craps
{
   // create random number generator for use in method rollDice
   private Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST };

   // constants that represent common rolls of the dice
   private final static int SNAKE_EYES = 2;
   private final static int TREY = 3;
   private final static int SEVEN = 7;
   private final static int YO_LEVEN = 11;
   private final static int BOX_CARS = 12;

   // plays one game of craps
   public void play()
   {
      int myPoint = 0; // point if no win or loss on first roll
      Status gameStatus; // can contain CONTINUE, WON or LOST

      int sumOfDice = rollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch ( sumOfDice )
      {
         case SEVEN: // win with 7 on first roll
         case YO_LEVEN: // win with 11 on first roll
            gameStatus = Status.WON;
            break;
         case SNAKE_EYES: // lose with 2 on first roll
         case TREY: // lose with 3 on first roll
         case BOX_CARS: // lose with 12 on first roll
            gameStatus = Status.LOST;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf( "Point is %d\n", myPoint );
            break; // optional at end of switch
      } // end switch

      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
      {
         sumOfDice = rollDice(); // roll dice again

```

Fig. 6.9 | Craps class simulates the dice game craps. (Part 1 of 2.)

```
         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
               gameStatus = Status.LOST;
      } // end while

      // display won or lost message
      if ( gameStatus == Status.WON )
         System.out.println( "Player wins" );
      else
         System.out.println( "Player loses" );
   } // end method play

   // roll dice, calculate sum and display results
   public int rollDice()
   {
      // pick random die values
      int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
      int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll

      int sum = die1 + die2; // sum of die values

      // display results of this roll
      System.out.printf( "Player rolled %d + %d = %d\n",
         die1, die2, sum );

      return sum; // return sum of dice
   } // end method rollDice
} // end class Craps
```

Fig. 6.9 | `Craps` class simulates the dice game craps. (Part 2 of 2.)

```
// Fig. 6.10: CrapsTest.java
// Application to test class Craps.

public class CrapsTest
{
   public static void main( String args[] )
   {
      Craps game = new Craps();
      game.play(); // play one game of craps
   } // end main
} // end class CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

Fig. 6.10 | Application to test class `Craps`. (Part 1 of 2.)

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Fig. 6.10 | Application to test class `Craps`. (Part 2 of 2.)

```java
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin is %d\n", x );

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf( "\nlocal x in method begin is %d\n", x );
   } // end method begin

   // create and initialize local variable x during each call
   public void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

```

Fig. 6.11 | `Scope` class demonstrating scopes of a field and local variables. (Part 1 of 2.)

```
      System.out.printf(
         "\nlocal x on entering method useLocalVariable is %d\n", x );
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d\n", x );
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public void useField()
   {
      System.out.printf(
         "\nfield x on entering method useField is %d\n", x );
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d\n", x );
   } // end method useField
} // end class Scope
```

Fig. 6.11 | Scope class demonstrating scopes of a field and local variables. (Part 2 of 2.)

```
// Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String args[] )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // end class ScopeTest
```

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Fig. 6.12 | Application to test class Scope.

```
// Fig. 6.13: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public void testOverloadedMethods()
   {
      System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
      System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
   } // end method testOverloadedMethods

   // square method with int argument
   public int square( int intValue )
   {
      System.out.printf( "\nCalled square with int argument: %d\n",
         intValue );
      return intValue * intValue;
   } // end method square with int argument

   // square method with double argument
   public double square( double doubleValue )
   {
      System.out.printf( "\nCalled square with double argument: %f\n",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method square with double argument
} // end class MethodOverload
```

Fig. 6.13 | Overloaded method declarations.

```
// Fig. 6.14: MethodOverloadTest.java
// Application to test class MethodOverload.

public class MethodOverloadTest
{
   public static void main( String args[] )
   {
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.testOverloadedMethods();
   } // end main
} // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.14 | Application to test class `MethodOverload`.

```java
// Fig. 6.15: MethodOverloadError.java
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.

public class MethodOverloadError
{
   // declaration of method square with int argument
   public int square( int x )
   {
      return x * x;
   }

   // second declaration of method square with int argument
   // causes compilation error even though return types are different
   public double square( int y )
   {
      return y * y;
   }
} // end class MethodOverloadError
```

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Fig. 6.15 | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different.

lesson 6

Arrays

Based on Chapter 7 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`).

Learning Objectives

- Declare arrays, initialize them and refer to their individual elements.
- Use the enhanced `for` statement to iterate through arrays.
- Pass arrays to methods.
- Declare and manipulate multidimensional arrays.
- Create methods that use variable-length argument lists.
- Read command-line arguments into a program.

Figures

```
// Fig. 7.2: InitArray.java
// Creating an array.

public class InitArray
{
   public static void main( String args[] )
   {
      int array[]; // declare array named array

      array = new int[ 10 ]; // create the space for array

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

Fig. 7.2 | Initializing the elements of an array to default values of zero. (Part 1 of 2.)

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Fig. 7.2 | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

```java
// Fig. 7.3: InitArray.java
// Initializing the elements of an array with an array initializer.

public class InitArray
{
   public static void main( String args[] )
   {
      // initializer list specifies the value for each element
      int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. 7.3 | Initializing the elements of an array with an array initializer.

```java
// Fig. 7.9: Card.java
// Card class represents a playing card.

public class Card
{
```

Fig. 7.9 | `Card` class represents a playing card. (Part 1 of 2.)

```
    private String face; // face of card ("Ace", "Deuce", ...)
    private String suit; // suit of card ("Hearts", "Diamonds", ...)

    // two-argument constructor initializes card's face and suit
    public Card( String cardFace, String cardSuit )
    {
       face = cardFace; // initialize face of card
       suit = cardSuit; // initialize suit of card
    } // end two-argument Card constructor

    // return String representation of Card
    public String toString()
    {
       return face + " of " + suit;
    } // end method toString
 } // end class Card
```

Fig. 7.9 | `Card` class represents a playing card. (Part 2 of 2.)

```
// Fig. 7.10: DeckOfCards.java
// DeckOfCards class represents a deck of playing cards.
import java.util.Random;

public class DeckOfCards
{
   private Card deck[]; // array of Card objects
   private int currentCard; // index of next Card to be dealt
   private final int NUMBER_OF_CARDS = 52; // constant number of Cards
   private Random randomNumbers; // random number generator

   // constructor fills deck of Cards
   public DeckOfCards()
   {
      String faces[] = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
         "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
      String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

      deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
      currentCard = 0; // set currentCard so first Card dealt is deck[ 0 ]
      randomNumbers = new Random(); // create random number generator

      // populate deck with Card objects
      for ( int count = 0; count < deck.length; count++ )
         deck[ count ] =
            new Card( faces[ count % 13 ], suits[ count / 13 ] );
   } // end DeckOfCards constructor

   // shuffle deck of Cards with one-pass algorithm
   public void shuffle()
   {
```

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards that can be shuffled and dealt one at a time. (Part 1 of 2.)

```java
      // after shuffling, dealing should start at deck[ 0 ] again
      currentCard = 0; // reinitialize currentCard

      // for each Card, pick another random Card and swap them
      for ( int first = 0; first < deck.length; first++ )
      {
         // select a random number between 0 and 51
         int second =  randomNumbers.nextInt( NUMBER_OF_CARDS );

         // swap current Card with randomly selected Card
         Card temp = deck[ first ];
         deck[ first ] = deck[ second ];
         deck[ second ] = temp;
      } // end for
   } // end method shuffle

   // deal one Card
   public Card dealCard()
   {
      // determine whether Cards remain to be dealt
      if ( currentCard < deck.length )
         return deck[ currentCard++ ]; // return current Card in array
      else
         return null; // return null to indicate that all Cards were dealt
   } // end method dealCard
} // end class DeckOfCards
```

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards that can be shuffled and dealt one at a time. (Part 2 of 2.)

```java
// Fig. 7.11: DeckOfCardsTest.java
// Card shuffling and dealing application.

public class DeckOfCardsTest
{
   // execute application
   public static void main( String args[] )
   {
      DeckOfCards myDeckOfCards = new DeckOfCards();
      myDeckOfCards.shuffle(); // place Cards in random order

      // print all 52 Cards in the order in which they are dealt
      for ( int i = 0; i < 13; i++ )
      {
         // deal and print 4 Cards
         System.out.printf( "%-20s%-20s%-20s%-20s\n",
            myDeckOfCards.dealCard(), myDeckOfCards.dealCard(),
            myDeckOfCards.dealCard(), myDeckOfCards.dealCard() );
      } // end for
   } // end main
} // end class DeckOfCardsTest
```

Fig. 7.11 | Card shuffling and dealing. (Part 1 of 2.)

```
Six of Spades        Eight of Spades      Six of Clubs         Nine of Hearts
Queen of Hearts      Seven of Clubs       Nine of Spades       King of Hearts
Three of Diamonds    Deuce of Clubs       Ace of Hearts        Ten of Spades
Four of Spades       Ace of Clubs         Seven of Diamonds    Four of Hearts
Three of Clubs       Deuce of Hearts      Five of Spades       Jack of Diamonds
King of Clubs        Ten of Hearts        Three of Hearts      Six of Diamonds
Queen of Clubs       Eight of Diamonds    Deuce of Diamonds    Ten of Diamonds
Three of Spades      King of Diamonds     Nine of Clubs        Six of Hearts
Ace of Spades        Four of Diamonds     Seven of Hearts      Eight of Clubs
Deuce of Spades      Eight of Hearts      Five of Hearts       Queen of Spades
Jack of Hearts       Seven of Spades      Four of Clubs        Nine of Diamonds
Ace of Diamonds      Queen of Diamonds    Five of Clubs        King of Spades
Five of Diamonds     Ten of Clubs         Jack of Spades       Jack of Clubs
```

Fig. 7.11 | Card shuffling and dealing. (Part 2 of 2.)

```java
// Fig. 7.12: EnhancedForTest.java
// Using enhanced for statement to total integers in an array.

public class EnhancedForTest
{
   public static void main( String args[] )
   {
      int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int number : array )
         total += number;

      System.out.printf( "Total of array elements: %d\n", total );
   } // end main
} // end class EnhancedForTest
```

```
Total of array elements: 849
```

Fig. 7.12 | Using the enhanced `for` statement to total integers in an array.

```java
// Fig. 7.13: PassArray.java
// Passing arrays and individual array elements to methods.

public class PassArray
{
   // main creates array and calls modifyArray and modifyElement
   public static void main( String args[] )
   {
      int array[] = { 1, 2, 3, 4, 5 };

```

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 1 of 2.)

```java
      System.out.println(
         "Effects of passing reference to entire array:\n" +
         "The values of the original array are:" );

      // output original array elements
      for ( int value : array )
         System.out.printf( "   %d", value );

      modifyArray( array ); // pass array reference
      System.out.println( "\n\nThe values of the modified array are:" );

      // output modified array elements
      for ( int value : array )
         System.out.printf( "   %d", value );

      System.out.printf(
         "\n\nEffects of passing array element value:\n" +
         "array[3] before modifyElement: %d\n", array[ 3 ] );

      modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
      System.out.printf(
         "array[3] after modifyElement: %d\n", array[ 3 ] );
   } // end main

   // multiply each element of an array by 2
   public static void modifyArray( int array2[] )
   {
      for ( int counter = 0; counter < array2.length; counter++ )
         array2[ counter ] *= 2;
   } // end method modifyArray

   // multiply argument by 2
   public static void modifyElement( int element )
   {
      element *= 2;
      System.out.printf(
         "Value of element in modifyElement: %d\n", element );
   } // end method modifyElement
} // end class PassArray
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 2 of 2.)

```java
// Fig. 7.14: GradeBook.java
// Grade book using an array to store test grades.

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents
   private int grades[]; // array of student grades

   // two-argument constructor initializes courseName and grades array
   public GradeBook( String name, int gradesArray[] )
   {
      courseName = name; // initialize courseName
      grades = gradesArray; // store grades
   } // end two-argument GradeBook constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // perform various operations on the data
   public void processGrades()
   {
      // output grades array
      outputGrades();

      // call method getAverage to calculate the average grade
      System.out.printf( "\nClass average is %.2f\n", getAverage() );

      // call methods getMinimum and getMaximum
      System.out.printf( "Lowest grade is %d\nHighest grade is %d\n\n",
         getMinimum(), getMaximum() );

      // call outputBarChart to print grade distribution chart
      outputBarChart();
   } // end method processGrades

```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 1 of 3.)

```java
    // find minimum grade
    public int getMinimum()
    {
       int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest

       // loop through grades array
       for ( int grade : grades )
       {
          // if grade lower than lowGrade, assign it to lowGrade
          if ( grade < lowGrade )
             lowGrade = grade; // new lowest grade
       } // end for

       return lowGrade; // return lowest grade
    } // end method getMinimum

    // find maximum grade
    public int getMaximum()
    {
       int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest

       // loop through grades array
       for ( int grade : grades )
       {
          // if grade greater than highGrade, assign it to highGrade
          if ( grade > highGrade )
             highGrade = grade; // new highest grade
       } // end for

       return highGrade; // return highest grade
    } // end method getMaximum

    // determine average grade for test
    public double getAverage()
    {
       int total = 0; // initialize total

       // sum grades for one student
       for ( int grade : grades )
          total += grade;

       // return average of grades
       return (double) total / grades.length;
    } // end method getAverage

    // output bar chart displaying grade distribution
    public void outputBarChart()
    {
       System.out.println( "Grade distribution:" );

       // stores frequency of grades in each range of 10 grades
       int frequency[] = new int[ 11 ];
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 2 of 3.)

```

      // for each grade, increment the appropriate frequency
      for ( int grade : grades )
         ++frequency[ grade / 10 ];

      // for each grade frequency, print bar in chart
      for ( int count = 0; count < frequency.length; count++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
               count * 10, count * 10 + 9  );

         // print bar of asterisks
         for ( int stars = 0; stars < frequency[ count ]; stars++ )
            System.out.print( "*" );

         System.out.println(); // start a new line of output
      } // end outer for
   } // end method outputBarChart

   // output the contents of the grades array
   public void outputGrades()
   {
      System.out.println( "The grades are:\n" );

      // output each student's grade
      for ( int student = 0; student < grades.length; student++ )
         System.out.printf( "Student %2d: %3d\n",
            student + 1, grades[ student ] );
   } // end method outputGrades
} // end class GradeBook
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 3 of 3.)

```
// Fig. 7.15: GradeBookTest.java
// Creates GradeBook object using an array of grades.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String args[] )
   {
      // array of student grades
      int gradesArray[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };

      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming", gradesArray );
```

Fig. 7.15 | GradeBookTest creates a GradeBook object using an array of grades, then invokes method processGrades to analyze them. (Part 1 of 2.)

```
      myGradeBook.displayMessage();
      myGradeBook.processGrades();
   } // end main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.15 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 2.)

```
// Fig. 7.17: InitArray.java
// Initializing two-dimensional arrays.

public class InitArray
{
   // create and output two-dimensional arrays
   public static void main( String args[] )
   {
      int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
      int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };

      System.out.println( "Values in array1 by row are" );
```

Fig. 7.17 | Initializing two-dimensional arrays. (Part 1 of 2.)

```java
      outputArray( array1 ); // displays array1 by row

      System.out.println( "\nValues in array2 by row are" );
      outputArray( array2 ); // displays array2 by row
   } // end main

   // output rows and columns of a two-dimensional array
   public static void outputArray( int array[][] )
   {
      // loop through array's rows
      for ( int row = 0; row < array.length; row++ )
      {
         // loop through columns of current row
         for ( int column = 0; column < array[ row ].length; column++ )
            System.out.printf( "%d  ", array[ row ][ column ] );

         System.out.println(); // start new line of output
      } // end outer for
   } // end method outputArray
} // end class InitArray
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. 7.17 | Initializing two-dimensional arrays. (Part 2 of 2.)

```java
// Fig. 7.18: GradeBook.java
// Grade book using a two-dimensional array to store grades.

public class GradeBook
{
   private String courseName; // name of course this grade book represents
   private int grades[][]; // two-dimensional array of student grades

   // two-argument constructor initializes courseName and grades array
   public GradeBook( String name, int gradesArray[][] )
   {
      courseName = name; // initialize courseName
      grades = gradesArray; // store grades
   } // end two-argument GradeBook constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName
```

Fig. 7.18 | `GradeBook` class using a two-dimensional array to store grades. (Part 1 of 4.)

```

    // method to retrieve the course name
    public String getCourseName()
    {
       return courseName;
    } // end method getCourseName

    // display a welcome message to the GradeBook user
    public void displayMessage()
    {
       // getCourseName gets the name of the course
       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
          getCourseName() );
    } // end method displayMessage

    // perform various operations on the data
    public void processGrades()
    {
       // output grades array
       outputGrades();

       // call methods getMinimum and getMaximum
       System.out.printf( "\n%s %d\n%s %d\n\n",
          "Lowest grade in the grade book is", getMinimum(),
          "Highest grade in the grade book is", getMaximum() );

       // output grade distribution chart of all grades on all tests
       outputBarChart();
    } // end method processGrades

    // find minimum grade
    public int getMinimum()
    {
       // assume first element of grades array is smallest
       int lowGrade = grades[ 0 ][ 0 ];

       // loop through rows of grades array
       for ( int studentGrades[] : grades )
       {
          // loop through columns of current row
          for ( int grade : studentGrades )
          {
             // if grade less than lowGrade, assign it to lowGrade
             if ( grade < lowGrade )
                lowGrade = grade;
          } // end inner for
       } // end outer for

       return lowGrade; // return lowest grade
    } // end method getMinimum

```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 2 of 4.)

```
   // find maximum grade
   public int getMaximum()
   {
      // assume first element of grades array is largest
      int highGrade = grades[ 0 ][ 0 ];

      // loop through rows of grades array
      for ( int studentGrades[] : grades )
      {
         // loop through columns of current row
         for ( int grade : studentGrades )
         {
            // if grade greater than highGrade, assign it to highGrade
            if ( grade > highGrade )
               highGrade = grade;
         } // end inner for
      } // end outer for

      return highGrade; // return highest grade
   } // end method getMaximum

   // determine average grade for particular set of grades
   public double getAverage( int setOfGrades[] )
   {
      int total = 0; // initialize total

      // sum grades for one student
      for ( int grade : setOfGrades )
         total += grade;

      // return average of grades
      return (double) total / setOfGrades.length;
   } // end method getAverage

   // output bar chart displaying overall grade distribution
   public void outputBarChart()
   {
      System.out.println( "Overall grade distribution:" );

      // stores frequency of grades in each range of 10 grades
      int frequency[] = new int[ 11 ];

      // for each grade in GradeBook, increment the appropriate frequency
      for ( int studentGrades[] : grades )
      {
         for ( int grade : studentGrades )
            ++frequency[ grade / 10 ];
      } // end outer for

      // for each grade frequency, print bar in chart
      for ( int count = 0; count < frequency.length; count++ )
      {
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 3 of 4.)

```
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
               count * 10, count * 10 + 9  );

         // print bar of asterisks
         for ( int stars = 0; stars < frequency[ count ]; stars++ )
            System.out.print( "*" );

         System.out.println(); // start a new line of output
      } // end outer for
   } // end method outputBarChart

   // output the contents of the grades array
   public void outputGrades()
   {
      System.out.println( "The grades are:\n" );
      System.out.print( "            " ); // align column heads

      // create a column heading for each of the tests
      for ( int test = 0; test < grades[ 0 ].length; test++ )
         System.out.printf( "Test %d  ", test + 1 );

      System.out.println( "Average" ); // student average column heading

      // create rows/columns of text representing array grades
      for ( int student = 0; student < grades.length; student++ )
      {
         System.out.printf( "Student %2d", student + 1 );

         for ( int test : grades[ student ] ) // output student's grades
            System.out.printf( "%8d", test );

         // call method getAverage to calculate student's average grade;
         // pass row of grades as the argument to getAverage
         double average = getAverage( grades[ student ] );
         System.out.printf( "%9.2f\n", average );
      } // end outer for
   } // end method outputGrades
} // end class GradeBook
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 4 of 4.)

```
// Fig. 7.19: GradeBookTest.java
// Creates GradeBook object using a two-dimensional array of grades.

public class GradeBookTest
{
```

Fig. 7.19 | Creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 1 of 2.)

```java
    // main method begins program execution
    public static void main( String args[] )
    {
       // two-dimensional array of student grades
       int gradesArray[][] = { { 87, 96, 70 },
                               { 68, 87, 90 },
                               { 94, 100, 90 },
                               { 100, 81, 82 },
                               { 83, 65, 85 },
                               { 78, 87, 65 },
                               { 85, 75, 83 },
                               { 91, 94, 100 },
                               { 76, 72, 84 },
                               { 87, 93, 73 } };

       GradeBook myGradeBook = new GradeBook(
          "CS101 Introduction to Java Programming", gradesArray );
       myGradeBook.displayMessage();
       myGradeBook.processGrades();
    } // end main
 } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.19 | Creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 2.)

```java
// Fig. 7.20: VarargsTest.java
// Using variable-length argument lists.

public class VarargsTest
{
   // calculate average
   public static double average( double... numbers )
   {
      double total = 0.0; // initialize total

      // calculate total using the enhanced for statement
      for ( double d : numbers )
         total += d;

      return total / numbers.length;
   } // end method average

   public static void main( String args[] )
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
         d1, d2, d3, d4 );

      System.out.printf( "Average of d1 and d2 is %.1f\n",
         average( d1, d2 ) );
      System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
         average( d1, d2, d3 ) );
      System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
         average( d1, d2, d3, d4 ) );
   } // end main
} // end class VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 7.20 | Using variable-length argument lists.

```java
// Fig. 7.21: InitArray.java
// Using command-line arguments to initialize an array.

public class InitArray
{
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 1 of 2.)

```java
    public static void main( String args[] )
    {
       // check number of command-line arguments
       if ( args.length != 3 )
          System.out.println(
             "Error: Please re-enter the entire command, including\n" +
             "an array size, initial value and increment." );
       else
       {
          // get array size from first command-line argument
          int arrayLength = Integer.parseInt( args[ 0 ] );
          int array[] = new int[ arrayLength ]; // create array

          // get initial value and increment from command-line arguments
          int initialValue = Integer.parseInt( args[ 1 ] );
          int increment = Integer.parseInt( args[ 2 ] );

          // calculate value for each array element
          for ( int counter = 0; counter < array.length; counter++ )
             array[ counter ] = initialValue + increment * counter;

          System.out.printf( "%s%8s\n", "Index", "Value" );

          // display array index and value
          for ( int counter = 0; counter < array.length; counter++ )
             System.out.printf( "%5d%8d\n", counter, array[ counter ] );
       } // end else
    } // end main
 } // end class InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 2 of 2.)

7 lesson

Classes and Objects: A Deeper Look

Based on Chapter 8 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Understand encapsulation and data hiding.
- Use keyword `this` to refer to an object's members.
- Create `static` variables and methods.
- Import `static` members of a class.
- Use the `enum` type to create sets of constants with unique identifiers, and to declare enum constants with parameters.
- Organize classes in packages for reusability.

Figures

```java
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime

```

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 1 of 2.)

```java
   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format( "%02d:%02d:%02d", hour, minute, second );
   } // end method toUniversalString

   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
         minute, second, ( hour < 12 ? "AM" : "PM" ) );
   } // end method toString
} // end class Time1
```

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 2 of 2.)

```java
// Fig. 8.2: Time1Test.java
// Time1 object used in an application.

public class Time1Test
{
   public static void main( String args[] )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // change time and output updated time
      time.setTime( 13, 27, 6 );
      System.out.print( "Universal time after setTime is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time after setTime is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // set time with invalid values; output updated time
      time.setTime( 99, 99, 99 );
      System.out.println( "After attempting invalid settings:" );
      System.out.print( "Universal time: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time: " );
      System.out.println( time.toString() );
   } // end main
} // end class Time1Test
```

Fig. 8.2 | `Time1` object used in an application. (Part 1 of 2.)

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 8.2 | `Time1` object used in an application. (Part 2 of 2.)

```java
// Fig. 8.3: MemberAccessTest.java
// Private members of class Time1 are not accessible.
public class MemberAccessTest
{
   public static void main( String args[] )
   {
      Time1 time = new Time1(); // create and initialize Time1 object

      time.hour = 7;    // error: hour has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
      time.second = 30; // error: second has private access in Time1
   } // end main
} // end class MemberAccessTest
```

```
MemberAccessTest.java:9: hour has private access in Time1
      time.hour = 7;    // error: hour has private access in Time1
          ^
MemberAccessTest.java:10: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
          ^
MemberAccessTest.java:11: second has private access in Time1
      time.second = 30; // error: second has private access in Time1
          ^
3 errors
```

Fig. 8.3 | Private members of class `Time1` are not accessible.

```java
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

public class ThisTest
{
   public static void main( String args[] )
   {
      SimpleTime time = new SimpleTime( 15, 30, 19 );
      System.out.println( time.buildString() );
   } // end main
} // end class ThisTest
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 2.)

```java

// class SimpleTime demonstrates the "this" reference
class SimpleTime
{
   private int hour;   // 0-23
   private int minute; // 0-59
   private int second; // 0-59

   // if the constructor uses parameter names identical to
   // instance variable names the "this" reference is
   // required to distinguish between names
   public SimpleTime( int hour, int minute, int second )
   {
      this.hour = hour;      // set "this" object's hour
      this.minute = minute;  // set "this" object's minute
      this.second = second;  // set "this" object's second
   } // end SimpleTime constructor

   // use explicit and implicit "this" to call toUniversalString
   public String buildString()
   {
      return String.format( "%24s: %s\n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()", toUniversalString() );
   } // end method buildString

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      // "this" is not required here to access instance variables,
      // because method does not have local variables with same
      // names as instance variables
      return String.format( "%02d:%02d:%02d",
         this.hour, this.minute, this.second );
   } // end method toUniversalString
} // end class SimpleTime
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 2 of 2.)

```java
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.

public class Time2
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 1 of 3.)

```java

   // Time2 no-argument constructor: initializes each instance variable
   // to zero; ensures that Time2 objects start in a consistent state
   public Time2()
   {
      this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 no-argument constructor

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2( int h )
   {
      this( h, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 one-argument constructor

   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2( int h, int m )
   {
      this( h, m, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 two-argument constructor

   // Time2 constructor: hour, minute and second supplied
   public Time2( int h, int m, int s )
   {
      setTime( h, m, s ); // invoke setTime to validate time
   } // end Time2 three-argument constructor

   // Time2 constructor: another Time2 object supplied
   public Time2( Time2 time )
   {
      // invoke Time2 three-argument constructor
      this( time.getHour(), time.getMinute(), time.getSecond() );
   } // end Time2 constructor with a Time2 object argument

   // Set Methods
   // set a new time value using universal time; ensure that
   // the data remains consistent by setting invalid values to zero
   public void setTime( int h, int m, int s )
   {
      setHour( h );   // set the hour
      setMinute( m ); // set the minute
      setSecond( s ); // set the second
   } // end method setTime

   // validate and set hour
   public void setHour( int h )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
   } // end method setHour

   // validate and set minute
   public void setMinute( int m )
   {
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 2 of 3.)

```
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
    } // end method setMinute

    // validate and set second
    public void setSecond( int s )
    {
        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
    } // end method setSecond

    // Get Methods
    // get hour value
    public int getHour()
    {
        return hour;
    } // end method getHour

    // get minute value
    public int getMinute()
    {
        return minute;
    } // end method getMinute

    // get second value
    public int getSecond()
    {
        return second;
    } // end method getSecond

    // convert to String in universal-time format (HH:MM:SS)
    public String toUniversalString()
    {
        return String.format(
            "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
    } // end method toUniversalString

    // convert to String in standard-time format (H:MM:SS AM or PM)
    public String toString()
    {
        return String.format( "%d:%02d:%02d %s",
            ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
            getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
    } // end method toString
} // end class Time2
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 3 of 3.)

```
// Fig. 8.6: Time2Test.java
// Overloaded constructors used to initialize Time2 objects.

public class Time2Test
{
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 1 of 3.)

```java
   public static void main( String args[] )
   {
      Time2 t1 = new Time2();             // 00:00:00
      Time2 t2 = new Time2( 2 );          // 02:00:00
      Time2 t3 = new Time2( 21, 34 );     // 21:34:00
      Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
      Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
      Time2 t6 = new Time2( t4 );         // 12:25:42

      System.out.println( "Constructed with:" );
      System.out.println( "t1: all arguments defaulted" );
      System.out.printf( "   %s\n", t1.toUniversalString() );
      System.out.printf( "   %s\n", t1.toString() );

      System.out.println(
         "t2: hour specified; minute and second defaulted" );
      System.out.printf( "   %s\n", t2.toUniversalString() );
      System.out.printf( "   %s\n", t2.toString() );

      System.out.println(
         "t3: hour and minute specified; second defaulted" );
      System.out.printf( "   %s\n", t3.toUniversalString() );
      System.out.printf( "   %s\n", t3.toString() );

      System.out.println( "t4: hour, minute and second specified" );
      System.out.printf( "   %s\n", t4.toUniversalString() );
      System.out.printf( "   %s\n", t4.toString() );

      System.out.println( "t5: all invalid values specified" );
      System.out.printf( "   %s\n", t5.toUniversalString() );
      System.out.printf( "   %s\n", t5.toString() );

      System.out.println( "t6: Time2 object t4 specified" );
      System.out.printf( "   %s\n", t6.toUniversalString() );
      System.out.printf( "   %s\n", t6.toString() );
   } // end main
} // end class Time2Test
```

```
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 3.)

```
t5: all invalid values specified
   00:00:00
   12:00:00 AM
t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 3.)

```java
// Fig. 8.7: Date.java
// Date class declaration.

public class Date
{
   private int month; // 1-12
   private int day;   // 1-31 based on month
   private int year;  // any year

   // constructor: call checkMonth to confirm proper value for month;
   // call checkDay to confirm proper value for day
   public Date( int theMonth, int theDay, int theYear )
   {
      month = checkMonth( theMonth ); // validate month
      year = theYear; // could validate year
      day = checkDay( theDay ); // validate day

      System.out.printf(
         "Date object constructor for date %s\n", this );
   } // end Date constructor

   // utility method to confirm proper month value
   private int checkMonth( int testMonth )
   {
      if ( testMonth > 0 && testMonth <= 12 ) // validate month
         return testMonth;
      else // month is invalid
      {
         System.out.printf(
            "Invalid month (%d) set to 1.", testMonth );
         return 1; // maintain object in consistent state
      } // end else
   } // end method checkMonth

   // utility method to confirm proper day value based on month and year
   private int checkDay( int testDay )
   {
      int daysPerMonth[] =
         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

```

Fig. 8.7 | `Date` class declaration. (Part 1 of 2.)

```
      // check if day in range for month
      if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
         return testDay;

      // check for leap year
      if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
           ( year % 4 == 0 && year % 100 != 0 ) ) )
         return testDay;

      System.out.printf( "Invalid day (%d) set to 1.", testDay );
      return 1;  // maintain object in consistent state
   } // end method checkDay

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format( "%d/%d/%d", month, day, year );
   } // end method toString
} // end class Date
```

Fig. 8.7 | `Date` class declaration. (Part 2 of 2.)

```
// Fig. 8.8: Employee.java
// Employee class with references to other objects.

public class Employee
{
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Date hireDate;

   // constructor to initialize name, birth date and hire date
   public Employee( String first, String last, Date dateOfBirth,
      Date dateOfHire )
   {
      firstName = first;
      lastName = last;
      birthDate = dateOfBirth;
      hireDate = dateOfHire;
   } // end Employee constructor

   // convert Employee to String format
   public String toString()
   {
      return String.format( "%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate );
   } // end method toString
} // end class Employee
```

Fig. 8.8 | `Employee` class with references to other objects.

```
// Fig. 8.9: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest
{
   public static void main( String args[] )
   {
      Date birth = new Date( 7, 24, 1949 );
      Date hire = new Date( 3, 12, 1988 );
      Employee employee = new Employee( "Bob", "Blue", birth, hire );

      System.out.println( employee );
   } // end main
} // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.

```
// Fig. 8.10: Book.java
// Declaring an enum type with constructor and explicit instance fields
// and accessors for these fields

public enum Book
{
   // declare constants of enum type
   JHTP6( "Java How to Program 6e", "2005" ),
   CHTP4( "C How to Program 4e", "2004" ),
   IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
   CPPHTP4( "C++ How to Program 4e", "2003" ),
   VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
   CSHARPHTP( "C# How to Program", "2002" );

   // instance fields
   private final String title; // book title
   private final String copyrightYear; // copyright year

   // enum constructor
   Book( String bookTitle, String year )
   {
      title = bookTitle;
      copyrightYear = year;
   } // end enum Book constructor

   // accessor for field title
   public String getTitle()
   {
      return title;
   } // end method getTitle
```

Fig. 8.10 | Declaring `enum` type with instance fields, constructor and methods. (Part 1 of 2.)

```java

   // accessor for field copyrightYear
   public String getCopyrightYear()
   {
      return copyrightYear;
   } // end method getCopyrightYear
} // end enum Book
```

Fig. 8.10 | Declaring `enum` type with instance fields, constructor and methods. (Part 2 of 2.)

```java
// Fig. 8.11: EnumTest.java
// Testing enum type Book.
import java.util.EnumSet;

public class EnumTest
{
   public static void main( String args[] )
   {
      System.out.println( "All books:\n" );

      // print all books in enum Book
      for ( Book book : Book.values() )
         System.out.printf( "%-10s%-45s%s\n", book,
             book.getTitle(), book.getCopyrightYear() );

      System.out.println( "\nDisplay a range of enum constants:\n" );

      // print first four books
      for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
         System.out.printf( "%-10s%-45s%s\n", book,
             book.getTitle(), book.getCopyrightYear() );
   } // end main
} // end class EnumTest
```

```
All books:

JHTP6     Java How to Program 6e                       2005
CHTP4     C How to Program 4e                          2004
IW3HTP3   Internet & World Wide Web How to Program 3e  2004
CPPHTP4   C++ How to Program 4e                        2003
VBHTP2    Visual Basic .NET How to Program 2e          2002
CSHARPHTP C# How to Program                            2002

Display a range of enum constants:

JHTP6     Java How to Program 6e                       2005
CHTP4     C How to Program 4e                          2004
IW3HTP3   Internet & World Wide Web How to Program 3e  2004
CPPHTP4   C++ How to Program 4e                        2003
```

Fig. 8.11 | Testing an `enum` type.

```java
// Fig. 8.12: Employee.java
// Static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   private String firstName;
   private String lastName;
   private static int count = 0; // number of objects in memory

   // initialize employee, add 1 to static count and
   // output String indicating that constructor was called
   public Employee( String first, String last )
   {
      firstName = first;
      lastName = last;

      count++;  // increment static count of employees
      System.out.printf( "Employee constructor: %s %s; count = %d\n",
         firstName, lastName, count );
   } // end Employee constructor

   // subtract 1 from static count when garbage
   // collector calls finalize to clean up object;
   // confirm that finalize was called
   protected void finalize()
   {
      count--; // decrement static count of employees
      System.out.printf( "Employee finalizer: %s %s; count = %d\n",
         firstName, lastName, count );
   } // end method finalize

   // get first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // get last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // static method to get static count value
   public static int getCount()
   {
      return count;
   } // end method getCount
} // end class Employee
```

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory.

```java
// Fig. 8.13: EmployeeTest.java
// Static member demonstration.

public class EmployeeTest
{
   public static void main( String args[] )
   {
      // show that count is 0 before creating Employees
      System.out.printf( "Employees before instantiation: %d\n",
         Employee.getCount() );

      // create two Employees; count should be 2
      Employee e1 = new Employee( "Susan", "Baker" );
      Employee e2 = new Employee( "Bob", "Blue" );

      // show that count is 2 after creating two Employees
      System.out.println( "\nEmployees after instantiation: " );
      System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
      System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
      System.out.printf( "via Employee.getCount(): %d\n",
         Employee.getCount() );

      // get names of Employees
      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
         e1.getFirstName(), e1.getLastName(),
         e2.getFirstName(), e2.getLastName() );

      // in this example, there is only one reference to each Employee,
      // so the following two statements cause the JVM to mark each
      // Employee object for garbage collection
      e1 = null;
      e2 = null;

      System.gc(); // ask for garbage collection to occur now

      // show Employee count after calling garbage collector; count
      // displayed may be 0, 1 or 2 based on whether garbage collector
      // executes immediately and number of Employee objects collected
      System.out.printf( "\nEmployees after System.gc(): %d\n",
         Employee.getCount() );
   } // end main
} // end class EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2
```

Fig. 8.13 | `static` member demonstration. (Part 1 of 2.)

```
Employee 1: Susan Baker
Employee 2: Bob Blue

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
```

Fig. 8.13 | `static` member demonstration. (Part 2 of 2.)

```java
// Fig. 8.14: StaticImportTest.java
// Using static import to import static methods of class Math.
import static java.lang.Math.*;

public class StaticImportTest
{
   public static void main( String args[] )
   {
      System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
      System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
      System.out.printf( "log( E ) = %.1f\n", log( E ) );
      System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
   } // end main
} // end class StaticImportTest
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Fig. 8.14 | Static import `Math` methods.

```java
// Fig. 8.15: Increment.java
// final instance variable in a class.

public class Increment
{
   private int total = 0; // total of all increments
   private final int INCREMENT; // constant variable (uninitialized)

   // constructor initializes final instance variable INCREMENT
   public Increment( int incrementValue )
   {
      INCREMENT = incrementValue; // initialize constant variable (once)
   } // end Increment constructor

   // add INCREMENT to total
   public void addIncrementToTotal()
   {
```

Fig. 8.15 | `final` instance variable in a class. (Part 1 of 2.)

```java
      total += INCREMENT;
   } // end method addIncrementToTotal

   // return String representation of an Increment object's data
   public String toString()
   {
      return String.format( "total = %d", total );
   } // end method toIncrementString
} // end class Increment
```

Fig. 8.15 | `final` instance variable in a class. (Part 2 of 2.)

```java
// Fig. 8.16: IncrementTest.java
// final variable initialized with a constructor argument.

public class IncrementTest
{
   public static void main( String args[] )
   {
      Increment value = new Increment( 5 );

      System.out.printf( "Before incrementing: %s\n\n", value );

      for ( int i = 1; i <= 3; i++ )
      {
         value.addIncrementToTotal();
         System.out.printf( "After increment %d: %s\n", i, value );
      } // end for
   } // end main
} // end class IncrementTest
```

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Fig. 8.16 | `final` variable initialized with a constructor argument.

```java
// Fig. 8.18: Time1.java
// Time1 class declaration maintains the time in 24-hour format.
package com.deitel.jhtp7.ch08;

public class Time1
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; perform
   // validity checks on the data; set invalid values to zero
```

Fig. 8.18 | Packaging class `Time1` for reuse. (Part 1 of 2.)

```java
    public void setTime( int h, int m, int s )
    {
       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
    } // end method setTime

    // convert to String in universal-time format (HH:MM:SS)
    public String toUniversalString()
    {
       return String.format( "%02d:%02d:%02d", hour, minute, second );
    } // end method toUniversalString

    // convert to String in standard-time format (H:MM:SS AM or PM)
    public String toString()
    {
       return String.format( "%d:%02d:%02d %s",
          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
          minute, second, ( hour < 12 ? "AM" : "PM" ) );
    } // end method toString
} // end class Time1
```

Fig. 8.18 | Packaging class `Time1` for reuse. (Part 2 of 2.)

```java
// Fig. 8.19: Time1PackageTest.java
// Time1 object used in an application.
import com.deitel.jhtp7.ch08.Time1; // import class Time1

public class Time1PackageTest
{
   public static void main( String args[] )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // calls Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // change time and output updated time
      time.setTime( 13, 27, 6 );
      System.out.print( "Universal time after setTime is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time after setTime is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

```

Fig. 8.19 | `Time1` object used in an application. (Part 1 of 2.)

```
      // set time with invalid values; output updated time
      time.setTime( 99, 99, 99 );
      System.out.println( "After attempting invalid settings:" );
      System.out.print( "Universal time: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time: " );
      System.out.println( time.toString() );
   } // end main
} // end class Time1PackageTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 8.19 | `Time1` object used in an application. (Part 2 of 2.)

lesson 1

Object-Oriented Programming: Inheritance

Based on Chapter 9 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Understand how inheritance promotes software reusability.
- Understand the notions of superclasses and subclasses.
- Use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- Use access modifier `protected` to give subclass methods access to superclass members.
- Access superclass members from a subclass with `super`.
- Understand how constructors are used in inheritance hierarchies.
- Learn the methods of class `Object`, the direct or indirect superclass of all classes in Java.

Figures

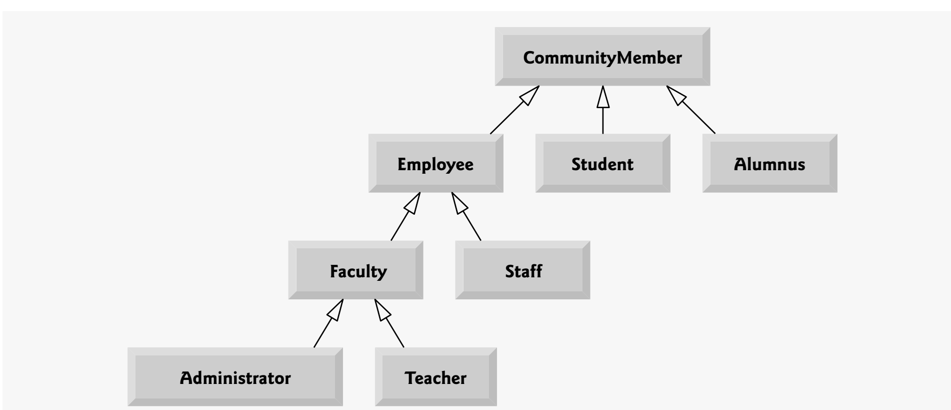

Fig. 9.2 | Inheritance hierarchy for university `CommunityMember`s.

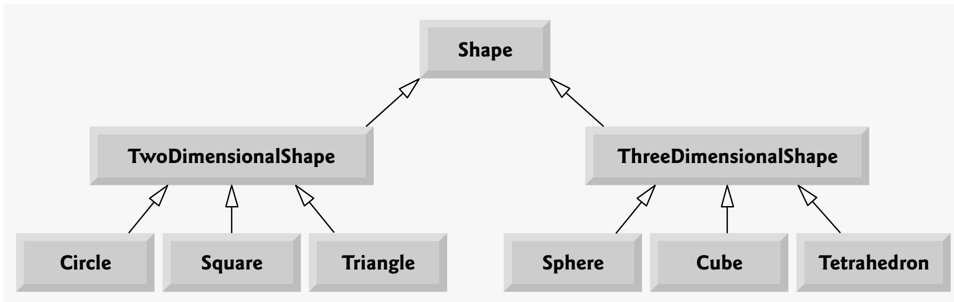

Fig. 9.3 | Inheritance hierarchy for `Shapes`.

```java
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents a commission employee.

public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 3.)

```
   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 3.)

```java
      return commissionRate * grossSales;
   } // end method earnings

   // return String representation of CommissionEmployee object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales,
         "commission rate", commissionRate );
   } // end method toString
} // end class CommissionEmployee
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 3.)

```java
// Fig. 9.5: CommissionEmployeeTest.java
// Testing class CommissionEmployee.

public class CommissionEmployeeTest
{
   public static void main( String args[] )
   {
      // instantiate CommissionEmployee object
      CommissionEmployee employee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // get commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );

      employee.setGrossSales( 500 ); // set gross sales
      employee.setCommissionRate( .1 ); // set commission rate

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString", employee );
   } // end main
} // end class CommissionEmployeeTest
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 1 of 2.)

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 2 of 2.)

```java
// Fig. 9.6: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class represents an employee that receives
// a base salary in addition to commission.

public class BasePlusCommissionEmployee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 3.)

```
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 3.)

```
      return commissionRate;
   } // end method getCommissionRate

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee
   public String toString()
   {
      return String.format(
         "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
         "base-salaried commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales, "commission rate", commissionRate,
         "base salary", baseSalary );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 3.)

```
// Fig. 9.7: BasePlusCommissionEmployeeTest.java
// Testing class BasePlusCommissionEmployee.

public class BasePlusCommissionEmployeeTest
{
   public static void main( String args[] )
   {
      // instantiate BasePlusCommissionEmployee object
      BasePlusCommissionEmployee employee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 2.)

```
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
      System.out.printf( "%s %.2f\n", "Base salary is",
         employee.getBaseSalary() );

      employee.setBaseSalary( 1000 ); // set base salary

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString",
          employee.toString() );
   } // end main
} // end class BasePlusCommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 2.)

```
// Fig. 9.8: BasePlusCommissionEmployee2.java
// BasePlusCommissionEmployee2 inherits from class CommissionEmployee.

public class BasePlusCommissionEmployee2 extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee2( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 1 of 3.)

```java
      // explicit call to superclass CommissionEmployee constructor
      super( first, last, ssn, sales, rate );

      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee2 constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      // not allowed: commissionRate and grossSales private in superclass
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee2
   public String toString()
   {
      // not allowed: attempts to access private superclass members
      return String.format(
         "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
         "base-salaried commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales, "commission rate", commissionRate,
         "base salary", baseSalary );
   } // end method toString
} // end class BasePlusCommissionEmployee2
```

```
BasePlusCommissionEmployee2.java:34: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee2.java:34: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee2.java:43: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 3.)

```
BasePlusCommissionEmployee2.java:43: lastName has private access in
CommissionEmploye
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee2.java:44: socialSecurityNumber has private access
in CommissionEmploye
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee2.java:45: grossSales has private access in
CommissionEmploye
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee2.java:45: commissionRate has private access in
CommissionEmploye
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 3.)

```java
// Fig. 9.9: CommissionEmployee2.java
// CommissionEmployee2 class represents a commission employee.

public class CommissionEmployee2
{
   protected String firstName;
   protected String lastName;
   protected String socialSecurityNumber;
   protected double grossSales; // gross weekly sales
   protected double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee2( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee2 constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
```

Fig. 9.9 | `CommissionEmployee2` with `protected` instance variables. (Part 1 of 3.)

```
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

```

Fig. 9.9 | `CommissionEmployee2` with `protected` instance variables. (Part 2 of 3.)

```java
   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings

   // return String representation of CommissionEmployee2 object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales,
         "commission rate", commissionRate );
   } // end method toString
} // end class CommissionEmployee2
```

Fig. 9.9 | `CommissionEmployee2` with `protected` instance variables. (Part 3 of 3.)

```java
// Fig. 9.10: BasePlusCommissionEmployee3.java
// BasePlusCommissionEmployee3 inherits from CommissionEmployee2 and has
// access to CommissionEmployee2's protected members.

public class BasePlusCommissionEmployee3 extends CommissionEmployee2
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee3( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee3 constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
```

Fig. 9.10 | `BasePlusCommissionEmployee3` inherits protected instance variables from `CommissionEmployee2`. (Part 1 of 2.)

```
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee3
   public String toString()
   {
      return String.format(
         "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
         "base-salaried commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales, "commission rate", commissionRate,
         "base salary", baseSalary );
   } // end method toString
} // end class BasePlusCommissionEmployee3
```

Fig. 9.10 | `BasePlusCommissionEmployee3` inherits protected instance variables from `CommissionEmployee2`. (Part 2 of 2.)

```
// Fig. 9.11: BasePlusCommissionEmployeeTest3.java
// Testing class BasePlusCommissionEmployee3.

public class BasePlusCommissionEmployeeTest3
{
   public static void main( String args[] )
   {
      // instantiate BasePlusCommissionEmployee3 object
      BasePlusCommissionEmployee3 employee =
         new BasePlusCommissionEmployee3(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
      System.out.printf( "%s %.2f\n", "Base salary is",
         employee.getBaseSalary() );

      employee.setBaseSalary( 1000 ); // set base salary

```

Fig. 9.11 | `protected` superclass members inherited into subclass `BasePlusCommission-Employee3`. (Part 1 of 2.)

```java
        System.out.printf( "\n%s:\n\n%s\n",
           "Updated employee information obtained by toString",
           employee.toString() );
    } // end main
} // end class BasePlusCommissionEmployeeTest3
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.11 | `protected` superclass members inherited into subclass `BasePlusCommissionEmployee3`. (Part 2 of 2.)

```java
// Fig. 9.12: CommissionEmployee3.java
// CommissionEmployee3 class represents a commission employee.

public class CommissionEmployee3
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee3( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee3 constructor

```

Fig. 9.12 | `CommissionEmployee3` class uses methods to manipulate its `private` instance variables. (Part 1 of 3.)

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
```

Fig. 9.12 | `CommissionEmployee3` class uses methods to manipulate its `private` instance variables. (Part 2 of 3.)

```
        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
    } // end method setCommissionRate

    // return commission rate
    public double getCommissionRate()
    {
        return commissionRate;
    } // end method getCommissionRate

    // calculate earnings
    public double earnings()
    {
        return getCommissionRate() * getGrossSales();
    } // end method earnings

    // return String representation of CommissionEmployee3 object
    public String toString()
    {
        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
            "commission employee", getFirstName(), getLastName(),
            "social security number", getSocialSecurityNumber(),
            "gross sales", getGrossSales(),
            "commission rate", getCommissionRate() );
    } // end method toString
} // end class CommissionEmployee3
```

Fig. 9.12 | `CommissionEmployee3` class uses methods to manipulate its `private` instance variables. (Part 3 of 3.)

```
// Fig. 9.13: BasePlusCommissionEmployee4.java
// BasePlusCommissionEmployee4 class inherits from CommissionEmployee3 and
// accesses CommissionEmployee3's private data via CommissionEmployee3's
// public methods.

public class BasePlusCommissionEmployee4 extends CommissionEmployee3
{
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee4( String first, String last,
        String ssn, double sales, double rate, double salary )
    {
        super( first, last, ssn, sales, rate );
        setBaseSalary( salary ); // validate and store base salary
    } // end six-argument BasePlusCommissionEmployee4 constructor

    // set base salary
    public void setBaseSalary( double salary )
    {
```

Fig. 9.13 | `BasePlusCommissionEmployee4` class extends `CommissionEmployee3`, which provides only private instance variables. (Part 1 of 2.)

```
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee4
   public String toString()
   {
      return String.format( "%s %s\n%s: %.2f", "base-salaried",
         super.toString(), "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee4
```

Fig. 9.13 | `BasePlusCommissionEmployee4` class extends `CommissionEmployee3`, which provides only private instance variables. (Part 2 of 2.)

```
// Fig. 9.14: BasePlusCommissionEmployeeTest4.java
// Testing class BasePlusCommissionEmployee4.

public class BasePlusCommissionEmployeeTest4
{
   public static void main( String args[] )
   {
      // instantiate BasePlusCommissionEmployee4 object
      BasePlusCommissionEmployee4 employee =
         new BasePlusCommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
```

Fig. 9.14 | Superclass `private` instance variables are accessible to a subclass via `public` or `protected` methods inherited by the subclass. (Part 1 of 2.)

```java
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
      System.out.printf( "%s %.2f\n", "Base salary is",
         employee.getBaseSalary() );

      employee.setBaseSalary( 1000 ); // set base salary

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString",
         employee.toString() );
   } // end main
} // end class BasePlusCommissionEmployeeTest4
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.14 | Superclass `private` instance variables are accessible to a subclass via `public` or `protected` methods inherited by the subclass. (Part 2 of 2.)

lesson 2

Object-Oriented Programming: Polymorphism

Based on Chapter 10 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Understand the concept of polymorphism.
- Use overridden methods to effect polymorphism.
- Distinguish between abstract and concrete classes.
- Declare abstract methods to create abstract classes.
- Learn how polymorphism makes systems extensible and maintainable.
- Determine an object's type at execution time.
- Declare and implement interfaces.

Figures

```java
// Fig. 10.1: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.

public class PolymorphismTest
{
   public static void main( String args[] )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 2.)

```
      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee4 basePlusCommissionEmployee =
         new BasePlusCommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call CommissionEmployee3's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee4's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

      // invoke toString on subclass object using superclass variable
      CommissionEmployee3 commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s %s:\n\n%s\n",
         "Call BasePlusCommissionEmployee4's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString() );
   } // end main
} // end class PolymorphismTest
```

```
Call CommissionEmployee3's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 2.)

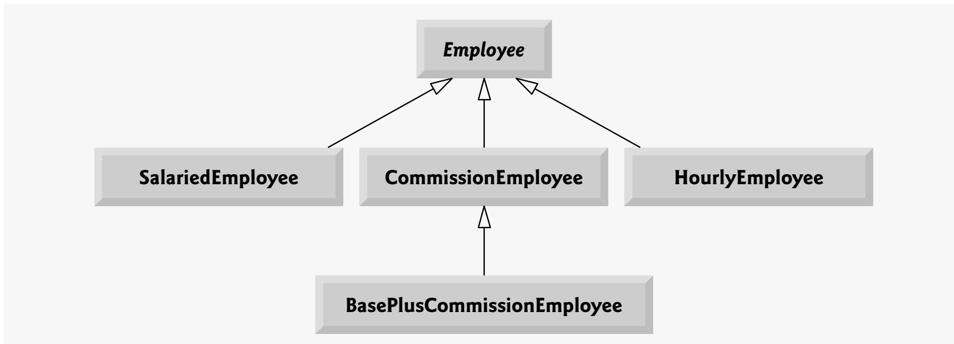

Fig. 10.2 | `Employee` hierarchy UML class diagram.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	if hours <= 40 wage * hours else if hours > 40 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

```
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
```

Fig. 10.4 | `Employee` abstract superclass. (Part 1 of 3.)

```java
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
```

Fig. 10.4 | `Employee` abstract superclass. (Part 2 of 3.)

```java
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // abstract method overridden by subclasses
   public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

Fig. 10.4 | `Employee` abstract superclass. (Part 3 of 3.)

```java
// Fig. 10.5: SalariedEmployee.java
// SalariedEmployee class extends Employee.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; override abstract method earnings in Employee
   public double earnings()
   {
      return getWeeklySalary();
   } // end method earnings

   // return String representation of SalariedEmployee object
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. 10.5 | `SalariedEmployee` class derived from `Employee`.

```java
// Fig. 10.6: HourlyEmployee.java
// HourlyEmployee class extends Employee.

public class HourlyEmployee extends Employee
{
   private double wage; // wage per hour
   private double hours; // hours worked for week

   // five-argument constructor
   public HourlyEmployee( String first, String last, String ssn,
      double hourlyWage, double hoursWorked )
   {
      super( first, last, ssn );
      setWage( hourlyWage ); // validate hourly wage
      setHours( hoursWorked ); // validate hours worked
   } // end five-argument HourlyEmployee constructor

   // set wage
   public void setWage( double hourlyWage )
   {
      wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
   } // end method setWage

   // return wage
   public double getWage()
   {
      return wage;
   } // end method getWage

   // set hours worked
   public void setHours( double hoursWorked )
   {
      hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
         hoursWorked : 0.0;
   } // end method setHours

   // return hours worked
   public double getHours()
   {
      return hours;
   } // end method getHours

   // calculate earnings; override abstract method earnings in Employee
   public double earnings()
   {
      if ( getHours() <= 40 ) // no overtime
         return getWage() * getHours();
      else
         return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
   } // end method earnings

```

Fig. 10.6 | `HourlyEmployee` class derived from `Employee`. (Part 1 of 2.)

```java
   // return String representation of HourlyEmployee object
   public String toString()
   {
      return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours() );
   } // end method toString
} // end class HourlyEmployee
```

Fig. 10.6 | `HourlyEmployee` class derived from `Employee`. (Part 2 of 2.)

```java
// Fig. 10.7: CommissionEmployee.java
// CommissionEmployee class extends Employee.

public class CommissionEmployee extends Employee
{
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      super( first, last, ssn );
      setGrossSales( sales );
      setCommissionRate( rate );
   } // end five-argument CommissionEmployee constructor

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales
```

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 1 of 2.)

```java

   // calculate earnings; override abstract method earnings in Employee
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee object
   public String toString()
   {
      return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
         "commission employee", super.toString(),
         "gross sales", getGrossSales(),
         "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee
```

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 2 of 2.)

```java
// Fig. 10.8: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class extends CommissionEmployee.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings
```

Fig. 10.8 | `BasePlusCommissionEmployee` class derived from `CommissionEmployee`. (Part 1 of 2.)

```java

   // return String representation of BasePlusCommissionEmployee object
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

Fig. 10.8 | BasePlusCommissionEmployee class derived from CommissionEmployee. (Part 2 of 2.)

```java
// Fig. 10.9: PayrollSystemTest.java
// Employee hierarchy test program.

public class PayrollSystemTest
{
   public static void main( String args[] )
   {
      // create subclass objects
      SalariedEmployee salariedEmployee =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee(
         "Sue", "Jones", "333-33-3333", 10000, .06 );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

      System.out.println( "Employees processed individually:\n" );

      System.out.printf( "%s\n%s: $%,.2f\n\n",
         salariedEmployee, "earned", salariedEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         hourlyEmployee, "earned", hourlyEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         commissionEmployee, "earned", commissionEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         basePlusCommissionEmployee,
         "earned", basePlusCommissionEmployee.earnings() );

      // create four-element Employee array
      Employee employees[] = new Employee[ 4 ];

      // initialize array with Employees
      employees[ 0 ] = salariedEmployee;
      employees[ 1 ] = hourlyEmployee;
      employees[ 2 ] = commissionEmployee;
      employees[ 3 ] = basePlusCommissionEmployee;
```

Fig. 10.9 | Employee class hierarchy test program. (Part 1 of 3.)

```java

      System.out.println( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      for ( Employee currentEmployee : employees )
      {
         System.out.println( currentEmployee ); // invokes toString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee instanceof BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            double oldBaseSalary = employee.getBaseSalary();
            employee.setBaseSalary( 1.10 * oldBaseSalary );
            System.out.printf(
               "new base salary with 10%% increase is: $%,.2f\n",
               employee.getBaseSalary() );
         } // end if

         System.out.printf(
            "earned $%,.2f\n\n", currentEmployee.earnings() );
      } // end for

      // get type name of each object in employees array
      for ( int j = 0; j < employees.length; j++ )
         System.out.printf( "Employee %d is a %s\n", j,
            employees[ j ].getClass().getName() );
   } // end main
} // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

Fig. 10.9 | `Employee` class hierarchy test program. (Part 2 of 3.)

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. 10.9 | `Employee` class hierarchy test program. (Part 3 of 3.)

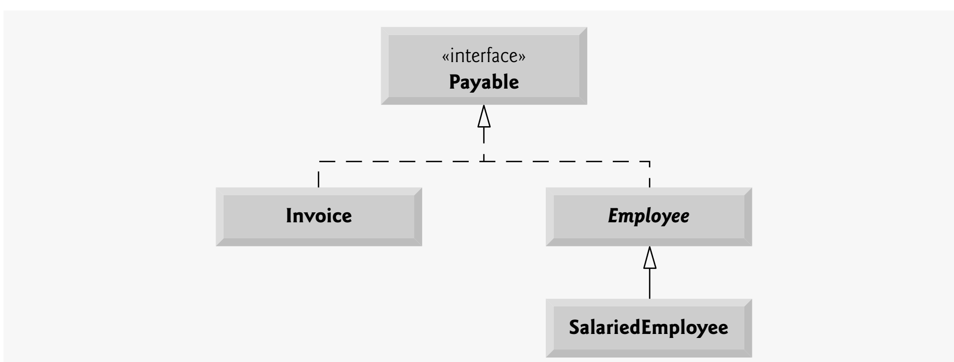

Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

```java
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Fig. 10.11 | `Payable` interface declaration.

```java
// Fig. 10.12: Invoice.java
// Invoice class implements Payable.

public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part;
   } // end method setPartNumber

   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description;
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
   } // end method setQuantity

```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 1 of 2.)

```java
   // get quantity
   public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

   // set price per item
   public void setPricePerItem( double price )
   {
      pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
   } // end method setPricePerItem

   // get price per item
   public double getPricePerItem()
   {
      return pricePerItem;
   } // end method getPricePerItem

   // return String representation of Invoice object
   public String toString()
   {
      return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
         "invoice", "part number", getPartNumber(), getPartDescription(),
         "quantity", getQuantity(), "price per item", getPricePerItem() );
   } // end method toString

   // method required to carry out contract with interface Payable
   public double getPaymentAmount()
   {
      return getQuantity() * getPricePerItem(); // calculate total cost
   } // end method getPaymentAmount
} // end class Invoice
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 2 of 2.)

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 1 of 2.)

```java

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 2 of 2.)

```java
// Fig. 10.14: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

   // return String representation of SalariedEmployee object
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`.

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

public class PayableInterfaceTest
{
```

Fig. 10.15 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 1 of 2.)

```java
   public static void main( String args[] )
   {
      // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 2.)

lesson 3

Introduction to Graphical User Interfaces (GUIs) and Event Handling

Based on Chapter 11 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Build basic GUIs and handle events generated by user interactions with GUIs.
- Create and use nested classes and anonymous inner classes, and understand the relationship between such classes and their outer classes.
- Learn the packages containing GUI components, event-handling classes and interfaces.
- Create and manipulate text fields, buttons, labels, comboboxes and panels.
- Handle mouse events.

Figures

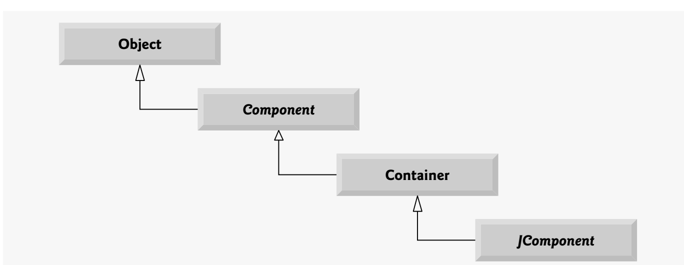

Fig. 11.5 | Common superclasses of many of the Swing components.

```java
// Fig. 11.9: TextFieldFrame.java
// Demonstrating the JTextField class.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
```

Fig. 11.9 | `JTextFields` and `JPasswordFields`. (Part 1 of 3.)

```java
import javax.swing.JOptionPane;

public class TextFieldFrame extends JFrame
{
   private JTextField textField1; // text field with set size
   private JTextField textField2; // text field constructed with text
   private JTextField textField3; // text field with text and size
   private JPasswordField passwordField; // password field with text

   // TextFieldFrame constructor adds JTextFields to JFrame
   public TextFieldFrame()
   {
      super( "Testing JTextField and JPasswordField" );
      setLayout( new FlowLayout() ); // set frame layout

      // construct textfield with 10 columns
      textField1 = new JTextField( 10 );
      add( textField1 ); // add textField1 to JFrame

      // construct textfield with default text
      textField2 = new JTextField( "Enter text here" );
      add( textField2 ); // add textField2 to JFrame

      // construct textfield with default text and 21 columns
      textField3 = new JTextField( "Uneditable text field", 21 );
      textField3.setEditable( false ); // disable editing
      add( textField3 ); // add textField3 to JFrame

      // construct passwordfield with default text
      passwordField = new JPasswordField( "Hidden text" );
      add( passwordField ); // add passwordField to JFrame

      // register event handlers
      TextFieldHandler handler = new TextFieldHandler();
      textField1.addActionListener( handler );
      textField2.addActionListener( handler );
      textField3.addActionListener( handler );
      passwordField.addActionListener( handler );
   } // end TextFieldFrame constructor

   // private inner class for event handling
   private class TextFieldHandler implements ActionListener
   {
      // process text field events
      public void actionPerformed( ActionEvent event )
      {
         String string = ""; // declare string to display

         // user pressed Enter in JTextField textField1
         if ( event.getSource() == textField1 )
            string = String.format( "textField1: %s",
               event.getActionCommand() );
```

Fig. 11.9 | `JTextFields` and `JPasswordFields`. (Part 2 of 3.)

```java

            // user pressed Enter in JTextField textField2
            else if ( event.getSource() == textField2 )
               string = String.format( "textField2: %s",
                  event.getActionCommand() );

            // user pressed Enter in JTextField textField3
            else if ( event.getSource() == textField3 )
               string = String.format( "textField3: %s",
                  event.getActionCommand() );

            // user pressed Enter in JTextField passwordField
            else if ( event.getSource() == passwordField )
               string = String.format( "passwordField: %s",
                  new String( passwordField.getPassword() ) );

            // display JTextField content
            JOptionPane.showMessageDialog( null, string );
         } // end method actionPerformed
      } // end private inner class TextFieldHandler
   } // end class TextFieldFrame
```

Fig. 11.9 | `JTextFields` and `JPasswordFields`. (Part 3 of 3.)

```java
// Fig. 11.10: TextFieldTest.java
// Testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest
{
   public static void main( String args[] )
   {
      TextFieldFrame textFieldFrame = new TextFieldFrame();
      textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textFieldFrame.setSize( 350, 100 ); // set frame size
      textFieldFrame.setVisible( true ); // display frame
   } // end main
} // end class TextFieldTest
```

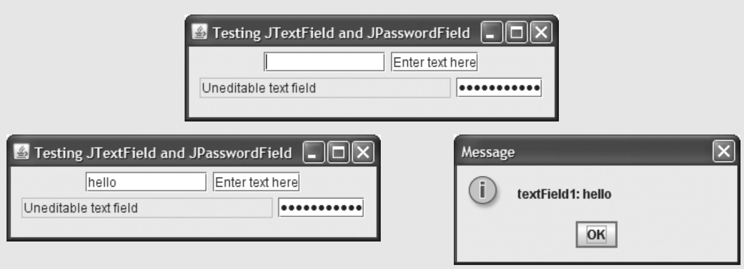

Fig. 11.10 | Test class for `TextFieldFrame`. (Part 1 of 2.)

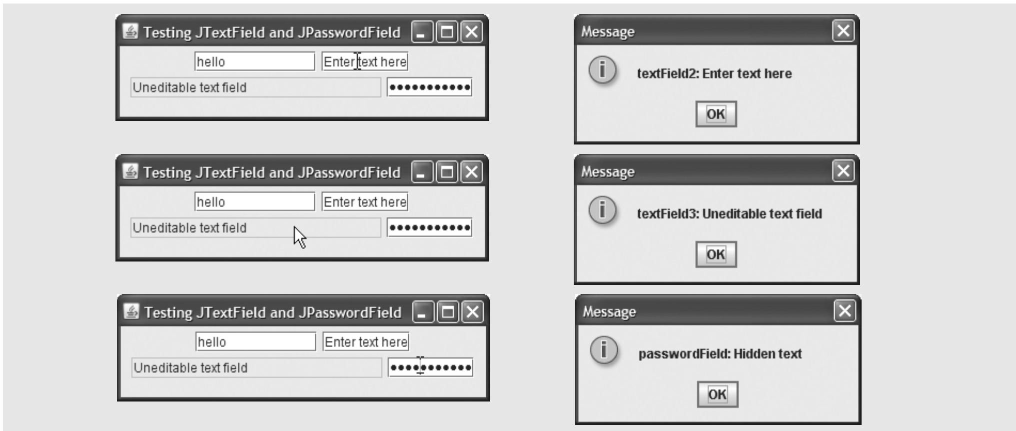

Fig. 11.10 | Test class for `TextFieldFrame`. (Part 2 of 2.)

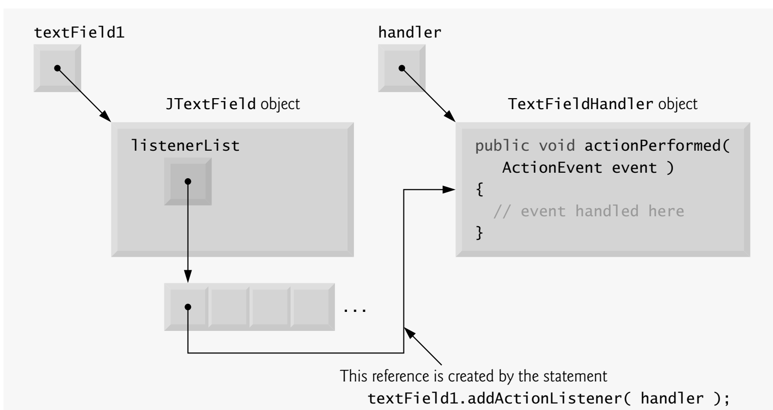

Fig. 11.13 | Event registration for `JTextField` `textField1`.

```java
// Fig. 11.15: ButtonFrame.java
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame
{
   private JButton plainJButton; // button with just text
   private JButton fancyJButton; // button with icons
```

Fig. 11.15 | Command buttons and action events. (Part 1 of 2.)

```java

   // ButtonFrame adds JButtons to JFrame
   public ButtonFrame()
   {
      super( "Testing Buttons" );
      setLayout( new FlowLayout() ); // set frame layout

      plainJButton = new JButton( "Plain Button" ); // button with text
      add( plainJButton ); // add plainJButton to JFrame

      Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
      Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
      fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
      fancyJButton.setRolloverIcon( bug2 ); // set rollover image
      add( fancyJButton ); // add fancyJButton to JFrame

      // create new ButtonHandler for button event handling
      ButtonHandler handler = new ButtonHandler();
      fancyJButton.addActionListener( handler );
      plainJButton.addActionListener( handler );
   } // end ButtonFrame constructor

   // inner class for button event handling
   private class ButtonHandler implements ActionListener
   {
      // handle button event
      public void actionPerformed( ActionEvent event )
      {
         JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
            "You pressed: %s", event.getActionCommand() ) );
      } // end method actionPerformed
   } // end private inner class ButtonHandler
} // end class ButtonFrame
```

Fig. 11.15 | Command buttons and action events. (Part 2 of 2.)

```java
// Fig. 11.16: ButtonTest.java
// Testing ButtonFrame.
import javax.swing.JFrame;

public class ButtonTest
{
   public static void main( String args[] )
   {
      ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
      buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      buttonFrame.setSize( 275, 110 ); // set frame size
      buttonFrame.setVisible( true ); // display frame
   } // end main
} // end class ButtonTest
```

Fig. 11.16 | Test class for `ButtonFrame`. (Part 1 of 2.)

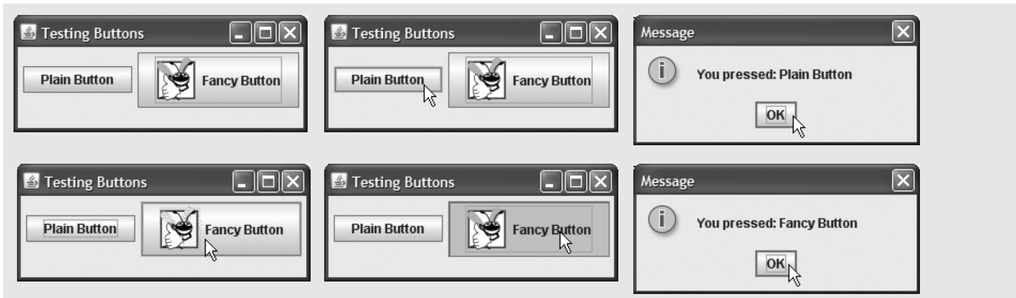

Fig. 11.16 | Test class for `ButtonFrame`. (Part 2 of 2.)

```java
// Fig. 11.21: ComboBoxFrame.java
// Using a JComboBox to select an image to display.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class ComboBoxFrame extends JFrame
{
   private JComboBox imagesJComboBox; // combobox to hold names of icons
   private JLabel label; // label to display selected icon

   private String names[] =
      { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
   private Icon icons[] = {
      new ImageIcon( getClass().getResource( names[ 0 ] ) ),
      new ImageIcon( getClass().getResource( names[ 1 ] ) ),
      new ImageIcon( getClass().getResource( names[ 2 ] ) ),
      new ImageIcon( getClass().getResource( names[ 3 ] ) ) };

   // ComboBoxFrame constructor adds JComboBox to JFrame
   public ComboBoxFrame()
   {
      super( "Testing JComboBox" );
      setLayout( new FlowLayout() ); // set frame layout

      imagesJComboBox = new JComboBox( names ); // set up JComboBox
      imagesJComboBox.setMaximumRowCount( 3 ); // display three rows

      imagesJComboBox.addItemListener(
         new ItemListener() // anonymous inner class
         {
            // handle JComboBox event
```

Fig. 11.21 | `JComboBox` that displays a list of image names. (Part 1 of 2.)

```
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether checkbox selected
               if ( event.getStateChange() == ItemEvent.SELECTED )
                  label.setIcon( icons[
                     imagesJComboBox.getSelectedIndex() ] );
            } // end method itemStateChanged
         } // end anonymous inner class
      ); // end call to addItemListener

      add( imagesJComboBox ); // add combobox to JFrame
      label = new JLabel( icons[ 0 ] ); // display first icon
      add( label ); // add label to JFrame
   } // end ComboBoxFrame constructor
} // end class ComboBoxFrame
```

Fig. 11.21 | `JComboBox` that displays a list of image names. (Part 2 of 2.)

```
// Fig. 11.22: ComboBoxTest.java
// Testing ComboBoxFrame.
import javax.swing.JFrame;

public class ComboBoxTest
{
   public static void main( String args[] )
   {
      ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
      comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      comboBoxFrame.setSize( 350, 150 ); // set frame size
      comboBoxFrame.setVisible( true ); // display frame
   } // end main
} // end class ComboBoxTest
```

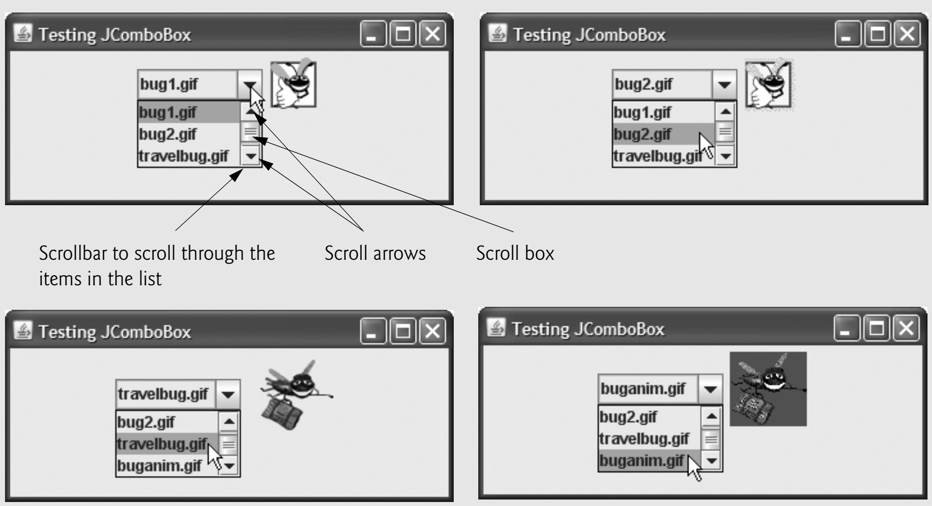

Fig. 11.22 | Test class for `ComboBoxFrame`.

```java
// Fig. 11.28: MouseTrackerFrame.java
// Demonstrating mouse events.
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame
{
   private JPanel mousePanel; // panel in which mouse events will occur
   private JLabel statusBar; // label that displays event information

   // MouseTrackerFrame constructor sets up GUI and
   // registers mouse event handlers
   public MouseTrackerFrame()
   {
      super( "Demonstrating Mouse Events" );

      mousePanel = new JPanel(); // create panel
      mousePanel.setBackground( Color.WHITE ); // set background color
      add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame

      statusBar = new JLabel( "Mouse outside JPanel" );
      add( statusBar, BorderLayout.SOUTH ); // add label to JFrame

      // create and register listener for mouse and mouse motion events
      MouseHandler handler = new MouseHandler();
      mousePanel.addMouseListener( handler );
      mousePanel.addMouseMotionListener( handler );
   } // end MouseTrackerFrame constructor

   private class MouseHandler implements MouseListener,
      MouseMotionListener
   {
      // MouseListener event handlers
      // handle event when mouse released immediately after press
      public void mouseClicked( MouseEvent event )
      {
         statusBar.setText( String.format( "Clicked at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseClicked

      // handle event when mouse pressed
      public void mousePressed( MouseEvent event )
      {
         statusBar.setText( String.format( "Pressed at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mousePressed
```

Fig. 11.28 | Mouse event handling. (Part 1 of 2.)

```java

      // handle event when mouse released after dragging
      public void mouseReleased( MouseEvent event )
      {
         statusBar.setText( String.format( "Released at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseReleased

      // handle event when mouse enters area
      public void mouseEntered( MouseEvent event )
      {
         statusBar.setText( String.format( "Mouse entered at [%d, %d]",
            event.getX(), event.getY() ) );
         mousePanel.setBackground( Color.GREEN );
      } // end method mouseEntered

      // handle event when mouse exits area
      public void mouseExited( MouseEvent event )
      {
         statusBar.setText( "Mouse outside JPanel" );
         mousePanel.setBackground( Color.WHITE );
      } // end method mouseExited

      // MouseMotionListener event handlers
      // handle event when user drags mouse with button pressed
      public void mouseDragged( MouseEvent event )
      {
         statusBar.setText( String.format( "Dragged at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseDragged

      // handle event when user moves mouse
      public void mouseMoved( MouseEvent event )
      {
         statusBar.setText( String.format( "Moved at [%d, %d]",
            event.getX(), event.getY() ) );
      } // end method mouseMoved
   } // end inner class MouseHandler
} // end class MouseTrackerFrame
```

Fig. 11.28 | Mouse event handling. (Part 2 of 2.)

```java
// Fig. 11.29: MouseTrackerFrame.java
// Testing MouseTrackerFrame.
import javax.swing.JFrame;

public class MouseTracker
{
   public static void main( String args[] )
   {
      MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
      mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

Fig. 11.29 | Test class for `MouseTrackerFrame`. (Part 1 of 2.)

```
      mouseTrackerFrame.setSize( 300, 100 ); // set frame size
      mouseTrackerFrame.setVisible( true ); // display frame
   } // end main
} // end class MouseTracker
```

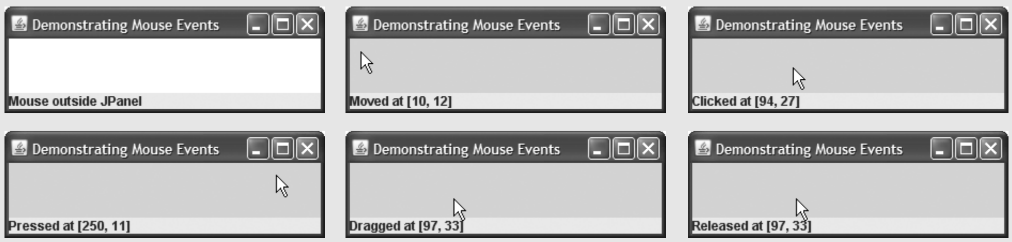

Fig. 11.29 | Test class for `MouseTrackerFrame`. (Part 2 of 2.)

```
// Fig. 11.31: MouseDetailsFrame.java
// Demonstrating mouse clicks and distinguishing between mouse buttons.
import java.awt.BorderLayout;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MouseDetailsFrame extends JFrame
{
   private String details; // String representing
   private JLabel statusBar; // JLabel that appears at bottom of window

   // constructor sets title bar String and register mouse listener
   public MouseDetailsFrame()
   {
      super( "Mouse clicks and buttons" );

      statusBar = new JLabel( "Click the mouse" );
      add( statusBar, BorderLayout.SOUTH );
      addMouseListener( new MouseClickHandler() ); // add handler
   } // end MouseDetailsFrame constructor

   // inner class to handle mouse events
   private class MouseClickHandler extends MouseAdapter
   {
      // handle mouse-click event and determine which button was pressed
      public void mouseClicked( MouseEvent event )
      {
         int xPos = event.getX(); // get x-position of mouse
         int yPos = event.getY(); // get y-position of mouse

         details = String.format( "Clicked %d time(s)",
            event.getClickCount() );
```

Fig. 11.31 | Left, center and right mouse-button clicks. (Part 1 of 2.)

```java

         if ( event.isMetaDown() ) // right mouse button
            details += " with right mouse button";
         else if ( event.isAltDown() ) // middle mouse button
            details += " with center mouse button";
         else // left mouse button
            details += " with left mouse button";

         statusBar.setText( details ); // display message in statusBar
      } // end method mouseClicked
   } // end private inner class MouseClickHandler
} // end class MouseDetailsFrame
```

Fig. 11.31 | Left, center and right mouse-button clicks. (Part 2 of 2.)

```java
// Fig. 11.32: MouseDetails.java
// Testing MouseDetailsFrame.
import javax.swing.JFrame;

public class MouseDetails
{
   public static void main( String args[] )
   {
      MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
      mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      mouseDetailsFrame.setSize( 400, 150 ); // set frame size
      mouseDetailsFrame.setVisible( true ); // display frame
   } // end main
} // end class MouseDetails
```

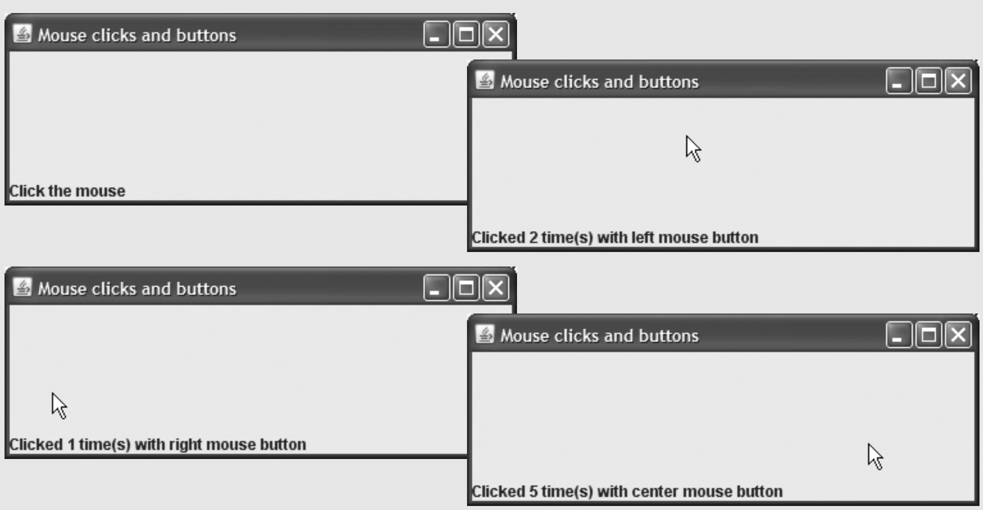

Fig. 11.32 | Test class for `MouseDetailsFrame`.

4 lesson

Exception Handling

Based on Chapter 13 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`).

Learning Objectives

- Use `try`, `throw` and `catch` to detect, indicate and handle exceptions, respectively.
- Use the `finally` block to release resources.
- Understand how stack unwinding enables exceptions not caught in one scope to be caught in another scope.
- Learn how stack traces help in debugging.
- Learn how exceptions are arranged in a class hierarchy.
- Determine the exceptions thrown by a method.
- Distinguish between checked and unchecked exceptions.
- Create chained exceptions that maintain complete stack-trace information.

Figures

```
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
```

Fig. 13.1 | Integer division without exception handling. (Part 1 of 2.)

```
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:20)
```

Fig. 13.1 | Integer division without exception handling. (Part 2 of 2.)

```
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient
```

Fig. 13.2 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 1 of 3.)

```java

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nException: %s\n",
               inputMismatchException );
            scanner.nextLine(); // discard input so user can try again
            System.out.println(
               "You must enter integers. Please try again.\n" );
         } // end catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
         } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 13.2 | Handling `ArithmeticException`s and `InputMismatchException`s. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 13.2 | Handling `ArithmeticException`s and `InputMismatchException`s. (Part 3 of 3.)

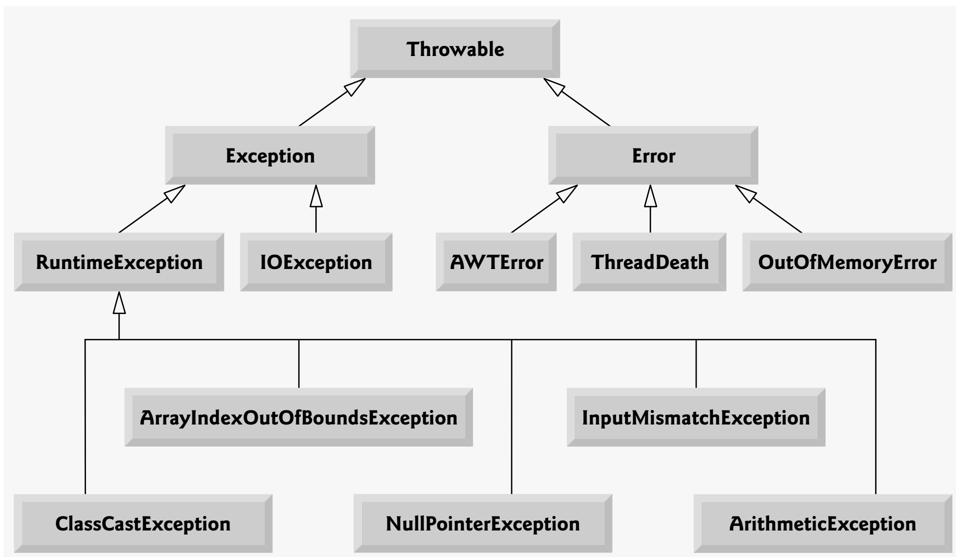

Fig. 13.3 | Portion of class `Throwable`'s inheritance hierarchy.

```
try
{
   statements
   resource-acquisition statements
} // end try
catch ( AKindOfException exception1 )
{
   exception-handling statements
} // end catch
...
catch ( AnotherKindOfException exception2 )
{
   exception-handling statements
} // end catch
finally
{
   statements
   resource-release statements
} // end finally
```

Fig. 13.4 | A `try` statement with a `finally` block.

```java
// Fig. 13.5: UsingExceptions.java
// Demonstration of the try...catch...finally exception handling
// mechanism.

public class UsingExceptions
{
   public static void main( String args[] )
   {
      try
      {
         throwException(); // call method throwException
      } // end try
      catch ( Exception exception ) // exception thrown by throwException
      {
         System.err.println( "Exception handled in main" );
      } // end catch

      doesNotThrowException();
   } // end main

   // demonstrate try...catch...finally
   public static void throwException() throws Exception
   {
      try // throw an exception and immediately catch it
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // generate exception
      } // end try
      catch ( Exception exception ) // catch exception thrown in try
      {
         System.err.println(
            "Exception handled in method throwException" );
         throw exception; // rethrow for further processing

         // any code here would not be reached

      } // end catch
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println( "Finally executed in throwException" );
      } // end finally

      // any code here would not be reached, exception rethrown in catch

   } // end method throwException

   // demonstrate finally when no exception occurs
   public static void doesNotThrowException()
   {
      try // try block does not throw an exception
      {
         System.out.println( "Method doesNotThrowException" );
      } // end try
```

Fig. 13.5 | `try`...`catch`...`finally` exception-handling mechanism. (Part 1 of 2.)

```
         catch ( Exception exception ) // does not execute
         {
            System.err.println( exception );
         } // end catch
         finally // executes regardless of what occurs in try...catch
         {
            System.err.println(
               "Finally executed in doesNotThrowException" );
         } // end finally

         System.out.println( "End of method doesNotThrowException" );
      } // end method doesNotThrowException
   } // end class UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Fig. 13.5 | `try...catch...finally` exception-handling mechanism. (Part 2 of 2.)

```
// Fig. 13.6: UsingExceptions.java
// Demonstration of stack unwinding.

public class UsingExceptions
{
   public static void main( String args[] )
   {
      try // call throwException to demonstrate stack unwinding
      {
         throwException();
      } // end try
      catch ( Exception exception ) // exception thrown in throwException
      {
         System.err.println( "Exception handled in main" );
      } // end catch
   } // end main

   // throwException throws exception that is not caught in this method
   public static void throwException() throws Exception
   {
      try // throw an exception and catch it in main
      {
         System.out.println( "Method throwException" );
         throw new Exception(); // generate exception
      } // end try
      catch ( RuntimeException runtimeException ) // catch incorrect type
      {
```

Fig. 13.6 | Stack unwinding. (Part 1 of 2.)

```java
            System.err.println(
               "Exception handled in method throwException" );
         } // end catch
         finally // finally block always executes
         {
            System.err.println( "Finally is always executed" );
         } // end finally
      } // end method throwException
   } // end class UsingExceptions
```

```
Method throwException
Finally is always executed
Exception handled in main
```

Fig. 13.6 | Stack unwinding. (Part 2 of 2.)

```java
// Fig. 13.8: UsingChainedExceptions.java
// Demonstrating chained exceptions.

public class UsingChainedExceptions
{
   public static void main( String args[] )
   {
      try
      {
         method1(); // call method1
      } // end try
      catch ( Exception exception ) // exceptions thrown from method1
      {
         exception.printStackTrace();
      } // end catch
   } // end main

   // call method2; throw exceptions back to main
   public static void method1() throws Exception
   {
      try
      {
         method2(); // call method2
      } // end try
      catch ( Exception exception ) // exception thrown from method2
      {
         throw new Exception( "Exception thrown in method1", exception );
      } // end try
   } // end method method1

   // call method3; throw exceptions back to method1
   public static void method2() throws Exception
   {
      try
      {
```

Fig. 13.8 | Chained exceptions. (Part 1 of 2.)

```
         method3(); // call method3
      } // end try
      catch ( Exception exception ) // exception thrown from method3
      {
         throw new Exception( "Exception thrown in method2", exception );
      } // end catch
   } // end method method2

   // throw Exception back to method2
   public static void method3() throws Exception
   {
      throw new Exception( "Exception thrown in method3" );
   } // end method method3
} // end class UsingChainedExceptions
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:10)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:40)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:23)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:47)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:36)
        ... 2 more
```

Fig. 13.8 | Chained exceptions. (Part 2 of 2.)

lesson 5

The Java Collections Framework

Based on Chapter 19 of *Java How to Program, 7/e* (`http://www.deitel.com/books/jhtp7/`)

Learning Objectives

- Learn what collections are.
- Use class `Arrays` for array manipulations.
- Use the collections framework implementations.
- Use the collections framework algorithms to manipulate collections.
- Use the collections framework interfaces to program with collections polymorphically.
- Use iterators to "walk through" a collection.
- Learn about the synchronization and modifiability wrappers for collections.

Figures

Note: The figures for this lesson are located in a PDF file on the DVD.