Java Programming

Java™ Programming, Second Edition

Joyce Farrell

THOMSON
COURSE TECHNOLOGY ™

Australia • Canada • Mexico • Singapore • Spain • United Kingdom • United States

THOMSON
COURSE TECHNOLOGY™

Authorised English Reprint from the English Language edition.

Java Programming (Second Edition)

ENGLISH language edition published by Galatea Training Services Limited.

ISBN 0-954307-1-9

Printed by B & Jo Enterprise Pte Ltd in Singapore.

BRIEF

Contents

TABLE OF Contents

Preface

Java Programming, Second Edition, provides the beginning programmer with a guide to developing applications and applets using the Java programming language. Java is popular among professional programmers because it can be used to build visually interesting GUI and Web-based applications. Java also provides an excellent environment for the beginning programmer—a student can quickly build useful programs while learning the basics of structured and object-oriented programming techniques.

This textbook assumes little or no programming experience. The writing is non-technical and emphasizes good programming practices. The examples are business examples; they do not assume mathematical background beyond high school business math. Additionally, the examples illustrate one or two major points; they do not contain so many features that one becomes lost following extraneous details. This book provides a solid background in good object-oriented programming techniques and introduces you to object-oriented terminology using clear, familiar language.

ORGANIZATION AND COVERAGE

Java Programming, Second Edition, presents Java programming concepts, enforcing good style, logical thinking, and the object-oriented paradigm. Objects are covered right from the beginning, earlier than in many other textbooks. You create your first Java program in Chapter 1. Chapters 2, 3 and 4 increase your understanding of how data, classes, objects and methods interact in an object-oriented environment.

Chapters 5 and 6 explore decision and repetition structures, which are the backbone of programming logic and essential to creating useful programs in any language. You learn the special considerations of String and array manipulation in Chapters 7 and 8.

Beginning with Chapter 9 you will write applets which are mini-programs meant to run in a browser. In Chapter 10 you learn to add graphics, images and sound to your applets. Chapters 11 and 12 provide thorough coverage of inheritance, the object-oriented concept that allows you to develop new objects quickly by adapting the features of existing ones. In Chapters 13 and 14 you begin to use Swing components–Java's visually pleasing user-friendly widgets.

Exception handling, the object-oriented methods of error control, is covered in Chapter 15. Chapter 16 teaches you to save and retrieve data from files. In Chapter 17 you learn about threads and how to create programs in which multiple activities take place concurrently allowing you to create programs that contain animation.

Java Programming, Second Edition, combines text explanation with step-by-step exercises that illustrate the concepts just learned, reinforcing your understanding and improving retention. Creating the step-by-step examples also provides you with a successful experience in the language; finishing the examples provides you with models for your own creations.

Using *Java Programming, Second Edition,* allows you to build applications and applets from the bottom up, rather than starting with pre-existing objects. This facilitates a deeper understanding of the concepts used in object-oriented programming, and engenders appreciation for the existing objects you use as your knowledge of the language advances. When you complete this book you will know how to modify and create simple Java programs and will have the tools to create more complex examples. You also will have a fundamental knowledge of object-oriented programming which will serve you well in advanced Java courses or in studying other object-oriented languages like C++, C# and Visual Basic.

FEATURES

Java Programming, Second Edition, is a superior textbook because it includes the following features:

- Objectives: Each chapter begins with a list of objectives so you know the topics that will be presented in the chapter. In addition to providing a quick reference to topics covered, this feature provides a useful study aid.
- Tips: These notes provide additional information—for example, an alternative method of performing a procedure, another term for a concept, background information on a technique, or a common error to avoid.
- Summaries: Following each chapter is a summary that recaps the programming concepts and techniques covered in the chapter. This feature provides a concise means for you to recap and check your understanding of the main points in each chapter.
- Review Questions: End-of-chapter assessment begins with a set of approximately 20 review questions that reinforce the main ideas introduced in each chapter. These questions ensure that you have mastered the concepts and understand the information you have learned.
- Exercises: Each chapter concludes with meaningful programming exercises that provide additional practice of the skills and concepts you learned in the chapter. These exercises increase in difficulty and are designed to allow you to explore logical programming concepts.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, especially Virgil Brewer of Eastern Kentucky University. Virgil created much of the new material for the second edition of this book, including all the material on Swing components and adding graphics, images and sound to programs. Additionally he contributed new examples and programming exercises.

I also thank Laurie Brown, Developmental Editor. She had to manage a variety of challenges to produce this book, yet she remained cheerful and professional throughout the process. Thanks also to Tricia Boyle, Product Manager, who kept everyone on track. Thanks to Kristen Duerr, Senior Vice President and Publisher; Jennifer Muroff, Senior Editor; Anne Valsangiacomo, Production Editor; and Shawn Day and Chris Scriver, Quality Assurance Testers, who provided thorough examination of the manuscript and consistently valuable suggestions. I am grateful to be able to work with so many fine people who are dedicated to producing quality instructional materials.

I am also grateful to the many reviewers who provided helpful comments and encouragement during this book's development, including Eric Berkowitz, Roosevelt University; Nelly Cardinale, Brevard Community College; R. Scott Cost, University of Maryland - Baltimore County; Jeff Howard, Finger Lakes Community College; Steve Hunt, Mid-State Technical College; Daniela Marghitu, Auburn University; Carmen Montanez-Juenke, College of the Sequoias; Charlie Morgan, Coleman College; Alok Pandey, Community College of Southern Nevada; Ed Sullivan, Indiana University—Purdue University at Indianapolis; and Dwight Watt, Athens Technical College.

Thanks, too, to my husband, Geoff, who sustained me with a mean bolognaise so I could work on the next chapter. Finally, this book is dedicated to Andrea and Audrey.

Joyce Farrell

Read This Before You Begin

The following information will help you as you prepare to use this textbook.

To the User of the Data Files

To complete the steps and projects, you will need data files that have been created specifically for this book. You can obtain the files electronically from the Course Technology Web site at *www.course.com* by searching for this book title. Note that you can use a computer in your school lab or your own computer to complete the Exercises in this book.

Using Your Own Computer

To use your own computer to complete the steps and Exercises, you will need the following:

- **Software.** Java 2 SDK v1.4.0; Internet Explorer 6.0; and Notepad.
- **Hardware.** A Pentium II-class processor, 450 MHz or higher, personal computer, and Windows NT, Windows 2000, or Windows XP.
- **Data Files.** You will be able to complete some of the projects using no special files. However, for many, you will want to use data files that you can obtain electronically from the Course Technology Web site at *www.course.com* by searching for this book title.

A Note on Operating Systems

The activities, figures and projects for *Java Programming, Second Edition* were created using Windows 2000 and tested in both Windows 2000 and Windows XP. While you can use Windows NT, Windows 2000, or Windows XP to complete the activities and projects, you may notice some minor differences in your output if you are not using Windows 2000.

Visit Our World Wide Web Site

Additional materials designed especially for this book might be available for your course. Periodically search *www.course.com* for more details.

CHAPTER

1 CREATING YOUR FIRST JAVA PROGRAM

In this chapter, you will:

- ♦ Learn about programming
- ♦ Understand object-oriented programming concepts
- ♦ Learn about the Java programming language
- ♦ Start a Java program
- ♦ Add comments to a Java program
- ♦ Run a Java program
- ♦ Modify a Java program

As you read your e-mail, a sinking feeling descends on you. There's no denying the message: "Please see me in my office as soon as you are free—Lynn Greenbrier." Lynn Greenbrier is the head of programming for Event Handlers Incorporated, and you have worked for her as an intern for only two weeks. Event Handlers manages the details of private and corporate parties; every client has different needs, and the events are interesting and exciting.

"Did I do something wrong?" you ask as you enter her office. "Are you going to fire me?"

Almost like a mind reader, Lynn stands to greet you and says, "Please wipe that worried look off your face! I want to see if you are interested in a new challenge. Our programming department is going to create several new programs in the next few months. We've decided that the Java programming language is the way to go. It's object-oriented, platform independent, and perfect for applications on the World Wide Web, which is where we want to expand our marketing efforts."

"I'm not sure what 'object-oriented' and 'platform independent' mean," you say, "but I've always been interested in computers, and I'd love to learn more about programming."

"Based on your aptitude tests, you're perfect for programming," Lynn says. "Let's get started now. I'll describe the basics to you."

Learning About Programming

A computer **program** is simply a set of instructions that you write to tell a computer what to do. Computers are constructed from circuitry that consists of small on/off switches, so you could write a computer program by writing something along the following lines:

```
first switch—on
second switch—off
third switch—off
fourth switch—on
```

Your program could go on and on, for several thousand switches. A program written in this style is written in **machine language**, which is the most basic circuitry-level language. The problems with this approach lie in keeping track of the many switches involved in programming any worthwhile task, and in discovering the errant switch or switches if the program does not operate as expected. Additionally, the number and location of switches varies from computer to computer, which means that you would need to customize a machine language program for every type of machine on which you want the program to run.

Fortunately, programming has evolved into an easier task because of the development of high-level programming languages. A **high-level programming language** allows you to use a vocabulary of reasonable terms such as "read," "write," or "add," instead of the sequences of on and off switches that perform these tasks. High-level languages also allow you to assign intuitive names to areas of computer memory, such as "hoursWorked" or "rateOfPay," rather than having to remember the memory locations (switch numbers) of those values.

Each high-level language has its own **syntax**, or rules of the language. For example, depending on the specific high-level language, you might use the verb "print" or "write" to produce output. All languages have a specific, limited vocabulary and a specific set of rules for using that vocabulary. Programmers use a computer program called a **compiler** to translate their high-level language statements into machine code. The compiler issues an error message each time the programmer uses the programming language incorrectly; subsequently, the programmer can correct the error and attempt another translation by compiling the program again. Other languages, including Java, use an **interpreter** to read the compiled code line by line at run time. A Java interpreter can run as a stand-alone program, or it can be part of a Web browser such as Netscape Navigator or Microsoft Internet Explorer where it can be invoked automatically to run applets in a Web page. The ability to run applets in a Web browser is one feature that distinguishes Java from other programming languages. When you are learning a computer programming language, such as the Java programming language, C++, or Visual Basic, you really are learning the vocabulary and syntax rules for that language.

In addition to learning the correct syntax for a particular language, a programmer also must understand computer programming logic. The **logic** behind any program involves executing the various statements and procedures in the correct order to produce the desired results. For example, you would not write statements to tell the computer program to process data until it had been properly read into the program. Similarly, you might be able to use a computer language's syntax correctly, but be unable to execute a logically constructed, workable program. Examples of logical errors include multiplying two values when you meant to divide them, or producing output prior to obtaining the appropriate input. Tools that will help you visualize and understand logic will be presented in Chapter 5.

Understanding Object-Oriented Programming Concepts

There are two popular approaches to writing computer programs: procedural programming and object-oriented programming.

Procedural programming involves using your knowledge of a programming language to create computer memory locations that can hold values—for example, numbers and text, in electronic form. The computer memory locations are called **variables** because they hold values that might vary. For example, a payroll program written for a company might contain a variable named rateOfPay. The memory location referenced by the name rateOfPay might contain different values (a different value for every employee of the company) at different times. During the execution of the payroll program, each value stored under the name rateOfPay might have many **operations** performed on it—the value might be read from an input device, the value might be multiplied by another variable representing hours worked, and the value might be printed on paper. For convenience, the individual operations used in a computer program are often grouped into logical units called **procedures**. For example, a series of four or five comparisons and calculations that together determine an individual's federal withholding tax value might be grouped as a procedure named calculateFederalWithholding. A procedural program defines the variable memory locations, and then **calls** a series of procedures to input, manipulate, and output the values stored in those locations. A single procedural program often contains hundreds of variables and thousands of procedure calls.

Object-oriented programming is an extension of procedural programming in which you take a slightly different approach to writing computer programs. Writing **object-oriented programs** involves both creating objects and creating applications that use those objects. Objects often interrelate with other objects, and once created, objects can be reused over and over again to develop new programs. Thinking in an object-oriented manner involves envisioning program components as objects that are similar to concrete objects in the real world; then you can manipulate the objects to achieve a desired result.

If you've ever used a computer that uses a command-line operating system (such as DOS), and if you've also used a GUI (graphical user interface, such as Windows), then you are familiar with the difference between procedural and object-oriented programs. If you want to move several files from a floppy disk to a hard disk, you can use either a typed command at a prompt or command line, or you can use a mouse in a graphical environment to accomplish the task. The difference lies in whether you issue a series of commands, in sequence, to move the three files, or you drag icons representing the files from one screen location to another, much as you would physically move paper files from one file cabinet to another in your office. You can move the same three files using either operating system, but the GUI system allows you to manipulate the files like their real-world paper counterparts. In other words, the GUI system allows you to treat files as objects.

Objects both in the real world and in object-oriented programming are made up of states and methods. The states of an object are commonly referred to as its attributes. **Attributes** are the characteristics that define an object as part of a class. For example, some of your automobile's attributes are its make, model, year, and purchase price. Other attributes include whether the automobile is currently running, its gear, its speed, and whether it is dirty. All automobiles possess the same attributes, but not, of course, the same values for those attributes. Similarly, your dog has the attributes of its breed, name, age, and whether his or her shots are current.

A **class** is a term that describes a group or collection of objects with common properties. An **instance** of a class is a technical term for an existing object of a class. Therefore, your red Chevrolet automobile with the dent can be considered an instance of the class that is made up of all automobiles, and your Golden Retriever dog named Goldie is an instance of the class that is made up of all dogs. Thinking of items as instances of a class allows you to apply your general knowledge of the class to individual members of the class. A particular instance of an object takes on, or **inherits**, its attributes from a more general category. If a general class Dog has defined attributes and methods, they can be passed on to a specific class Golden Retriever Dog with minimal programming effort. If your friend purchases an Automobile, you know it has a model name, and if your friend gets a Dog, you know the dog has a breed. You might not know the exact contents of your friend's Automobile, its current state or her Automobile's speed or her Dog's shots, but you do know what attributes exist for the Automobile and Dog classes. Similarly, in a GUI operating environment, you expect each component to have specific, consistent attributes, such as a menu bar and a title bar, because each component inherits these attributes as a member of the general class of GUI components.

Tip

By convention, programmers using the Java programming language begin their class names with an uppercase letter. Thus, the class that defines the attributes and methods of an automobile would probably be named Automobile, and the class for dogs would probably be named Dog. However, following this convention is not required to produce a workable program.

Besides attributes, objects can use methods to accomplish tasks. A **method** is a self-contained block of program code. Automobiles, for example, can move forward and backward. They can also be filled with gasoline or be washed, both of which can be programmed as methods to change some of their attributes. Methods exist for ascertaining certain attributes, such as the current speed of an Automobile and the current status of its gas tank. Similarly, a Dog can walk or run, eat food, and get a bath, and there are methods to determine how hungry the Dog is. GUI operating system components can be maximized, minimized, and dragged. Like procedural programs, object-oriented programs have variables (attributes) and procedures (methods), but the attributes and methods are encapsulated into objects that are then used much like real-world objects. **Encapsulation** refers to the hiding of data and methods within an object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a "black box," or a device that you can use without regard to the internal mechanisms. A programmer gets to access and use the methods and data contained in the black box but cannot change them.

If an object's methods are well written, the user is unaware of the low-level details of how the methods are executed, and the user must simply understand the interface or interaction between the method and the object. For example, if you can fill your Automobile with gasoline, it is because you understand the interface between the gas pump nozzle and the vehicle's gas tank opening. You don't need to understand how the pump works mechanically or where the gas tank is located inside your vehicle. If you can read your speedometer, it does not matter how the displayed figure is calculated. As a matter of fact, if someone produces a superior, more accurate speed-determining device and inserts it in your Automobile, you don't have to know or care how it operates, as long as your interface remains the same. The same principles apply to well-constructed objects used in object-oriented programs.

Learning About The Java Programming Language

The **Java programming language** was developed by Sun Microsystems as an object-oriented language that is used both for general-purpose business programs and for interactive World Wide Web-based Internet programs. Some of the advantages that have made the Java programming language so popular in recent years are its security features, and the fact that it is **architecturally neutral**, which means that you can use the Java programming language to write a program that will run on any platform, or operating system.

Java can be run on a wide variety of computers because Java does not execute instructions on a computer directly. Instead, Java runs on a hypothetical computer known as the **Java virtual machine(JVM)**.

The Java Environment shown in Figure 1-1 is described as follows: the Java programming statements known as **source code** are first constructed using a text editor. Then a special compiler known as the Java compiler converts the source code into a binary program of **byte code**. A program called the **Java Interpreter** checks the byte code and executes the byte code instructions line by line within the Java virtual machine. Since the Java program is isolated from the native operating system, the Java program is insulated from the particular hardware on which it is run. Because of this insulation, the JVM provides security against intruders getting at your computer's hardware through the operating system. In contrast, when using other programming languages, software vendors usually have to produce multiple versions of the same product (a DOS version, Windows version, Macintosh version, Unix version, and so on) so all users can use the program. With the Java programming language, one program version will run on all these platforms.

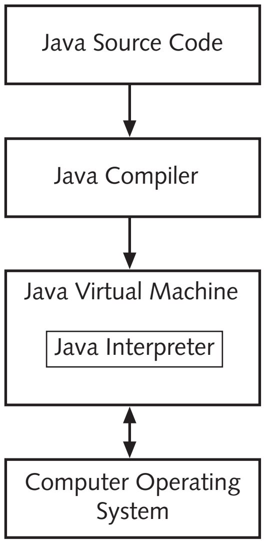


Figure 1-1 Java enviroment

Tip: For simplicity, the terms "Java program" and "program for the Java programming language" are used interchangeably throughout this text.

The Java programming language also is simpler to use than many other object-oriented languages. The Java programming language is modeled after the C++ programming language. Although neither language is "simple" to read or understand on first exposure, the Java programming language does eliminate some of the most difficult-to-understand features in C++, such as pointers and multiple inheritance.

Java Program Types

You can write two kinds of programs using Java. Programs that are embedded in a Web page are called Java **applets**. Stand-alone programs are called Java **applications**. Java applications can be further subdivided into **console** applications, which support character output to a computer screen in a DOS window, for example, and **windowed** applications, which create a graphical user interface (GUI) with elements such as menus, toolbars, and dialog boxes. Console applications are the easiest applications to create; we will start using them in the next section. Windowed applications will be introduced in Chapter 11.

Starting a Java Program

At first glance, even the simplest Java program involves a fair amount of confusing syntax. Consider the following simple program. This program is written on seven lines, and its only task is to print "First Java program" on the screen.

```
public class First
{
  public static void main(String[] args)
  {
   System.out.println("First Java program");
  }
}
```

The statement that does the actual work in this program is `System.out.println("First Java program");`.

Tip

All Java programming language statements end with a semicolon.

The text "First Java program" is a **literal string** of characters; that is, it is a series of characters that will appear exactly as entered. Any literal string in Java appears between double quotation marks, as opposed to single quotation marks as in 'First Java program'. Even though code is written on multiple lines for ease of reading, the literal string cannot be broken because parts of the literal string will appear on different lines.

The string "First Java program" appears within parentheses because the string is an argument to a method, and arguments to methods always appear within parentheses. **Arguments** consist of information that a method requires to perform its task. For example, you might place a catalog order with a company that sells sporting goods. Processing a catalog order is a method that consists of a set of standard procedures. However, each catalog order requires information such as which item number you are ordering and the quantity of the item desired; this information can be considered the order's argument. If you order two of item 5432 from a catalog, you expect different results than if you order 1,000 of item 9008. Likewise, if you pass the argument "Happy Holidays" to a method, you expect different results than if you pass the argument "First Java program".

Within the statement `System.out.println("First Java program");`, the method to which you are passing "First Java program" is named println(). The println() method prints a line of output on the screen, positions the insertion point on the next line, and stands ready for additional output.

Tip

Method names usually are referenced followed by their parentheses, as in println(), so you can distinguish method names from variable names.

Within the statement `System.out.println("First Java program");`, *out* is an object. The out object represents the screen. Several methods, including println(), are available with the out object. Of course, not all objects have a println() method (for instance, you can't print to a keyboard, to your Automobile, or to your Dog), but the creators of the Java platform assumed you frequently would want to display output on a screen. Therefore, the out object was created and endowed with the method named println(). In this chapter, you will create your own objects and endow them with your own methods.

Tip

The print() method is very similar to the println() method. With println(), after the message prints, the insertion point appears on the following line. With print(), the insertion point does not advance to a new line; it remains on the same line as the output.

Within the statement `System.out.println("First Java program");`, *System* is a class. Therefore, System defines the attributes of a collection of similar "System" objects, just as the Dog class defines the attributes of a collection of similar Dog objects. One of the System objects is *out*. (You can probably guess that another object is *in*, and that it represents an input device.)

Tip

The Java programming language is case sensitive; the class named System is a completely different class from one named system, SYSTEM, or even sYsTeM.

The dots (periods) in the statement `System.out.println("First Java program");` are used to separate the names of the class, object, and method. You will use this same class-dot-object-dot-method format repeatedly in your Java programs.

The statement that prints the string "First Java program" is embedded in the program shown in Figure 1-2.

```
public class First
{
   public static void main(String[] args)
      {
       System.out.println("First Java program");
      }
}
```

Figure 1-2 Printing a string

Everything that you use within a Java program must be part of a class. When you write `public class First`, you are defining a class named `First`. You can define a Java class using any name or identifier you need, as long as it meets the following requirements:

- A class name must begin with a letter of the alphabet (which includes any non-English letter, such as α or π), an underscore, or a dollar sign.

- A class name can contain only letters, digits, underscores, or dollar signs.
- A class name cannot be a Java programming language reserved keyword, such as `public` or `class` (see Figure 1-3 for a list of reserved keywords).
- A class name cannot be one of the following values: `true`, `false`, or `null`.

Tip

The Java programming language is based on Unicode, which is an international system of character representation. The term *letter* indicates English-language letters, as well as characters from Arabic, Greek, and other alphabets.

abstract	float	private
Boolean	for	protected
break	future	public
byte	generic	rest
byvalue	goto	return
case	if	short
cast	implements	static
catch	import	super
char	inner	switch
class	instanceof	synchronized
const	int	this
continue	interface	throw
default	long	throws
do	native	transient
double	new	try
else	null	var
extends	operator	void
final	outer	volatile
finally	package	while

Figure 1-3 Java programming language reserved keywords

It is a Java programming language standard to begin class names with an uppercase letter and employ other uppercase letters as needed to improve readability. Table 1-1 lists some valid and conventional class names for the Java programming language.

Table 1-1 Some valid class names in the Java programming language

Class Name	Description
Employee	Begins with an uppercase letter
UnderGradStudent	Begins with an uppercase letter, contains no spaces, and emphasizes each new word with an initial uppercase letter
InventoryItem	Begins with an uppercase letter, contains no spaces, and emphasizes the second word with an initial uppercase letter
Budget2004	Begins with an uppercase letter and contains no spaces

Tip

You should follow established conventions for the Java programming language so your programs will be easy for other programmers to interpret and follow. This book uses established Java programming conventions.

Table 1-2 Some unconventional class names in the Java programming language

Class Name	Description
employee	Begins with a lowercase letter
Undergradstudent	New words are not indicated with initial uppercase letters; difficult to read
Inventory_Item	The underscore is not commonly used to indicate new words
BUDGET2004	Appears as all uppercase letters

Table 1-3 Some illegal class names in the Java programming language

Class Name	Description
an employee	Space character is illegal
Inventory Item	Space character is illegal
class	`class` is a reserved word
2001Budget	Class names cannot begin with a digit
phone#	The # symbol is not allowed

In Figure 1-2, the line `public class First` contains the keyword `class`, which identifies First as a class. The reserved word `public` is an access modifier. An **access modifier** defines the circumstances under which a class can be accessed. Public access is the most liberal type of access; you will learn about public and other types of access in Chapter 3.

You enclose the contents of all classes within curly braces ({ and }). A class can contain any number of data items and methods. In Figure 1-2, the class First contains only one method within its curly braces. The name of the method is main(), and the main() method contains its own set of parentheses and only one statement—the println() statement.

Tip

In general, whitespace is optional in the Java programming language. Whitespace is any combination of spaces, tabs, and carriage returns (blank lines). However, you cannot use whitespace within any identifier or keyword. You can insert whitespace between words or lines in your program code by typing spaces, tabs, or blank lines because the compiler will ignore these extra spaces. You use whitespace to organize your program code and make it easier to read.

For every opening curly brace ({) in a Java program, there must be a corresponding closing curly brace (}). The placement of the opening and closing curly braces is not important to the compiler. For example, the following method is executed exactly the same as the one shown in Figure 1-2. The only difference is that the method is organized differently. This location of the curly braces is common in professional code. Usually, code in which you vertically align each pair of opening and closing curly braces is easier to read and is used throughout this textbook. Strive to type your code so it is easy to read.

```
public static void main(String[] args) {
System.out.println("First Java program");
}
```

The method header for the main() method is quite complex. The meaning and purpose of each of the terms used in the method header will become clearer as you complete this textbook; a brief explanation will suffice for now.

In the method header `public static void main(String[] args)`, the word `public` is an access modifier, just as it is when you define the First class. In the English language, the word *static* means showing little change, or stationary. In the Java programming language, the reserved keyword **`static`** ensures that main() is accessible even though no objects of the class exist. It also indicates that every member created for the First class will have an identical, unchanging main() method. Within the Java programming language, `static` also implies uniqueness. Only one main() method for the First class will ever be stored in the memory of the computer. Of course, other classes eventually might have their own, different main() methods.

In English, the word *void* means empty. When the keyword **`void`** is used in the main() method header, it does not indicate that the main() method is empty, but rather, that the main() method does not return any value when it is called. This doesn't mean that main() doesn't produce output—in fact, the method does. The main() method does not send any value back to any other method that might use it. You will learn more about return values in Chapter 3.

Not all classes have a main() method; in fact many do not. All Java applications however, must include a method named main(), and most Java applications have additional methods. When you execute a Java application, the compiler always executes the main() method first.

In the method header `public static void main(String[] args)`, you already might recognize that the contents between the parentheses, `(String[] args)`, must represent an argument passed to the main() method, just as the string "First Java program" is an argument passed to the println() method. `String` represents a Java class that can be used to represent character strings. The identifier `args` is used to hold any Strings that might be sent to the main() method. The main() method could do something with those arguments, such as print them, but in Figure 1-2 the main() method does not actually use the `args` identifier. Nevertheless, you must place an identifier within the main() method's parentheses. The identifier does not need to be named `args`—it could be any legal Java identifier—but the name `args` is traditional.

Tip

When you refer to the String class in the main() method header, the square brackets indicate an array of String objects. You will learn more about arrays and the String class in Chapter 6.

The simple program shown in Figure 1-2 has many pieces to remember. However, for now, you can use the program shown in Figure 1-4 as a shell, where you replace the line /******/ with any statements that you want to execute.

```
public class First
{
   public static void main(String[] args)
   {
     /******/
   }
}
```

Figure 1-4 Shell output program

Now that you understand the basic framework of a program written in the Java programming language, you are ready to enter your first Java program into a text editor. It is a tradition among programmers that the first program you write in any language produces "Hello, world!" as its output. You will create such a program now. You can use any text editor, such as Notepad, Textpad, or any other text-processing program.

To write your first Java program:

1. Start any text editor (such as Notepad or Textpad, but be sure the file is saved with the extension .txt), and then open a new document, if necessary. (Textpad is the easiest program to use to write your programs.)
2. Type the class header **`public class Hello`**. In this example, the class name is `Hello`. You can use any valid name you want for the class. If you choose Hello, you must refer to the class as Hello, and not as hello, because the Java programming language is case sensitive.
3. Press **[Enter]** once, type **{**, press **[Enter]** again, and then type **}**. You will add the main() method between the curly braces. Although it is not required, it is a good practice to place each curly brace on its own line. This makes your code easier to read.
4. As shown in Figure 1-5, add the main() method header between the curly braces and then type a set of curly braces for main().

```
public class Hello
{
  public static void main(String[] args)
  {
  }
}
```

Figure 1-5 The main() method shell for the Hello class

Next add the statement within the main() method's brackets that will produce the output, "Hello, world!".

5. Use Figure 1-6 as a guide for adding a println() statement to the main() method.

```
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello, world");
  }
}
```

Figure 1-6 main() method for the Hello class

6. Save the program as **Hello.java** in the Chapter.01 folder on your Student Disk. It is important that the file extension is .java. If it is not, the compiler for the Java programming language will not recognize the program.

Help

Many text editors attach their own filename extension (such as .txt or .doc) to a saved file. Double-check your saved file to ensure that it does not have a double extension (such as Hello.java.txt). If the file has a double extension, rename the file. If you explicitly type quotation marks surrounding a filename (such as "Hello.java"), most text editors will save the file as you specify, without adding an extension. Make sure that you save your .java files as text documents. The default for Notepad is to save all documents as text.

Adding Comments to a Java Program

As you can see, even the simplest Java program takes several lines of code, and contains somewhat perplexing syntax. Large programs that perform many tasks include much more code, and as you write longer programs, it becomes increasingly difficult to remember why you included steps, or how you intended to use particular variables. **Program comments** are nonexecuting statements that you add to a program for the purpose of documentation. Programmers use comments to leave notes for themselves and for others who might read their programs in the future. At the very least, your programs should include comments indicating the program's author, the date, and the program's name or function. The best practice dictates that a brief comment be made to tell the purpose for each class and its methods.

Tip

It is suggested that as you work through this book you add comments as the first three lines of every program. The comments should contain the program name, your name, and the date. Your instructor might ask you to include additional comments.

Comments also can serve a useful purpose when you are developing a program. If a program is not performing as expected, you can comment out various statements and subsequently run the program to observe the effect. When you **comment out** a statement, you turn it into a comment so the compiler will not execute its command. This helps you pinpoint the location of errant statements in malfunctioning programs.

There are three types of comments in the Java programming language:

- **Line comments** start with two forward slashes (//) and continue to the end of the current line. Line comments can appear on a line by themselves or at the end of a line following executable code.
- **Block comments** start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/). Block comments can appear on a line by themselves, on a line before executable code, or after executable code. Block comments also can extend across as many lines as needed.
- A special case of block comments are **javadoc** comments. They begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/). You can use javadoc comments to generate documentation with a program named javadoc.

Tip

The forward slash (/) and the backslash (\) characters often are confused, but they are two distinct characters. You cannot use them interchangeably.

Tip

The Java Development Kit (JDK) includes the javadoc tool, which contains classes that you can use when writing programs in the Java programming language.

Figure 1-7 shows how comments are used in code.

```
//Demonstrating comments
/* This shows
   that these comments
        don't matter */
System.out.println("Hello");//This line executes
     // up to where the comment started
/**Everything but the println() line
 is a comment. */
```

Figure 1-7 Using comments in a program

Next you will add comments to your Hello.java program.

To add comments to your program:

1. Position the insertion point at the top of the file, press **[Enter]** to insert a new line, press the **Up arrow** key to go to that line, and then type the following comments at the top of the file. Press **[Enter]** after typing each line. Insert your name and today's date where indicated.

```
// Filename Hello2.java
// Written by <your name>
// Written on <today's date>
```

2. Scroll to the end of the line that reads `public class Hello`, change the class name to **`Hello2`**, press **[Enter]**, and then type the following block comment in the program:

```
/*  This program demonstrates the use of the println()
method to print the message Hello, world!  */
```

3. Save the file as **Hello2.java** in the Chapter.01 folder on your Student Disk.

Running a Java Program

After you write and save your program, there are two steps that must occur before you can view the program output.

1. You must compile the program you wrote (called the source code) into bytecode.
2. You must use the Java interpreter to translate the bytecode into executable statements.

To compile your source code from the command line, you type `javac` followed by the filename of the file that contains the source code. For example, to compile a file named First.java, you would type `javac First.java` and then press [Enter]. There will be one of three outcomes:

- You receive a message such as `Bad command or filename`.
- You receive one or more program language error messages.
- You receive no messages, which means that the program compiled successfully.

Tip: When compiling, if the source code file is not in the current path, you can type a full path with the filename, for example, `javac c:\java\myprograms\First.java`.

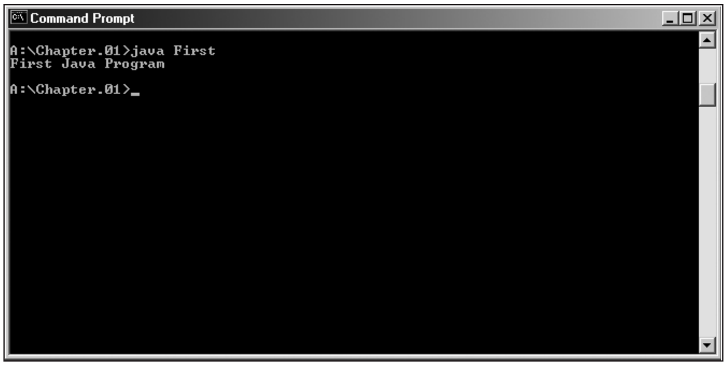
If you receive a message such as `Bad command or filename`, it might mean one of the following:

- You misspelled the command `javac`.
- You misspelled the filename.
- You are not within the correct subfolder or subdirectory on your command line.
- The Java programming language was not installed properly. (See Appendix for information on installation.)

If you receive a programming language error message, then there are one or more syntax errors in the source code. A **syntax error** is a programming error that occurs when you introduce typing errors into your program. For example, if your class name is "first" (with a lowercase f) in the source code, but you saved the file as First.java, you will get an error message, such as `public class first should not be defined in First.java`, after compiling the program because "first" and "First" are not the same in a case-sensitive language. If this error occurs, you must reopen the text file that contains the source code and make the necessary corrections.

If you receive no error messages after compiling the code in a file named First.java, then the program compiled successfully, and a file named First.class was created and saved in the same folder as the program text file. After a successful compile, you can run the class file on any computer that has a Java language interpreter.

To run the program from the command line, you type `java First`. Figure 1-8 shows the program's output. Next you will compile and interpret your Hello2.java program.



```
A:\Chapter.01>java First
First Java Program

A:\Chapter.01>_
```

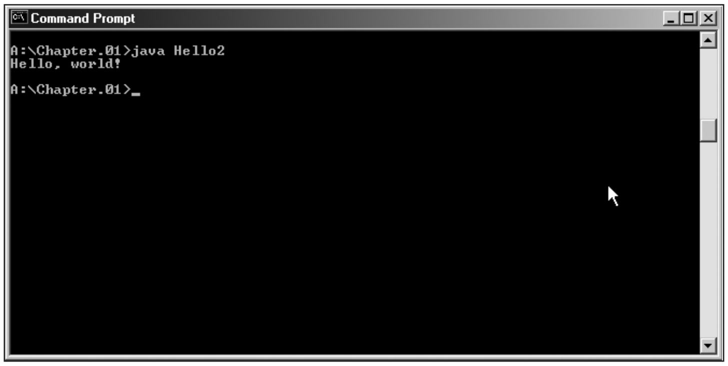
Figure 1-8 Output of the First program

To compile and interpret your Hello2.java program with comments:

1. Go to the command-line prompt for the drive and folder or subdirectory in which you saved Hello2.java.
2. At the command line, type **`javac Hello2.java`**.

If you receive an error message, look in the section "Running a Java Program" to find its cause, and then make the necessary corrections. Save the file again, and then repeat Steps 1 and 2 until your program compiles successfully.

3. When the compile is successful, execute your program by typing **`java Hello2`** at the command line, and then press **[Enter]**. The output should appear on the next line, as shown in Figure 1-9.



```
A:\Chapter.01>java Hello2
Hello, world!

A:\Chapter.01>_
```

Figure 1-9 Output of the Hello2 program

When you run a Java program using the `java` command, do not add the .class extension to the filename. If you type `java First`, the interpreter will look for a file named First.class. If you type `java First.class`, the interpreter will incorrectly look for a file named First.class.class.

Modifying a Java Program

After viewing the program output, you might decide to modify the program to get a different result. For example, you might decide to change the First program's output from `First Java program` to the following:

```
My new and improved
Java program
```

To produce the new output, first you must modify the text file that contains the existing program. You may also want to change the class name to First2 to indicate that the First program has changed. You will need to change the literal string that currently prints, and then add an additional text string. Figure 1-10 shows the program to change the output.

```
public class First2
{
   public static void main(String[] args)
   {
    System.out.println("My new and improved");
    System.out.println("Java program");
   }
}
```

Figure 1-10 Changing a program's output

The three changes are the class name change of First to First2, the addition of the statement `System.out.println("My new and improved");`, and the removal of the word "First" from the string in the statement `System.out.println("Java' program");`. However, if you type `java First2` at the command line right now, you will not see the new output—you will see the old output. Before the new source code will execute, you must do the following:

1. Save the file with the changes using the same filename (First2.java).
2. Compile the First2 class with the `javac` command.
3. Interpret the First2.class bytecode with the `java` command.

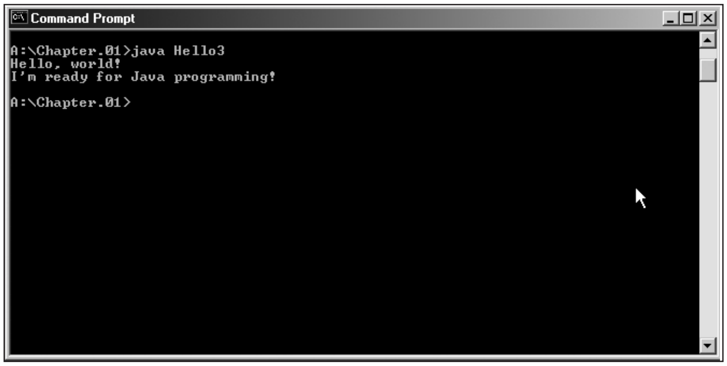
Next you will change your Hello2 class and rerun your program.

To change the Hello2 class and rerun the program:

1. Open the file **Hello2.java** in your text editor. Change both the comment name and the class name to **Hello3**.
2. Add the following statement below the statement that prints "Hello, world!": **`System.out.println("I'm ready for Java programming!");`**. Make sure to type the semicolon at the end of the statement and use the correct case.
3. Save the file as **Hello3.java** in the Chapter.01 folder on your Student Disk. Changing the Hello2 class name to Hello3 and saving the file as Hello3 calls attention to the change being made to the Hello2 class.
4. At the command line, compile the file by typing the command **`javac Hello3.java`**.

? Help If you receive compile errors, return to the Hello3.java file in the text editor, fix the errors, and then repeat Steps 3 and 4 until the program compiles successfully.

5. Interpret and execute the class by typing the command **`java Hello3`**, and then press **[Enter]**. Your output should look like Figure 1-11.



```
A:\Chapter.01>java Hello3
Hello, world!
I'm ready for Java programming!

A:\Chapter.01>
```

Figure 1-11 Output of the revised Hello program

Errors and Debugging

When typing errors are made as code is entered and compiled, the compiler will produce an error message as explained in the examples earlier in the chapter. The exact ror message depends on the compiler. In the First program of Figure 1-2, typing the `System.out.println()` code as `system.out.println("First Java Program");` produces an error message similar to "cannot resolve symbol…". This is a compile-time error commonly referred to as a syntax error. The compiler detects the violation of language rules and refuses to translate the program to machine code. The compiler will try to report as many errors as it can find during compilation so that you can fix as many errors as possible. Sometimes one error in syntax causes subsequent errors that normally would not be errors if the first syntax error did not exist. You should correct the errors that make sense to you and then recompile.

A second kind of error occurs when the syntax of the program is correct and the program is compiled but produces incorrect results. This is a run-time error or logic error. In the First program of Figure 1-1, typing the `System.out.println()` code as `System.out.println("Frst Java Program";` does not produce an error. The compiler does not find the spelling error of "Frst" instead of "First". Errors of this type must be detected by carefully examining the program output. It is the responsibility of the program author to test programs and find any logic errors. Good programming practice stresses programming structure and development that helps minimize errors. Additionally, each chapter in this book has four programs for debugging that will give you practice finding and correcting syntax and logic errors.

FAQs, JDKs, Documentation, and Tutorials

A great wealth of material exists at the Sun Microsystems Web site, *http://java.sun.com*. Of particular value are the FAQs (Frequently Asked Questions) that you can link to from a site search of the Web site. By searching the site for FAQs, you can locate the collections of frequently asked questions that provide brief answers to many common questions about Java software and products. A site search for the Java Development Kit (JDK) will help you locate and download the latest JDK for free. At the time of this writing the latest version is The Java™ 2 Platform, Standard Edition v1.4.0, available in Windows, Linux, and Solaris operating systems. You can search and browse documentation online or you can download the documentation file for the SDK and install it on your computer. Once installed, you can search and browse documentation locally.

A downloadable Java tutorial titled "The Java Tutorial: A practical guide for programmers", with hundreds of complete working examples is available from *http://java.sun.com/docs/books/tutorial/*. The tutorial is organized into trails—groups of lessons on a particular subject. You can start the tutorial at the beginning and navigate sequentially from beginning to end, or jump from one trail to another. As you study each textbook chapter you are encouraged to make good use of these support materials.

Chapter Summary

- Objects are made up of states and methods. The states of an object also are known as its attributes. An individual object is an instance of a class; the object inherits its attributes from the class. The user of an object does not need to understand the details of any method, but must understand the interface with the object.
- A program written in Java is run on a standardized hypothetical computer called the Java virtual machine (JVM) that exists virtually inside your machine by a program. When your program is compiled into bytecode, an interpreter within the JVM subsequently interprets the bytecode and interfaces with the operating system to produce the program results.
- All Java programming language statements end with a semicolon. Periods (called dots) are used to separate classes, objects, and methods in program code. The contents of all classes are contained within opening and closing curly braces. A series of characters that appears between double quotation marks is a literal string. Java programming language methods might require arguments or messages to perform the appropriate task.
- Everything that you use within a Java program must be part of a class. A Java programming language class might take any name or identifier that begins with either an uppercase or lowercase letter of the alphabet, and contains only uppercase and lowercase letters, digits, and underscores. A class name cannot be a reserved keyword of the Java programming language.
- The reserved word `public` is an access modifier that defines the circumstances under which a class can be accessed. The keyword `static` in a method header indicates that every member of a class will have an identical, unchanging method. The keyword `void` in a method header indicates that the method does not return any value when it is called.

- All Java application programs must have a method named main(). Most Java applications have additional methods.
- Program comments are nonexecuting statements that you add to a program for the purpose of documentation. There are three types of comments in the Java programming language: line comments begin with two forward slashes (//); block comments begin with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/); and javadoc comments begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/).
- To compile your source code from the command line, type `javac` followed by the name of the file that contains the source code. If the file resides in a different path from the command prompt, use the full path and filename. When you compile your source code, the compiler creates a file with a .class extension. You can run the .class file on any computer that has a Java language interpreter by entering the `java` command followed by the name of the class file. Do not type the .class extension with the filename.
- When you modify a program, before it will execute correctly, you must do the following: save the file with the changes using the same filename, compile the class with the `javac` command, and interpret the class bytecode with the `java` command.
- To avoid and minimize syntax and logic errors you must enter code carefully and closely examine your program's output.

REVIEW QUESTIONS

1. The most basic circuitry-level computer language, which consists of on and off switches, is ________________.
   a. a high-level language
   b. machine language
   c. the Java programming language
   d. C++
2. Languages that let you use a vocabulary of descriptive terms such as "read," "write," or "add" are known as ________________ languages.
   a. high-level
   b. machine
   c. procedural
   d. object-oriented

3. The rules of a programming language constitute its ________________.
   a. objects
   b. logic
   c. format
   d. syntax
4. A ________________ translates high-level language statements into machine code.
   a. programmer
   b. syntax detector
   c. compiler
   d. decipherer
5. Computer memory locations are called ________________.
   a. compilers
   b. variables
   c. addresses
   d. appellations
6. For convenience, the individual operations used in a computer program are often grouped into logical units called ________________.
   a. procedures
   b. variables
   c. constants
   d. logistics
7. Envisioning program components as objects that are similar to concrete objects in the real world is the hallmark of ________________.
   a. command-line operating systems
   b. procedural programming
   c. object-oriented programming
   d. machine languages
8. An object's attributes also are known as its ________________.
   a. states
   b. orientations
   c. methods
   d. procedures
9. An instance of a(n) ________________ inherits its attributes from it.
   a. object
   b. procedure

   c. method
   d. class

10. The Java programming language is architecturally ________________.
    a. specific
    b. oriented
    c. neutral
    d. abstract

11. You must compile programs written in the Java programming language into ________________.
    a. bytecode
    b. source code
    c. javadoc statements
    d. object code

12. All Java programming language statements must end with a ________________.
    a. period
    b. comma
    c. semicolon
    d. closing parenthesis

13. Arguments to methods always appear within ________________.
    a. parentheses
    b. double quotation marks
    c. single quotation marks
    d. curly braces

14. In a Java program, you must use ________________ to separate classes, objects, and methods.
    a. commas
    b. semicolons
    c. periods
    d. forward slashes

15. All Java programs must have a method named ________________.
    a. method()
    b. main()
    c. java()
    d. Hello()

16. Nonexecuting program statements that provide documentation are called ________________.
    a. classes
    b. notes
    c. comments
    d. commands
17. The Java programming language supports three types of comments: ________________, ________________, and javadoc.
    a. line, block
    b. string, literal
    c. constant, variable
    d. single, multiple
18. After you write and save a program file, you ________________ it.
    a. interpret and then compile
    b. interpret and then execute
    c. compile and then resave
    d. compile and then interpret
19. The command to execute a compiled program is ________________.
    a. `run`
    b. `execute`
    c. `javac`
    d. `java`
20. You save text files containing Java language source code using the file extension ________________.
    a. .java
    b. .class
    c. .txt
    d. .src
21. The Java Virtual machine, or JVM, refers to a(n) ________________.
    a. interpreter
    b. operating system
    c. hypothetical computer
    d. compiler

EXERCISES

1. For each of the following Java programming language identifiers, note whether they are legal or illegal:
   a. weeklySales
   b. last character
   c. class
   d. MathClass
   e. myfirstinitial
   f. phone#
   g. abcdefghijklmnop
   h. 23jordan
   i. my_code
   j. 90210
   k. year2000problem
   l. abffraternity
2. Name some attributes that might be appropriate for each of the following classes:
   a. TelevisionSet
   b. EmployeePaycheck
   c. PatientMedicalRecord
3. Write, compile, and test a program that prints your first name on the screen. Save the program as **Name.java** in the Chapter.01 folder on your Student Disk.
4. Write, compile, and test a program that prints your full name, street address, city, state, and zip code on three separate lines on the screen. Save the program as **Address.java** in the Chapter.01 folder on your Student Disk.
5. Write, compile, and test a program that displays the following pattern on the screen:

```
   X
  XXX
 XXXXX
XXXXXXX
   X
```

   Save the program as **Tree.java** in the Chapter.01 folder on your Student Disk.

6. Write, compile, and test a program that prints your initials on the screen. Compose each initial with five lines of initials, as in the following example:

```
    J  FFFFFF
    J  F
    J  FFFF
J   J  F
JJJJJ  F
```

   Save the program as **Initial.java** in the Chapter.01 folder on your Student Disk.

7. Write, compile, and test a program that prints all the objectives listed at the beginning of this chapter. Save the program as **Objectives.java** in the Chapter.01 folder on your Student Disk.

8. Write, compile, and test a program that displays the following pattern on the screen:

```
  *
 * *
* * *
 * *
  *
```

   Save the program as **Diamond.java** in the Chapter.01 folder on your Student Disk.

9. Write, compile, and test a program that displays the following statement about comments:

   "Program comments are nonexecuting statements you add to a program for the purpose of documentation."

   Also include the same statement in three different comments in the program; each comment should use one of the three different methods of including comments in a Java program. Save the program as **Comments.java** in the Chapter.01 folder on your Student Disk.

10. Each of the following files in the Chapter.01 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugOne1.java will become FixDebugOne1.java.

    a. DebugOne1.java

    b. DebugOne2.java

    c. DebugOne3.java

    d. DebugOne4.java

Case Project

Business Cards Limited is a company that designs and prints personal business cards. They have asked you to write a simple program using Java to provide personal business card information for a typical order. The format to be displayed is shown in the table below:

Table 1-4 CardLayout format

Information Layout
First Name Last Name
Address 1
Address 2
City State Zip
Home Phone
Work Phone

Write, compile, and test a Java program to print the table layout using your own personal information. Be sure to use good documentation principles for your program. Save the program as **CardLayout** in the Chapter.01 folder on your Student Disk.

CHAPTER

2

USING DATA WITHIN A PROGRAM

In this chapter, you will:

- ♦ Use constants and variables
- ♦ Learn about the int data type
- ♦ Write arithmetic statements
- ♦ Use the Boolean data type
- ♦ Learn about floating-point data types
- ♦ Understand numeric type conversion
- ♦ Work with the char data type
- ♦ Learn about ASCII and Unicode

How are you doing with your first programs?" asked Lynn Greenbrier during a coffee break. "OK, I think," I replied with just a bit of doubt in my voice. "I sure wish I could do some calculations, though," I continued. "Writing code that only prints the output I coded using println() statements isn't exactly what I had in mind when I considered a job in programming." "Well then," Lynn replied, "let's start learning how Java uses different data types to perform arithmetic and other kinds of calculations."

Using Constants and Variables

You can categorize data as variable or constant. Data is **constant** when it cannot be changed after a program is compiled; data is **variable** when it might change. For example, if you include the statement `System.out.println(459);` in a Java program, the number 459 is a constant. Every time the program containing the constant 459 is executed, the value 459 will print. You can refer to the number 459 as a **literal constant** because its value is taken literally at each use.

Tip

Besides using literal constants, you can use symbolic constants, which you will learn about in Chapter 4.

On the other hand, you can set your data up as a variable. For example, if you create a variable named `ovenTemperature`, and include the statement `System.out.println(ovenTemperature);` within a Java program, then different values might display when the program is executed multiple times, depending on what value is stored in `ovenTemperature` each time the program is run.

Variables are named memory locations that your program can use to store values. The Java programming language provides for eight primitive types of data:

- `boolean`
- `float`
- `byte`
- `int`
- `char`
- `long`
- `double`
- `short`

The eight primitive data types are called **primitive types** because they are simple and uncomplicated. Primitive types also serve as the building blocks for more complex data types, called reference types. The objects you begin creating in Chapter 3 are examples of reference types.

Declaring Variables

You name variables using the same naming rules for legal class identifiers described in Chapter 1. Basically, variable names must start with a letter and cannot be any reserved keyword. You must declare all variables you want to use in a program. A **variable declaration** includes the following:

- A data type that identifies the type of data that the variable will store
- An identifier that is the variable's name
- An optional assigned value, when you want a variable to contain an initial value
- An ending semicolon

Tip

Variable names usually begin with lowercase letters to distinguish them from class names. However, variable names can begin with either an uppercase or a lowercase letter.

For example, the variable declaration `int myAge = 25;` declares a variable of type int named myAge and assigns it an initial value of 25. This is a complete statement that ends in a semicolon. The equals sign (=) is the **assignment operator**. Any value to the right of the equals sign is assigned to the variable on the left of the equals sign. An assignment made when you declare a variable is an **initialization**; an assignment made later is simply an **assignment**. Thus, `int myAge = 25;` initializes myAge to 25, and a subsequent statement `myAge = 42;` might assign a new value to the variable. You should note that the expression `25 = myAge` is illegal.

The variable declaration `int myAge;` also declares a variable of type int named myAge, but no value is assigned at the time of creation.

You can declare multiple variables of the same type in separate statements on different lines. For example, the following statements declare two variables—the first variable is named myAge and its value is 25. The second variable is named yourAge and its value is 19.

```
int myAge = 25;
int yourAge = 19;
```

You also can declare two variables of the same type in a single statement by separating the variable declarations with a comma, as shown in the following statement:

```
int myAge = 25, yourAge = 19;
```

However, if you want to declare variables of different types, you must use a separate statement for each type. The following statements declare two variables of type int (myAge and yourAge) and two variables of type double (mySalary and yourSalary):

```
int myAge, yourAge;
double mySalary, yourSalary;
```

LEARNING ABOUT THE INT DATA TYPE

In the Java programming language, you use variables of type int to store (or hold) **integers**, or whole numbers. An integer can hold any whole number value from –2,147,483,648 to 2,147,483,647. When you assign a value to an int variable, you do not type any commas; you type only digits and an optional plus or minus sign to indicate a positive or negative integer.

Tip

The legal integer values are -2^{31} through $2^{31}-1$. These are the highest and lowest values that you can store in four bytes of memory, which is the size of an int variable.

The types **byte**, **short**, and **long** are all variations of the integer type. You use byte or short if you know a variable will need to hold only small values so you can save space in memory. You use a long if you know you will be working with very large values. Table 2-1 shows the upper and lower value limits for each of these types. It is important to choose appropriate types for the variables you will use in a program. If you attempt to assign a value that is too large for the data type of the variable, the compiler will issue an error message and the program will not execute. If you choose a data type that is larger than you need, you waste memory. For example, a personnel program might use a byte variable for number of dependents (because a limit of 127 is more than enough), a short for hours worked in a month (because 127 isn't enough), and an integer for an annual salary (because even though a limit of 32,000 might be large enough for your salary, it isn't enough for the CEO).

Tip

If your program uses a literal constant integer, such as 932, the integer is an int by default. If you need to use a constant higher than 2,147,483,647, you must follow the number with the letter L to indicate long. For example, `long mosquitosInTheNorthWoods = 2444555888L;` stores a number that is greater than the maximum limit for the int type. You can type either an uppercase or lowercase L to indicate the long type, but the uppercase L is preferred to avoid confusion with the number one.

Table 2-1 Limits on integer values by type

Type	Minimum Value	Maximum Value	Size in Bytes
byte	–128	127	1
short	–32,768	32,767	2
int	–2,147,483,648	2,147,483,647	4
long	–9,223,372,036,854,775,808	9,223,372,036,854,775,807	8

Next you will write a program to declare and display numeric values.

To declare and display values in a program:

1. Open a new document in your text editor.
2. Create a class header and an opening and closing curly brace for a new class named **DemoVariables** by typing the following:

```
public class DemoVariables
{
}
```

3. Position the insertion point after the opening curly brace, press **[Enter]**, press **[Spacebar]** several times to indent the line, and then type the following main() method and its curly braces:

```
public static void main(String[] args)
{
}
```

4. Position the insertion point after the opening curly brace in the main() method, press **[Enter]**, press **[Spacebar]** several times to indent the line, and then type `int oneInt = 315;` to declare a variable of type int named oneInt with a value of 315.

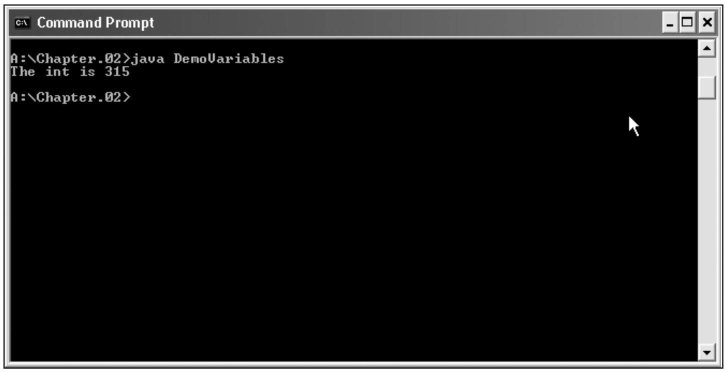
Tip You can declare variables at any point within a method prior to their first use. However, it is common practice to declare variables first, and place method calls second.

5. Press **[Enter]** at the end of the oneInt declaration statement, indent the line, and then type the following two output statements. The first statement uses the print() method to output "The int is " and leaves the insertion point on the same output line. The second statement uses the println() method to output the value of oneInt and advances the insertion point to a new line.

```
System.out.print("The int is ");
System.out.println(oneInt);
```

Tip When your output contains a literal string such as "The int is ", you should type a space before the closing quotation mark so there is a space between the end of the literal string and the value that prints.

6. Save the file as **DemoVariables.java** in the Chapter.02 folder on your Student Disk.
7. Compile the file from the command line by typing `javac DemoVariables.java`. If necessary, correct any errors, save the file, and then compile again.
8. Execute the program from the command line by typing `java DemoVariables`. The output is shown in Figure 2-1.



```
A:\Chapter.02>java DemoVariables
The int is 315

A:\Chapter.02>
```

Figure 2-1 Output of the DemoVariables program

Even though you intend to add additional statements to the DemoVariables program, you compiled and executed the program at this point to make sure that it is working exactly as intended. Sometimes it is a good idea to write and compile your programs in steps, so you can identify any syntax or logical errors as you go, instead of waiting until you finish writing the entire program. Next you will declare two more variables in your program.

To declare two more variables in the program:

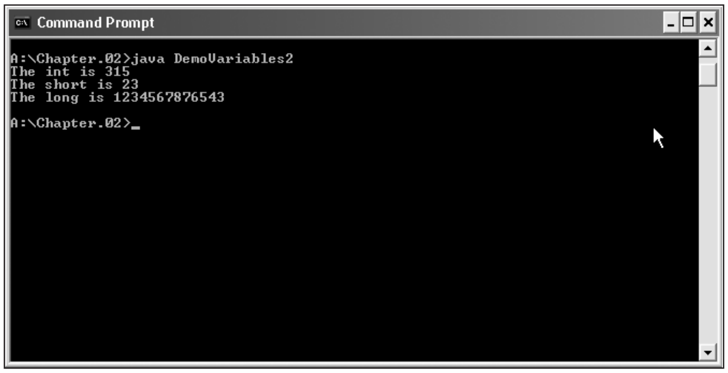
1. Return to the **DemoVariables.java** file in the text editor. Rename the class **DemoVariables2**.
2. Position the insertion point at the end of the line that contains the oneInt declaration, press **[Enter]**, and then type the following variable declarations on separate lines:

   ```
   short oneShort = 23;
   long oneLong = 1234567876543L;
   ```

3. Position the insertion point at the end of the line that contains the println() method that displays the oneInt value, press **[Enter]**, and then type the following statements to display the values of the two new variables:

   ```
   System.out.print("The short is ");
   System.out.println(oneShort);
   System.out.print("The long is ");
   System.out.println(oneLong);
   ```

4. Save the program using the filename **DemoVariables2.java**.
5. Compile the program by typing **`javac DemoVariables2.java`**. If necessary, correct any errors, save the file, and then compile again.
6. Execute the program by typing **`java DemoVariables2`**. The output is shown in Figure 2-2.



```
A:\Chapter.02>java DemoVariables2
The int is 315
The short is 23
The long is 1234567876543

A:\Chapter.02>_
```

Figure 2-2 Output of the DemoVariables2 program

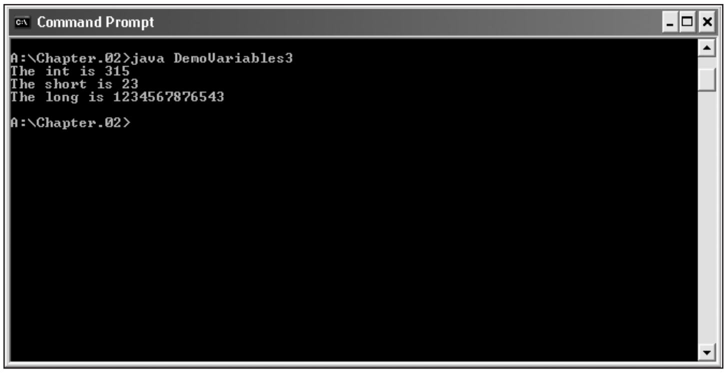
In the previous program, you used two print methods to print a compound phrase with the following code:

```
System.out.print("The long is ");
System.out.println(oneLong);
```

To reduce the amount of typing, you can use one method and combine the arguments with a plus sign using the following statement: `System.out.println("The long is " + oneLong);`. It doesn't matter which format you use—the result is the same, as you will see next.

To change the two print methods into a single statement:

1. Open the **DemoVariables2.java** text file, and rename the class **DemoVariables3**.
2. Use the mouse to select the two statements that print "The int is " and the value of oneInt, and then press **[Delete]** to delete them. In place of the deleted statements, type the following println() statement: **`System.out.println("The int is " + oneInt);`**.
3. Select the two statements that produce output for the short variable, press **[Delete]** to delete them, and then type the statement **`System.out.println("The short is " + oneShort);`**.
4. Finally, select the two statements that produce output for the long variables, delete them, and replace them with **`System.out.println("The long is " + oneLong);`**.
5. Save, compile, and test the program. The output is shown in Figure 2-3.



```
A:\Chapter.02>java DemoVariables3
The int is 315
The short is 23
The long is 1234567876543

A:\Chapter.02>
```

Figure 2-3 Output of the DemoVariables3 program

Writing Arithmetic Statements

Table 2-2 describes the five standard arithmetic operators for integers. You use the arithmetic operators to manipulate values in your programs.

Tip: You will learn about the shortcut arithmetic operators for the Java programming language in Chapter 5.

Table 2-2 Integer arithmetic operators

Operator	Description	Example
+	Addition	45 + 2, the result is 47
–	Subtraction	45 – 2, the result is 43
*	Multiplication	45 * 2, the result is 90
/	Division	45 / 2, the result is 22 (not 22.5)
%	Modulus (remainder)	45 % 2, the result is 1 (that is, 45 / 2 = 22 with a remainder of 1)

Tip: You do not need to perform a division operation before you can perform a modulus operation. A modulus operation can stand alone.

The operators / and % deserve special consideration. When you divide two integers, whether they are integer constants or integer variables, the result is an integer. In other words, any fractional part of the result is lost. For example, the result of 45 / 2 is 22, even though the result is 22.5 in a mathematical expression. When you use the modulus operator with two integers, the result is an integer with the value of the remainder after division takes place—the result of 45 % 2 is 1 because 2 "goes into" 45 twenty-two times with a remainder of 1.

Next you will add some arithmetic statements to the DemoVariables3.java program.

To use arithmetic statements in a program:

1. Open the **DemoVariables3.java** file in your text editor, and change the class to **DemoVariables4**.
2. Position the insertion point at the end of the last line of the current variable declarations, press **[Enter]**, and then type the following declarations:

```
int value1 = 43, value2 = 10, sum, difference,
  product, quotient, modulus;
```

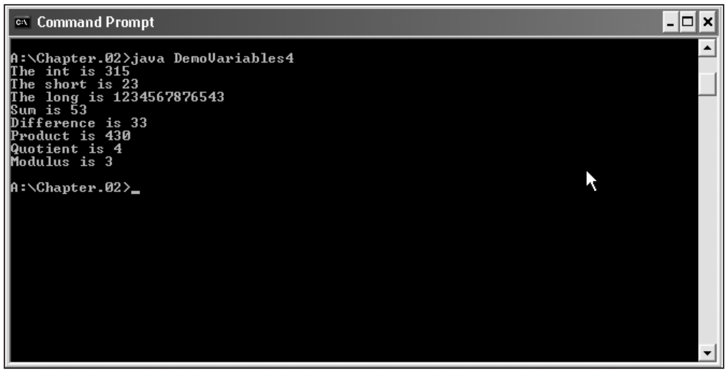
3. Position the insertion point after the statement that prints the oneLong variable, press **[Enter]**, and then type the following statements on separate lines:

```
sum = value1 + value2;
difference = value1 - value2;
product = value1 * value2;
quotient = value1 / value2;
modulus = value1 % value2;
```

4. Press **[Enter]**, and then type the following output statements:

```
System.out.println("Sum is " + sum);
System.out.println("Difference is " + difference);
System.out.println("Product is " + product);
System.out.println("Quotient is " + quotient);
System.out.println("Modulus is " + modulus);
```

5. Save the program as **DemoVariables4.java**.
6. Compile and run the program. Your output should look like Figure 2-4. Analyze the output and confirm that the arithmetic is correct.



```
A:\Chapter.02>java DemoVariables4
The int is 315
The short is 23
The long is 1234567876543
Sum is 53
Difference is 33
Product is 430
Quotient is 4
Modulus is 3

A:\Chapter.02>_
```

Figure 2-4 Output of the DemoVariables4 program

When you combine mathematical operations in a single statement, you must understand operator **precedence**, or the order in which parts of a mathematical expression are evaluated. Multiplication, division, and modulus always take place prior to addition or subtraction in an expression. For example, the expression `int result = 2 + 3 * 4;` results in 14, because the multiplication (3 * 4) occurs before adding 2. You can override normal operator precedence by putting the operation to perform first in parentheses. The statement `int result = (2 + 3) * 4;` results in 20, because the addition within the parentheses takes place first, and then that result (5) is multiplied by 4.

Tip

You will learn more about operator precedence in Chapter 5.

USING THE BOOLEAN DATA TYPE

Boolean logic is based on true-or-false comparisons. Whereas an int variable can hold millions of different values (at different times), a **Boolean variable** can hold only one of two values—`true` or `false`. The following statements declare and assign appropriate values to Boolean variables:

```
boolean isItPayday = false;
boolean areYouBroke = true;
```

You also can assign values based on the result of comparisons to Boolean variables. The Java programming language supports six comparison operators. A **comparison operator** compares two items; an expression containing a comparison operator has a Boolean value. Table 2-3 describes the comparison operators.

Tip

You will learn about other Boolean operators in Chapter 5.

Table 2-3 Comparison operators

Operator	Description	`true` Example	`false` Example
<	Less than	3 < 8	8 < 3
>	Greater than	4 > 2	2 > 4
==	Equal to	7 == 7	3 == 9
<=	Less than or equal to	5 <= 5	8 <= 6
>=	Greater than or equal to	7 >= 3	1 >= 2
!=	Not equal to	5 != 6	3 != 3

When you use any of the operators that have two symbols (==, <=, >=, or !=), you cannot place any whitespace between the two symbols.

Legal, declaration statements might include the following statements, which compare two values directly:

```
boolean isSixBigger = (6 > 5);
  // Value stored would be true
boolean isSevenSmallerOrEqual = (7 <= 4);
  // Value stored would be false
```

Tip

Variable names are easily identified as Boolean if you use a form of "to be" (such as "is" or "are") as part of the variable name.

The Boolean expressions are more meaningful when variables (that have been assigned values) are used in the comparisons, as in the following examples. In the first statement, the hours variable is compared to a constant value of 40. If the hours variable is not greater than 40, then the expression evaluates to `false`. In the second statement, the income variable must be greater than 100000 for the expression to evaluate to true.

```
boolean overtime = (hours > 40);
boolean highTaxBracket = (income > 100000);
```

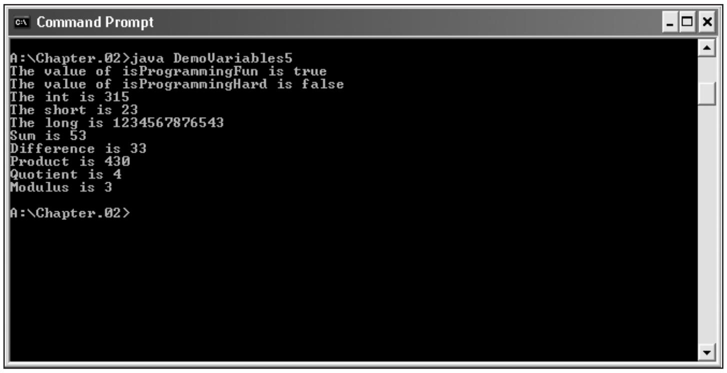
Next you will add two Boolean variables to the DemoVariables4.java file.

To add Boolean variables to a program:

1. Open the **DemoVariables4.java** file in your text editor and change the class to **DemoVariables5**.
2. Position the insertion point at the end of the line with the integer variable declarations, press **[Enter]**, and then type **`boolean isProgrammingFun = true, isProgrammingHard = false;`** on one line to add two new Boolean variables to the program.

   Next add some print statements to display the values.
3. Press **[Enter]**, and then type the following statements:

   ```
   System.out.println("The value of isProgrammingFun is "
     + isProgrammingFun);
   System.out.println("The value of isProgrammingHard is "
     + isProgrammingHard);
   ```
4. Save the file as **DemoVariables5.java**, compile it, and then test the program. The output appears in Figure 2-5.



```
A:\Chapter.02>java DemoVariables5
The value of isProgrammingFun is true
The value of isProgrammingHard is false
The int is 315
The short is 23
The long is 1234567876543
Sum is 53
Difference is 33
Product is 430
Quotient is 4
Modulus is 3

A:\Chapter.02>
```

Figure 2-5 Output of the DemoVariables5 program

LEARNING ABOUT FLOATING-POINT DATA TYPES

A **floating-point** number contains decimal positions. The Java programming language supports two floating-point data types: float and double. A **float** data type can hold values up to six or seven significant digits of accuracy. A **double** data type can hold 14 or 15 significant digits of accuracy. The term **significant digits** refers to the mathematical accuracy of a value. For example, a float given the value 0.324616777 will display as 0.324617 because the value is only accurate to the sixth decimal position. Table 2-4 shows the minimum and maximum values for each data type.

Tip: A float given the value 324616777 will display as 3.24617e+008, which means approximately 3.24617 times 10 to the 8th power, or 324617000. The *e* stands for exponent; the format is called scientific notation. The large value contains only six significant digits.

Table 2-4 Limits on floating-point values

Type	Minimum	Maximum	Size in Bytes
Float	$-3.4 * 10^{38}$	$3.4 * 10^{38}$	4
Double	$-1.7 * 10^{308}$	$1.7 * 10^{308}$	8

Tip: A value written as $-3.4 * 10^{38}$ indicates that the value is –3.4 multiplied by 10 to the 38th power, or 10 with 38 trailing zeros—a very large number.

Just as an integer constant, such as 178, is an int by default, a floating-point number constant such as 18.23 is a double by default. To store a value explicitly as a float, you can type the letter *F* after the number, as in `float pocketChange = 4.87F;`. You can type either a lowercase or an uppercase *F*. You also can type *D* (or *d*) after a floating-point value to indicate it is a double, but even without the *D*, the value will be stored as a double by default.

As with ints, you can perform the mathematical operations of addition, subtraction, multiplication, and division with floating-point numbers; however, you cannot perform modulus operations using floating-point values. (Floating-point division yields a floating-point result, so there is no remainder.)

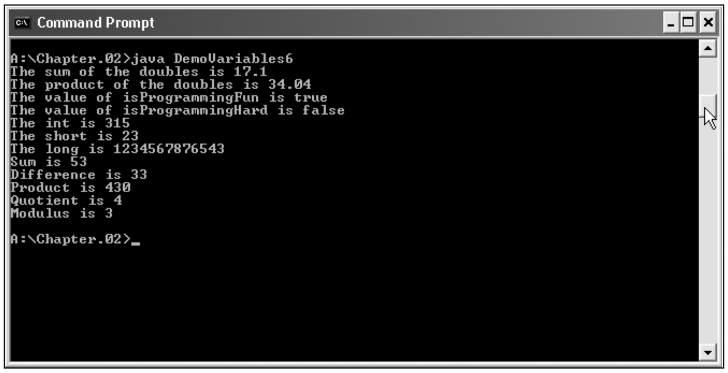
Next you will add some floating-point variables to the DemoVariables5.java file and perform arithmetic with them.

To add floating-point variables to the program:

1. Open the **DemoVariables5.java** file in your text editor and change the class name to **DemoVariables6**.
2. Position the insertion point after the line that declares the Boolean variables, press **[Enter]**, and then type **`double doubNum1 = 2.3, doubNum2 = 14.8, doubResult;`** on one line to add some new floating-point variables.
3. Press **[Enter]**, and then type the following statements to perform arithmetic and produce output:

```
doubResult = doubNum1 + doubNum2;
System.out.println("The sum of the doubles is "
  + doubResult);
doubResult = doubNum1 * doubNum2;
System.out.println("The product of the doubles is "
  + doubResult);
```

4. Save the file as **DemoVariables6.java**, compile it, and then run the program. The output is shown in Figure 2-6.



```
A:\Chapter.02>java DemoVariables6
The sum of the doubles is 17.1
The product of the doubles is 34.04
The value of isProgrammingFun is true
The value of isProgrammingHard is false
The int is 315
The short is 23
The long is 1234567876543
Sum is 53
Difference is 33
Product is 430
Quotient is 4
Modulus is 3

A:\Chapter.02>_
```

Figure 2-6 Output of the DemoVariables6 program

Understanding Numeric Type Conversion

When you are performing arithmetic with variables or constants of the same type, the result of the arithmetic retains the same type. For example, when you divide two integers, the result is an integer, and when you subtract two doubles, the result is a double. Often, however, you might want to perform mathematical operations on unlike types. In the following example you multiply an integer by a double:

```
int hoursWorked = 37;
double payRate = 6.73;
double grossPay = hoursWorked * payRate;
```

When you perform arithmetic operations with operands of unlike types, the Java programming language chooses a **unifying type** for the result. The Java programming language then **implicitly** (or automatically) converts nonconforming operands to the unifying type. The following list of operands ranks the order for establishing unifying types between two variables:

1. double
2. float
3. long
4. int
5. short
6. byte

Tip: An operand is simply any value used in an arithmetic or logical operation.

In other words, grossPay is the result of multiplication of an int and a double, so grossPay itself must be a double. Similarly, the addition of a short and an int results in an int.

You can **explicitly** (or purposely) override the unifying type imposed by the Java programming language by performing a type cast. **Type casting** involves placing the desired result type in parentheses, followed by the variable or constant to be cast. For example, two type casts are performed in the following code:

```
double bankBalance = 189.66;
float weeklyBudget = (float) bankBalance / 4;
  // weeklyBudget is 47.415, one-fourth of bankBalance
int dollars = (int) weeklyBudget;
  // dollars is 47, the integer part of weeklyBudget
```

Tip

It is easy to lose data when performing a cast. For example, the largest byte value is 127 and the largest int value is 2,147,483,647, so the following statements produce distorted results:

```
int anOkayInt = 200;
byte aBadByte = (byte)anOkayInt;
```

A byte is constructed from eight 1s and 0s, or binary digits. The first binary digit, or bit, holds a 0 or 1 to represent positive or negative. The remaining seven bits store the actual value. When the integer value 200 is stored in the byte variable, its large value consumes the eighth bit, turning it to a 1, and forcing the aBadByte variable to appear to hold the value –72, which is inaccurate and misleading.

The double value `bankBalance / 4` is converted to a float before it is stored in weeklyBudget, and the float value weeklyBudget is converted to an int before it is stored in dollars. When the float value is converted to an int, the decimal place values are lost.

Working with the char Data Type

You use the **char** data type to hold any single character. You place constant character values within single quotation marks because the computer stores characters and integers differently. For example, the statements `char aCharValue = '9';` and `int aNumValue = 9;` are legal. The statements `char aCharValue = 9;` and `int aNumValue = '9';` might produce undesirable results. If these variables are used in a println statement such as `System.out.println("aCharValue is" + aCharValue + "aNumValue is " + aNumValue);` the resulting output is a blank and the number 57, which are ASCII codes. Figure 2-7 shows ASCII decimal codes and character equivalents, which are covered later in this chapter. A number can be a character, but it must be enclosed in single quotation marks and declared as a char type. However, you cannot store an alphabetic letter in a numeric type. The following code shows how you can store any character string using the char data type:

```
    char myInitial = 'J';
    char percentSign = '%';
    char numThatIsAChar = '9';
```

A variable of type char can hold only one character. To store a string of characters, such as a person's name, you must use a data structure called a **String**. Unlike single characters, which use single quotation marks, string constants are written between double quotation marks. For example, the expression that stores the name Audrey as a string in a variable named firstName is `string firstName = "Audrey";`.

Tip

You will learn more about Strings in Chapter 7.

You can store any character—including nonprinting characters such as a backspace or a tab—in a char variable. To store these characters, you must use an **escape sequence**, which always begins with a backslash. For example, the following code stores a backspace character and a tab character in the char variables aBackspaceChar and aTabChar:

```
char aBackspaceChar = '\b';
char aTabChar = '\t';
```

In the preceding code, the escape sequence indicates a unique value for the character, instead of the letters b or t. Table 2-5 describes some common escape sequences that are used in the Java programming language.

Table 2-5 Common escape sequences

Escape Sequence	Description
\b	Backspace
\t	Tab
\n	Newline or linefeed
\f	Form feed
\r	Carriage return
\"	Double quotation mark
\'	Single quotation mark
\\	Backslash

Learning about ASCII and Unicode

The characters used in the Java programming language are represented in **Unicode**, which is a 16-bit coding scheme for characters. For example, the letter A actually is stored in computer memory as a set of 16 zeros and ones as 0000 0000 0100 0001 (the space inserted after each set of four digits is for readability). Because 16-digit numbers are difficult to read, programmers often use a shorthand notation called **hexadecimal**, or **base 16**. In hexadecimal shorthand, 0000 becomes 0; 0100 becomes 4; and 0001 becomes 1; so the letter A is represented in hexadecimal as 0041. You tell the compiler to treat the four-digit hexadecimal 0041 as a single character by preceding it with the \u escape sequence. Therefore, there are two ways to store the character A:

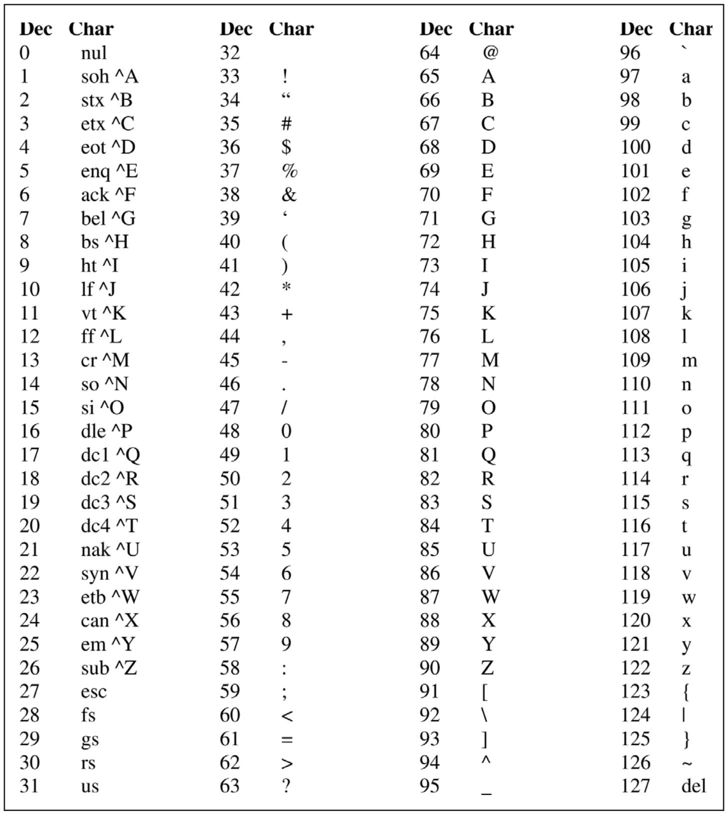
```
char letter = 'A';
char letter = '\u0041';
```

Tip

For more information about Unicode, go to *http://www.unicode.org*.

The second option using hexadecimal is obviously more difficult and confusing than the first method, so it is not recommended that you store letters of the alphabet using the hexadecimal method. However, there are some interesting values you can produce using the Unicode format. For example, the sequence '\u0007' is a bell that produces a noise if you send it to output. Letters from foreign alphabets that use characters instead of letters (Greek, Hebrew, Chinese, and so on) and other special symbols (foreign currency symbols, mathematical symbols, geometric shapes, and so on) are available using Unicode, but not on a standard keyboard, so it is important that you know how to use Unicode characters.

The most widely used character set is ASCII (American Standard Code for Information Interchange). There are 128 characters in the ASCII character set, which are shown in Figure 2-7 with their decimal code or numerical code and equivalent character representation. The first 32 characters and the last character are control characters and are nonprintable. These characters can be entered by holding down [Ctrl] and pressing a letter on the keyboard. For example, The Tab key or ^I(Ctrl I) produces a character 9, which produces a hard tab when pressed.



Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	nul	32		64	@	96	`
1	soh ^A	33	!	65	A	97	a
2	stx ^B	34	“	66	B	98	b
3	etx ^C	35	#	67	C	99	c
4	eot ^D	36	$	68	D	100	d
5	enq ^E	37	%	69	E	101	e
6	ack ^F	38	&	70	F	102	f
7	bel ^G	39	‘	71	G	103	g
8	bs ^H	40	(	72	H	104	h
9	ht ^I	41	)	73	I	105	i
10	lf ^J	42	*	74	J	106	j
11	vt ^K	43	+	75	K	107	k
12	ff ^L	44	,	76	L	108	l
13	cr ^M	45	-	77	M	109	m
14	so ^N	46	.	78	N	110	n
15	si ^O	47	/	79	O	111	o
16	dle ^P	48	0	80	P	112	p
17	dc1 ^Q	49	1	81	Q	113	q
18	dc2 ^R	50	2	82	R	114	r
19	dc3 ^S	51	3	83	S	115	s
20	dc4 ^T	52	4	84	T	116	t
21	nak ^U	53	5	85	U	117	u
22	syn ^V	54	6	86	V	118	v
23	etb ^W	55	7	87	W	119	w
24	can ^X	56	8	88	X	120	x
25	em ^Y	57	9	89	Y	121	y
26	sub ^Z	58	:	90	Z	122	z
27	esc	59	;	91	[	123	{
28	fs	60	<	92	\	124	|
29	gs	61	=	93	]	125	}
30	rs	62	>	94	^	126	~
31	us	63	?	95	_	127	del

Figure 2-7 ASCII Character Set

The relationship of ASCII and Unicode is that by adding eight zeros to any ASCII character gives the character's value in Unicode. The ASCII values are important because they allow you to show nonprintable characters, such as a carriage return, in decimal codes. Also, ASCII codes are often used when sorting numbers and strings. So when you need to sort characters in ascending order, numbers beginning with decimal 48 or 0 are sorted first, then capital letters starting with decimal 65 or A, and then lowercase letters starting with decimal code 97 or a.

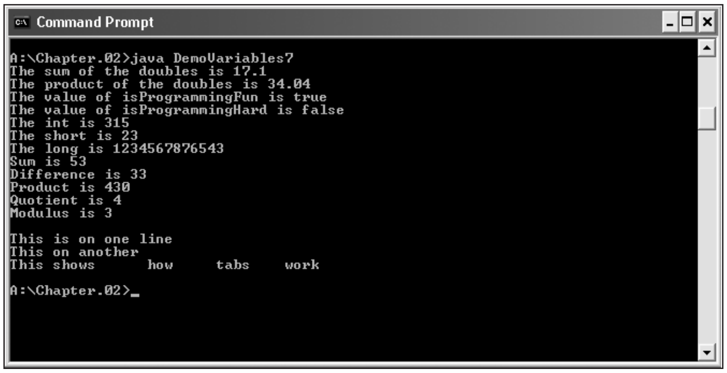
Next you will add statements to your DemoVariables6.java file to use the \n and \t escape sequences.

To use escape sequences in a program:

1. Open the **DemoVariables6.java** file in your text editor and change the class name to **DemoVariables7**.
2. Position the insertion point after the last method line in the program, press **[Enter]**, and then type the following:

   ```
   System.out.println("\nThis is on one line\nThis on
      another");
   System.out.println("This shows\thow\ttabs\twork");
   ```

3. Save the file as **DemoVariables7.java**, compile, and then test the program. Your output should look like Figure 2-8.



```
A:\Chapter.02>java DemoVariables7
The sum of the doubles is 17.1
The product of the doubles is 34.04
The value of isProgrammingFun is true
The value of isProgrammingHard is false
The int is 315
The short is 23
The long is 1234567876543
Sum is 53
Difference is 33
Product is 430
Quotient is 4
Modulus is 3

This is on one line
This on another
This shows      how     tabs    work

A:\Chapter.02>_
```

Figure 2-8 Output of the DemoVariables7 program demonstrating escape sequences

Chapter Summary

- Data is constant when it cannot be changed after a program is compiled; data is variable when it might change.
- Variables are named memory locations that your program can use to store values. You can name a variable using any legal identifier. A variable name must start with a

letter and cannot be a reserved keyword. You must declare all variables you want to use in a program. A variable declaration requires a type and a name; it can also include an assigned value.

- The Java programming language provides for eight primitive types of data: Boolean, byte, char, double, float, int, long, and short.
- You can declare multiple variables of the same type in separate statements or in a single statement, separated by commas.
- There are five standard arithmetic operators for integers: + – * / and %.
- Operator precedence is the order in which parts of a mathematical expression are evaluated. Multiplication, division, and modulus always take place prior to addition or subtraction in an expression. Right and left parentheses can be added within an expression when exceptions to this rule are required. When more than one pair of parentheses are added, the innermost expression surrounded by parentheses is evaluated first.
- A Boolean type variable can hold a `true` or `false` value.
- There are six comparison operators: > < == >= <= and !=.
- A floating-point number contains decimal positions. The Java programming language supports two floating-point data types: float and double.
- When you perform mathematical operations on unlike types, Java implicitly converts the variables to a unifying type. You can explicitly override the unifying type imposed by the Java programming language by performing a type cast.
- You use the char data type to hold any single character. You type constant character values in single quotation marks. You type String constants that store more than one character between double quotation marks. You can store some characters using an escape sequence, which always begins with a backslash.
- The characters used in Java programming are represented in the 16-bit Unicode scheme.

Review Questions

1. When data cannot be changed after a program is compiled, the data is ________________.
   a. constant
   b. variable
   c. volatile
   d. mutable

2. Which of the following is not a primitive data type in the Java programming language?
   a. Boolean
   b. byte
   c. int
   d. sector
3. Which of the following elements is not required in a variable declaration?
   a. a type
   b. an identifier
   c. an assigned value
   d. a semicolon
4. The assignment operator in the Java programming language is ________________.
   a. =
   b. ==
   c. :=
   d. ::
5. Which of the following values can you assign to a variable of type int?
   a. 0
   b. 98.6
   c. 'S'
   d. 5,000,000,000,000
6. Which of the following data types can store a value in the least amount of memory?
   a. short
   b. long
   c. int
   d. byte
7. The modulus operator ________________.
   a. is represented by a forward slash
   b. provides the remainder of integer division
   c. provides the remainder of floating-point division
   d. Answers b. and c. are correct.
8. According to the rules of operator precedence, division always takes place prior to ________________.
   a. multiplication
   b. modulus

c. subtraction

d. Answers a. and b. are correct.

9. A Boolean variable can hold ________________.

a. any character

b. any whole number

c. any decimal number

d. the values `true` or `false`

10. The "equal to" comparison operator is ________________.

a. =

b. ==

c. !=

d. !!

11. The value 137.68 can be held by a variable of type ________________.

a. int

b. float

c. double

d. Two of the preceding answers are correct

12. When you perform arithmetic with values of diverse types, the Java programming language ________________.

a. issues an error message

b. implicitly converts the values to a unifying type

c. requires you to explicitly convert the values to a unifying type

d. requires you to perform a cast

13. If you attempt to add a float, an int, and a byte, the result will be a(n) ________________.

a. float

b. int

c. byte

d. error message

14. You use a ________________ to explicitly override an implicit type.

a. mistake

b. type cast

c. format

d. type set

15. Which assignment is correct?
    a. `char aChar = 5;`
    b. `char aChar = "W";`
    c. `char aChar = '*';`
    d. Two of the preceding answers are correct
16. An escape sequence always begins with a(n) ________________.
    a. 'e'
    b. forward slash
    c. backslash
    d. equals sign
17. The 16-bit coding scheme employed by the Java programming language is ________________.
    a. Unicode
    b. ASCII
    c. EBCDIC
    d. hexadecimal

Exercises

1. What is the numeric value of each of the following expressions as evaluated by the Java programming language?
    a. 4 + 6 * 3
    b. 6 / 3 * 7
    c. 18 / 2 + 14 / 2
    d. 16 / 2
    e. 17 / 2
    f. 28 / 5
    g. 16 % 2
    h. 17 % 2
    i. 28 % 5
    j. 28 % 5 * 3 + 1
    k. (2 + 3) * 4
    l. 20 / (4 + 1)

2. What is the value of each of the following Boolean expressions?
   a. 4 > 1
   b. 5 <= 18
   c. 43 >= 43
   d. 2 == 3
   e. 2 + 5 == 7
   f. 3 + 8 <= 10
   g. 3 != 9
   h. 13 != 13
   i. –4 != 4
   j. 2 + 5 * 3 == 21
3. Which of the following expressions are illegal? For the legal expressions, what is the numeric value of each, as evaluated by the Java programming language?
   a. 2.3 * 1.2
   b. 5.67 – 2
   c. 25.0 / 5.0
   d. 7.0 % 3.0
   e. 8 % 2.0
4. Choose the best data type for each of the following so that no memory storage is wasted. Give an example of a typical value that would be held by the variable, and explain why you chose the type you did.
   a. your age
   b. the U.S. national debt
   c. your shoe size
   d. your middle initial
5. Use a text editor to write a Java program that declares variables to represent the length and width of a room in feet. Use Room as the class name. Assign appropriate values to the variables—for example, length = 15 and width = 25. Compute and display the floor space of the room in square feet (area = length * width). Display more than just a value as output; also display explanatory text with the value—for example, `The floor space is 375 square feet.`. Save the program as **Room.java** in the Chapter.02 folder on your Student Disk.
6. Use a text editor to write a Java program that declares variables to represent the length and width of a room in feet, and the price of carpeting per square foot in dollars and cents. Use Carpet as the class name. Assign appropriate values to the variables. Compute and display, with explanatory text, the cost of carpeting the room. Save the program as **Carpet.java** in the Chapter.02 folder on your Student Disk.

7. Write a program that declares variables to represent the length and width of a room in feet, and the price of carpeting per square yard in dollars and cents. Use Yards as the class name. Assign the value 25 to the length variable and the value 42 to the width variable. Compute and display the cost of carpeting the room. (*Hint*: There are nine square feet in one square yard.) Save the program as **Yards.java** in the Chapter.02 folder on your Student Disk.
8. Write a program that declares a minutes variable that represents minutes worked on a job, and assign a value. Use Time as the class name. Display the value in hours and minutes. For example, 197 minutes becomes 3 hours and 17 minutes. Save the program as **Time.java** in the Chapter.02 folder on your Student Disk.
9. Write a program that declares variables to hold your three initials. Display the three initials with a period following each one, as in J.M.F. Save the program as **Initials.java** in the Chapter.02 folder on your Student Disk.
10. Write a program that contains variables that hold your tuition fee and your book fee. Display the sum of the variables. Save the program as **Fees.java** in the Chapter.02 folder on your Student Disk.
11. Write a program that contains variables that hold your hourly rate of pay and number of hours that you worked. Display your gross pay, your withholding tax, which is 15 percent of your gross pay, and your net pay (gross pay – withholding). Save the program as **Payroll.java** in the Chapter.02 folder on your Student Disk.
12. a. Write a program that calculates and displays the conversion of $57 into dollar-bill form—20's, 10's, 5's, and 1's. Create a separate method to do the calculation and the display. Pass 57 as a variable to this method. Save the program as **Dollars.java** in the Chapter.02 folder on your Student Disk.

    b. In the Dollars.java program, alter the value of the variable that holds the amount of money. Run the program and confirm that the amount of each denomination calculates correctly.
13. Write a program that calculates and displays the amount of money you would have if you invested $1,000 at 5 percent interest for one year. Use the formula: Future Amount = Principal * Rate * Time. Save the program as **Interest.java** in the Chapter.02 folder on your Student Disk.
14. Write a program that illustrates the use of casting. Start with an integer, float, and double, and perform casts on the integer, float, and double. Print the results of each cast. Save the program as **Types.java** in the Chapter.02 folder on your Student Disk.
15. Write a program that displays FirstName, LastName, Address, and Phone on one line of output, and your first name, last name, address, and phone number on the second line. Make sure that your data lines up with the headings. Save the program as **Escape.java** in the Chapter.02 folder on your Student Disk.

16. Write a program to convert Fahrenheit temperature to Centigrade. Use the normal human body temperature of 98.6 degrees Fahrenheit, as the test case. Use the formula Centigrade = 5/9 (Fahrenheit -32). Save the program as **FahrenheitToCentigrade.java** in the Chapter.02 folder on your Student Disk.
17. Write a program that will output the following table of inventory items:

Item Num	Item Name	Units	On Hand
B1242	Bolt	Each	1000
N1242	Nut	Each	1200
W1242	Washer	Each	1150
N2323	Nails	Lbs	2250

Save the program as **Inventory.java** in the Chapter.02 folder on your Student Disk.

18. Each of the following files in the Chapter.02 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugTwo1.java will become FixDebugTwo1.java.
    a. DebugTwo1.java
    b. DebugTwo2.java
    c. DebugTwo3.java
    d. DebugTwo4.java

CASE PROJECT

Travel Tickets Company sells tickets for airlines, tours, and other travel-related services. Because long ticket numbers have often been entered incorrectly by agents, Travel Tickets has asked you to code a program that will indicate if a ticket number entry is invalid. The program also should prompt the agent to check and reenter the correct ticket number. Ticket numbers are 10 to 12 characters long. Ticket numbers are designed so that if you drop the last digit of the number, then divide the number by 7, the remainder of the division will be identical to the last dropped digit. This process is illustrated in the following example:

Step 1	Ticket number	12344321566
Step 2	Remove last digit, leaving	1234432156
Step 3	Divide remaining number by 7.	1234432156 divided by 7
Step 4	Remainder of 6 matches the digit dropped from the ticket.	
Step 5	Because the last digit matches the remainder of the division, the ticket number is valid.	

Although you cannot write the entire application, write a program that allows the declaration of a variable to hold the ticket number 1234432156 (first 10 digits of the 11-digit ticket number 12344321566), remove the last digit, divide the number by 7, and print the result of the division, which should be the digit 6 in this example for the ticket to be valid. Declare a second variable to hold the ticket number 2323454567 (first 10 digits of the 11-digit ticket number 23234545678), divide the number by 7, and print the result of the division. Is either of the entered ticket numbers valid by Travel Tickets rules? Save the case as **TicketNumber.java** in the Chapter.02 folder on your Student Disk.

CHAPTER

3

USING METHODS, CLASSES, AND OBJECTS

In this chapter, you will:

- Create methods with no arguments, a single argument, and multiple arguments
- Create methods that return values
- Learn about class concepts
- Create a class
- Use instance methods
- Declare objects
- Organize classes
- Use constructors

How do you feel about programming so far?" asks your new mentor, Lynn Greenbrier, who is head of computer programming for Event Handlers Incorporated.

"It's fun!" you reply. "It's great to see something actually work, but I still don't understand what the other programmers are talking about when they mention 'object-oriented programming.' I think everything is an object, and objects have methods, but I'm not really clear on this whole thing at all."

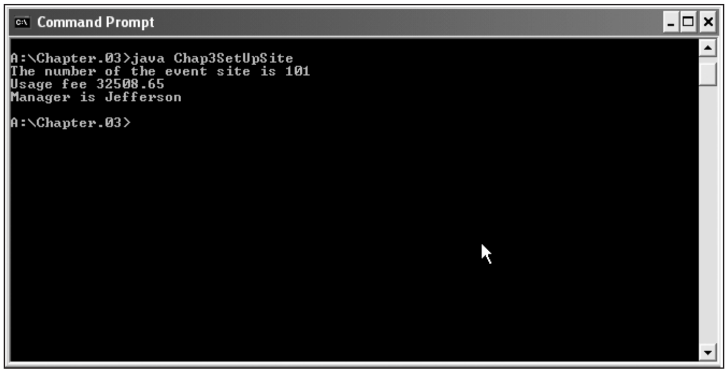
"Well then," Lynn says, "let me explain methods, classes, and objects."

Previewing the SetUpSite Program Using the EventSite Class

You will now preview the SetUpSite program that is saved on your Student Disk.

To preview the SetUpSite program on your Student Disk:

1. Start your text editor, open the **Chap3EventSite.java** file from the Chapter.03 folder on your Student Disk, and then examine the code. This file contains a class definition for a class that stores information about event sites used by Event Handlers Incorporated to host planned events.
2. Go to the command line, and then type **`javac Chap3EventSite.java`** to compile the Chap3EventSite.java file.
3. Use your text editor to open the **Chap3SetUpSite.java** file from the Chapter.03 folder, and then examine the code. This file contains a program that assigns values to the data regarding event sites that Event Handlers Incorporated uses to hold events. The program then displays that data on the screen. You will create a similar program in this chapter.
4. At the command line, type **`javac Chap3SetUpSite.java`** to compile the Chap3SetUpSite.java file.
5. Type **`java Chap3SetUpSite`** to execute the program. Information about an event site used by Event Handlers Incorporated will appear on the screen as shown in Figure 3-1.



```
A:\Chapter.03>java Chap3SetUpSite
The number of the event site is 101
Usage fee 32508.65
Manager is Jefferson

A:\Chapter.03>
```

Figure 3-1 Output of the Chap3SetUpSite program

Although the output shown in Figure 3-1 is modest, you have just witnessed several important programming concepts in action. The Chap3EventSite file contains a class definition that represents a real-life object—a site at which Event Handlers can hold an event. The class includes methods to assign values to and get values from data fields that pertain to event sites. The Chap3SetUpSite file creates an actual site with data representing a site number, a fee, and a manager's name. You will create similar files in this chapter.

Creating Methods with No Arguments, a Single Argument, and Multiple Arguments

A **method** is a series of statements that carry out a task. Any class can contain an unlimited number of methods. Within a class, the simplest methods you can invoke don't require any arguments or return any values. Consider the simple First Java program's First class that you saw in Chapter 1 and that appears in Figure 3-2.

```
public class First
{
   public static void main(String[] args)
   {
    System.out.println("First Java Program");
   }
}
```

Figure 3-2 The First class

Suppose you want to add three additional lines of output to this program to display your company's name and address. You can simply add three new println() statements, but instead you might choose to create a method to display the three lines.

Tip: Although there are differences, if you have used other programming languages, you can think of methods as being similar to procedures, functions, or subroutines.

There are two major reasons to create a method to display the three lines. First, the main() method will remain short and easy to follow because main() will contain just one statement to call a method, rather than three separate println() statements to perform the work of the method. What is more important is that a method is easily reusable. After you create the name and address method, you can use it in any program that needs the company's name and address. In other words, you do the work once, and then you can use the method many times. A method must include the following:

- A declaration (or header or definition)
- An opening curly brace
- A body
- A closing curly brace

The method declaration contains the following:

- Optional access modifiers
- The return type for the method

- The method name
- An opening parenthesis
- An optional list of method arguments (you separate the arguments with commas if there is more than one)
- A closing parenthesis

You first learned about access modifiers in Chapter 1. The access modifier for a method can be any of the following modifiers: `public`, `private`, `protected`, or `static`. Most often, methods are given `public` access. Endowing a method with `public` access means any class can use it. Additionally, like main(), any method that can be used from anywhere within the class (that is, any class-wide method) requires the keyword modifier `static`. Therefore, you can write the nameAndAddress() method shown in Figure 3-3. According to its declaration, the method is `public` and `static`. It returns nothing, so its return type is `void`. The method receives nothing, so its parentheses are empty. Its body, consisting of three println() statements, appears within curly braces.

```
public static void nameAndAddress()
{
  System.out.println("Event Handlers Incorporated");
  System.out.println("8900 U.S. Hwy 14");
  System.out.println("Crystal Lake, IL 60014");
}
```

Figure 3-3 The nameAndAddress() method

You place the entire method within the program that will use it, but not within any other method. Figure 3-4 shows where you can place a method in the First program.

```
public class First
{
   public static void main(String[] args)
   {
     System.out.println("First java program");
   }
|
// You can place additional methods here,
// outside the main() method
}
```

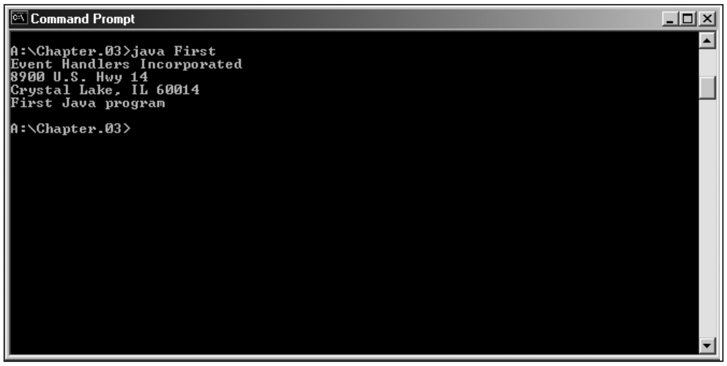
Figure 3-4 Placement of methods

If the main() method calls the nameAndAddress() method, then you simply use the nameAndAddress() method's name as a statement within the body of main(). Figure 3-5 shows the complete program.

```
public class First
{
   public static void main(String[] args)
   {
     nameAndAddress();
     System.out.println("First Java program");
   }
   public static void nameAndAddress()
   {
     System.out.println
       ("Event Handlers Incorporated");
     System.out.println("8900 U.S. Hwy 14");
     System.out.println("Crystal Lake, IL 60014");
   }
}
```

Figure 3-5 First class calling the nameAndAddress() method

The output from the program shown in Figure 3-5 appears in Figure 3-6. Because the main() method calls the nameAndAddress() method before it prints the phrase "First Java program", the name and address appear first in the output.



```
A:\Chapter.03>java First
Event Handlers Incorporated
8900 U.S. Hwy 14
Crystal Lake, IL 60014
First Java program

A:\Chapter.03>
```

Figure 3-6 Output of the First program with the nameAndAddress() method

If you want to use the nameAndAddress() method in another program, one additional step is required. In the Java programming language, the new program, with its own main() method, is a different class. If you place the nameAndAddress() method within the new class, the compiler will not recognize it unless you write it as **`First.nameAndAddress();`** to notify the new class that the method is located in the First class. Notice the use of the class name, followed by a dot, and then followed by the method. You have used similar syntax for the System.out.println() method.

Tip

Each of two different classes can have their own method named nameAndAddress(). Such a method in the second class would be entirely distinct from the identically named method in the first class.

Next you will create a new class named SetUpSite, which you will eventually use to set up one EventSite object. For now, the class will contain a main() method and a statementOfPhilosophy() method for Event Handlers Incorporated.

To create the SetUpSite class:

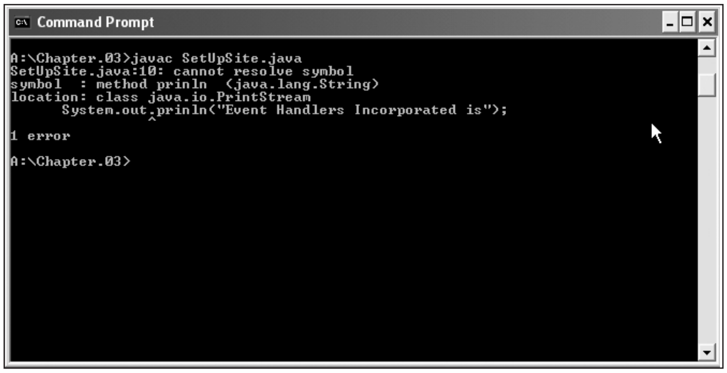
1. Open a new document in your text editor.
2. Type the following shell program to create a SetUpSite class and an empty main() method:

   ```
   public class SetUpSite
   {
     public static void main(String[] args)
     {
     }
   }
   ```

3. Place the insertion point to the right of the opening bracket in the main() method, press **[Enter]** to start a new line, and then type **statementOfPhilosophy();** between the curly braces of the main() method to place a call to a statementOfPhilosophy() method.
4. Type the following code for the statementOfPhilosophy() method just before the closing curly brace for the SetUpSite class code:

   ```
   public static void statementOfPhilosophy()
   {
     System.out.println("Event Handlers Incorporated is");
     System.out.println
       ("dedicated to making your event");
     System.out.println("a most memorable one.");
   }
   ```

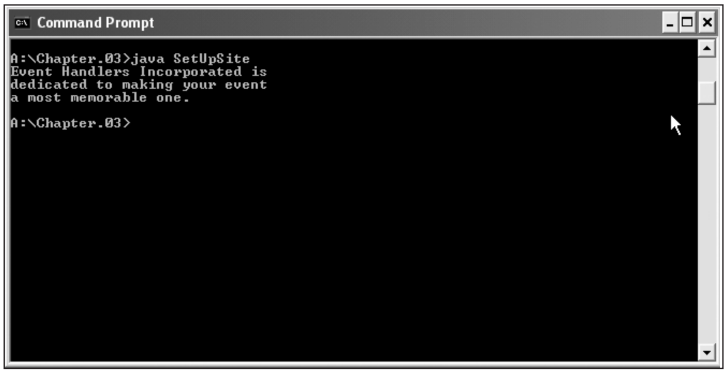
5. Save the file as **SetUpSite.java** in the Chapter.03 folder on your Student Disk.
6. At the command line, compile the program by typing **`javac SetUpSite.java`**. If you receive any error messages, you must correct their cause. Figure 3-7 shows the error message received when println()is spelled incorrectly within the SetUpSite.java file. Notice the message indicates that the file is SetUpSite.java, the line on which the error occurs is line 10, and the error is "method prinln…not found…" To help you, Java displays the offending line, and a caret appears just below and after the word that the compiler doesn't understand. To correct the spelling error, you return to the SetUpSite.java file, fix the mistake, save the file, and then compile it again.



```
A:\Chapter.03>javac SetUpSite.java
SetUpSite.java:10: cannot resolve symbol
symbol  : method prinln  (java.lang.String)
location: class java.io.PrintStream
      System.out.prinln("Event Handlers Incorporated is");
                ^
1 error

A:\Chapter.03>
```

Figure 3-7 SetUpSite program with a syntax error

7. Execute the program using the command **`java SetUpSite`**. Your output should look like Figure 3-8.



```
A:\Chapter.03>java SetUpSite
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.

A:\Chapter.03>
```

Figure 3-8 Output of the SetUpSite program

Help If you want one class to call the method of another class, both classes should reside in the same folder. If they are not saved in the same place, your compiler will issue the error message, "undefined variable or class name."

Next you will see how to call the statementOfPhilosophy() method from another class.

To call a method from another class:

1. First, open a new document in your text editor, and then enter the program that appears in Figure 3-9.

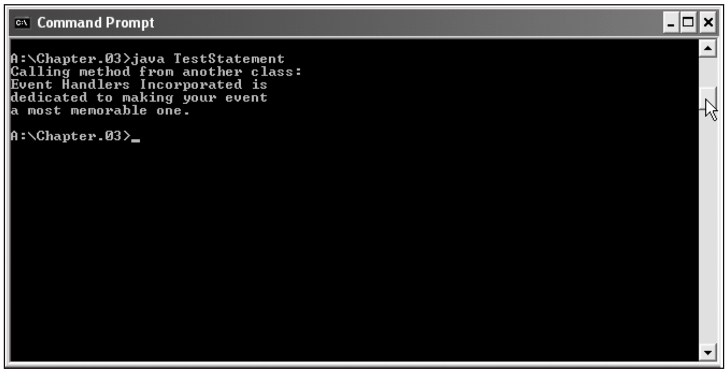
```
public class TestStatement
{
   public static void main(String[] args)
   {
     System.out.println
     ("Calling method from another class:");
     SetUpSite.statementOfPhilosophy();
   }
}
```

Figure 3-9 TestStatement program

2. Save the file as **TestStatement.java** in the Chapter.03 folder on your Student Disk.
3. Compile the program with the command **`javac TestStatement.java`**.

Help If necessary, correct any errors, save the file, and then repeat Step 3 to compile the file again.

4. Execute the program with the command **`java TestStatement`**. Your output should look like Figure 3-10.



```
A:\Chapter.03>java TestStatement
Calling method from another class:
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.

A:\Chapter.03>_
```

Figure 3-10 Output of the TestStatement program

It is also possible to revise SetUpSite.java by removing the main() method and the statementOfPhilosophy() call from the SetUpSite.java text document. The remaining text would appear as shown in Figure 3-11, with only the statementOfPhilosophy() method remaining.

To revise the SetUpSite program:

1. Remove the main() method and the statementOfPhilosophy() call from the SetUpSite.java text document.
2. Save the revised program as **SetUpSite2.java**.
3. Type **`javac SetUpSite2.java`** and compile the program as the SetUpSite2.class. Because there is no longer a main() method, you can't run the revised program by typing **`java SetUpSite2`**.

```
public class SetUpSite2
{
  public static void statementOfPhilosophy()
    {
      System.out.println("Event Handlers Incorporated is");
      System.out.println("dedicated to making your event");
      System.out.println("a most memorable one.");
    }
}
```

Figure 3-11 SetUpSite2 program

You can also include the new SetUpSite2 program as a class within the TestStatement program.

To add the SetUpSite2 program to TestStatement.java:

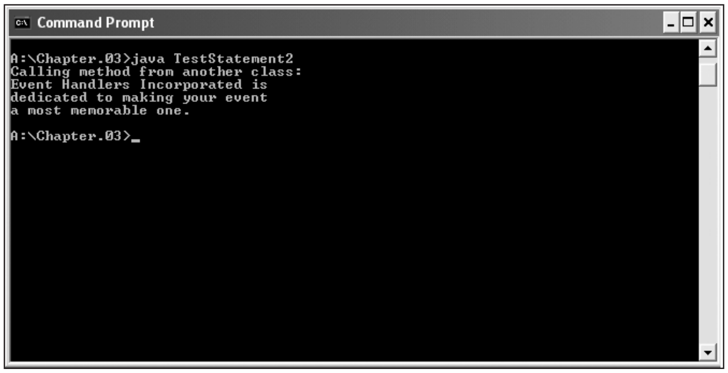
1. Open the **TestStatement.java** text document in your text editor, and then change the class name to **TestStatement2**.
2. Open the **SetUpSite2** text document containing the StatementOfPhilosophy() method in a separate text window.
3. Click **Edit** in the **SetUpSite2** text window menu, and then click **Select All** to select the **SetUpSite2** class text.
4. Click **Edit** and then click **Copy** to copy the **SetUpSite2** class text.
5. Switch to the open **TestStatement2.java** text document, place the insertion point in front of `public classTestStatement2` and then press **[Enter]** to create a blank line above `public classTestStatement2`.
6. Place the insertion point at the beginning of the blank line, click **Edit**, and then click **Paste** to paste the **SetUpSite2** class text.
7. Position the insertion point on the keyword **`public`** and delete it.
8. Save the program as **TestStatement2.java**.
9. Type the command **`javac TestStatement2.java`**.

10. Type the command **`java TestStatement2`**. Your revised TestStatement2 program should look like Figure 3-12. The output for this program is shown in Figure 3-13.

```
class SetUpSite2
{
  public static void statementOfPhilosophy()
  {
     System.out.println("Event Handlers Incorporated is");
     System.out.println("dedicated to making your event");
     System.out.println("a most memorable one.");
  }
}
public class TestStatement2
{
  public static void main(String[] args)
  {
     System.out.println
     ("Calling method from another class:");
     SetUpSite.statementOfPhilosophy();
  }
}
```

Figure 3-12 Revised TestStatement2 program

Tip: Because you must name a file from a class exactly the same as the public class name, you must remove the public class modifier from the SetUpSite2 class. If you don't, you will receive an error message. The compiler will not know which name to assign to the class file from two class programs if each program has a public class modifier.



```
A:\Chapter.03>java TestStatement2
Calling method from another class:
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.

A:\Chapter.03>_
```

Figure 3-13 Output of the TestStatement2 program

Creating Methods that Require a Single Argument

Some methods require additional information. If a method could not receive your communications, called **arguments**, then you would have to write an infinite number of methods to cover every possible situation. For example, when you make a restaurant reservation, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the method, which is then carried out in the same manner, no matter what date and time are involved. If you design a method to square numeric values, it makes sense to design a square() method that you can supply with an argument that represents the value to be squared, rather than having to develop a square1() method, a square2() method, and so on.

An important principle of object-oriented programming is the notion of implementation hiding. When you make a request to a method, you don't know the details of how the method is executed. For example, when you make a reservation, you do not need to know how the reservation is actually made at the restaurant—perhaps it is written in a book, marked on a large chalkboard, or entered into a computerized database. The implementation details don't concern you as a client, and if the restaurant changes its methods from one year to the next, the change does not affect your use of the reservation method. With well-written object-oriented programming methods, **implementation hiding** allows that the invoking program must know the name of the method and what type of information to send it (and what type of return to expect), but the program does not need to know how the method works. Additionally, you can substitute a new, improved method and, as long as the interface to the method does not change, you won't need to make any changes in programs that invoke the method.

Tip

At any call, the println() method can receive any one of an infinite number of arguments. No matter what message is sent to println(), the message displays correctly.

Tip

Hidden implementation methods are often referred to as existing in a black box.

When you write the method declaration for a method that can receive an argument, you must include the following items within the method declaration parentheses:

- The type of the argument
- A local name for the argument

For example, the declaration for a public method named predictRaise() that displays a person's salary plus a 10 percent raise could have the declaration `public void predictRaise(double moneyAmount)`. You can think of the parentheses in a method declaration as a funnel into the method—data arguments listed there are "dropped in" to the method.

The argument double moneyAmount within the parentheses indicates that the predictRaise() method will receive a figure of type double. Within the method, the figure (or salary amount) will be known as moneyAmount. Figure 3-14 shows a complete method.

```
public void predictRaise(double moneyAmount)
{
  double newAmount;
  newAmount = moneyAmount * 1.10;
  System.out.println
    ("With raise salary is " + newAmount);
}
```

Figure 3-14 The predictRaise() method

The predictRaise() method is a `void` method because it does not need to return any value to any class that uses it—its only function is to receive the moneyAmount value, multiply it by 1.10 (resulting in a 10 percent salary increase), and then display the result.

Within a program, you can call the predictRaise() method by using either a constant value or a variable as an argument. Thus, both `predictRaise(472.25);` and `predictRaise(mySalary);` invoke the predictRaise() method correctly, assuming that mySalary is declared as a double value and assigned an appropriate value. You can call the predictRaise() method any number of times, with a different constant or variable argument each time. Each of these arguments becomes known as moneyAmount within the method. The identifier moneyAmount holds any double value passed into the method. It's interesting to note that if the value in the method call is a variable, it might possess the same identifier as moneyAmount, or a different one, such as mySalary. The identifier moneyAmount is simply a placeholder while it is being used within the method, no matter what name it "goes by" in the calling program.

Tip

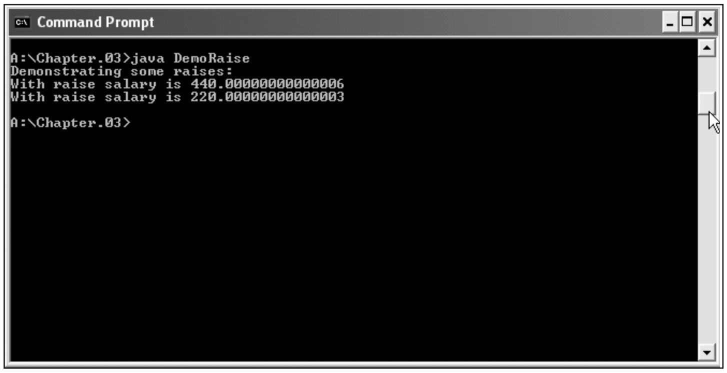
The variable moneyAmount is a local variable to the predictRaise() method.

If a programmer changes the way in which the 10 percent raise is calculated—for example, by coding `newAmount = moneyAmount + (moneyAmount * .10);`—no program that uses the predictRaise() method will ever know the difference. The program will pass a value into predictRaise() and then a calculated result will appear on the screen.

Figure 3-15 shows a complete program that uses the predictRaise() method twice. The program's output appears in Figure 3-16.

```
public class DemoRaise
{
   public static void main(String[] args)
   {
     double mySalary = 200.00;
     System.out.println("Demonstrating some raises:");
     predictRaise(400.00);
     predictRaise(mySalary);
   }
   public static void predictRaise(double moneyAmount)
   {
     double newAmount;
     newAmount = moneyAmount * 1.10;
     System.out.println("With raise salary is " +
       newAmount);
   }
}
```

Figure 3-15 Complete program using the predictRaise() method twice



```
A:\Chapter.03>java DemoRaise
Demonstrating some raises:
With raise salary is 440.00000000000006
With raise salary is 220.00000000000003

A:\Chapter.03>
```

Figure 3-16 Output of the DemoRaise program

Tip

Notice the output in Figure 3-16. Floating-point arithmetic is always somewhat imprecise.

Creating Methods that Require Multiple Arguments

A method can require more than one argument. You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas. For example, rather than creating a predictRaise() method that adds a 10 percent raise to every person's salary, you might prefer creating a method to which you can pass two values—the salary to be raised, as well as a percentage figure by which to raise it. Figure 3-17 shows a method that uses two such arguments.

```
public void predictRaiseGivenIncrease
  (double moneyAmount, double percentRate)
{
  double newAmount;
  newAmount = moneyAmount * (1 + percentRate);
  System.out.println("With raise salary is " +
    newAmount);
}
```

Figure 3-17 The predictRaiseGivenIncrease() method

Tip: Note that a declaration for a method that receives two or more arguments must list the type for each argument separately, even if the arguments have the same type.

In the header of the predictRaiseGivenIncrease() method, the arguments in parentheses are shown on separate lines to fit in this book's margin space. You could place the parentheses and arguments on the same line as the function header.

In Figure 3-17, two arguments (double moneyAmount and double percentRate) appear within the parentheses in the method header. The arguments are separated by a comma, and each argument requires its own named type (in this case, both are double) as well as an identifier. When values are passed to the method in a statement such as `predictRaiseGivenIncrease(mySalary,promisedRate);`, the first value passed will be referenced as moneyAmount within the method, and the second value passed will be referenced as percentRate. Therefore, it is very important that arguments passed to the method be passed in the correct order. The call `predictRaiseGivenIncrease(200.00,.10);` results in output representing a 10 percent raise based on a $200 salary amount (or $220), but `predictRaiseGivenIncrease(.10,200.00);` results in output representing a 200 percent raise based on a salary of 10 cents (or $20.10).

Tip: If two method arguments are the same type—for example, two doubles—passing them to a method in the wrong order results in a logical error. If a method expects arguments of diverse types, then passing arguments in reverse order constitutes a syntax error.

You can write a method so that it takes any number of arguments in any order. However, when you call a method, the arguments you send to a method must match—both in number and in type—the arguments listed in the method declaration. Thus, a method to compute an automobile salesperson's commission amount might require arguments such as an integer value of a car sold, a double percentage commission rate, and a character code for the vehicle type. The correct method will execute only when three arguments of the correct types are sent in the correct order.

Tip

The arguments in a method call are often referred to as actual parameters. The variables in the method declaration that accept the values from the actual parameters are the formal parameters.

CREATING METHODS THAT RETURN VALUES

The return type for a method can be any type used in the Java programming language, which includes the primitive (or scalar) types int, double, char, and so on, as well as class types (including class types you create). Of course, a method can also return nothing, in which case the return type is void.

A method's return type is known more succinctly as a method's type. For example, the declaration for the nameAndAddress() method is written `public static void nameAndAddress()`. This method is public and it returns no value, so it is type void. A method that returns `true` or `false`, depending on whether or not an employee worked overtime hours might be `public Boolean overtime()`. This method is public and it returns a Boolean value, so it is type Boolean.

Tip

In addition to returning the primitive types, a method can return a class type. If a class named BankLoan exists, a method might return an instance of a BankLoan, as in `public BankLoan approvalProcess()`. In other words, a method can return anything from a simple int to a complicated BankLoan with 20 data fields.

The header for a method that displays a raise amount is `public static void predictRaise(double moneyAmount)`. If you want to create a method to return the new, calculated salary value rather than display the raised salary, the header would be `public double calculateRaise(double moneyAmount)`. Figure 3-18 shows this method.

```
public void calculateRaise(double moneyAmount)
{
  double newAmount;
  newAmount = moneyAmount * 1.10;
  return newAmount;
}
```

Figure 3-18 The calculateRaise() method

Notice the return type double in the method header. Also notice the return statement that is the last statement within the method. The **return statement** causes the value stored in newAmount to be sent back to any method that calls the calculateRaise() method.

If a method returns a value, then when you call the method, you will usually want to use the returned value, although you are not required to do so. For example, when you invoke the calculateRaise() method, you might want to assign the value to a double variable named myNewSalary, as in `myNewSalary = calculateRaise(mySalary);`. The calculateRaise() method returns a double, so it is appropriate to assign the returned value to a double variable.

Alternately, you can choose to display a method's returned value directly, without storing it in any variable, as in `System.out.println("New salary is " + calculateRaise(mySalary));`. In this last statement, the call to the calculateRaise() method is made from within the println() method call. Because calculateRaise() returns a double, you can use the method call calculateRaise() in the same way that you would use any simple double value. For example, besides printing the value of calculateRaise(), you can perform math with it, assign it, and so on.

Next you will add a method to the SetUpSite2 class that both receives an argument and returns a value. The purpose of the method is to take the current year and calculate how long Event Handlers Incorporated has been in business.

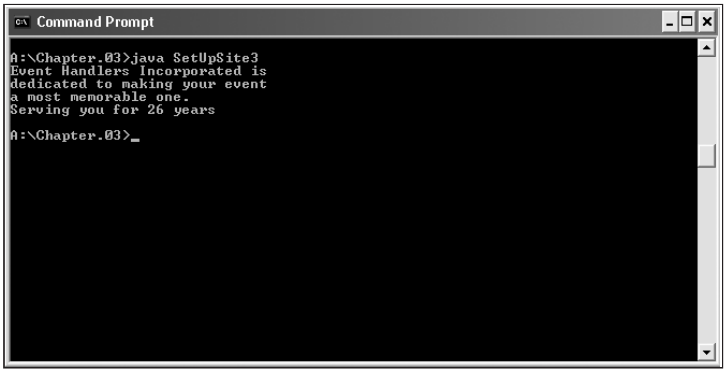
To add a method that receives an argument and returns a value:

1. Open the **SetUpSite2.java** file in the text editor, and then change the class file to **SetUpSite3**.
2. Position the insertion point to the right of the opening curly brace of the main() method of the class, and then press **[Enter]** to start a new line.
3. Type **`int currentYear = 2003;`** to declare a variable to hold the current year, and then press **[Enter]**.
4. Type **`int age;`** to declare another variable to hold the age of Event Handlers Incorporated.
5. Position the insertion point at the end of the call to the statementOfPhilosophy() method in the main() method of the class, and then press **[Enter]** to start a new line. You will add a call to receive the current year and calculate how long Event Handlers Incorporated has been in business by subtracting the year of its inception, which is 1977.
6. Type **`age = calculateAge(currentYear);`** as a call to a calculateAge() method.
7. Press **[Enter]**, and then type **`System.out.println("Serving you for " + age + " years");`** to print the number of years the company has been in business. Now you will write the calculateAge() method.
8. Position the insertion point after the closing bracket of the statementOfPhilosophy() method, press **[Enter]** to start a new line before the closing bracket of the program, and then enter the method shown in Figure 3-19. The method will receive an integer value. Within the calculateAge() method, the value will be known as currDate. Note that the name currDate does not possess the same identifier as currentYear, which is the variable being passed in, although it could. Notice also that the method declaration indicates an int value will be returned.

```
public static int calculateAge(int currDate)
{
  int yrs;
  yrs = currDate - 1977;
  return yrs;
}
```

Figure 3-19 The calculateAge() method

9. Save the file as **SetUpSite3.java**, compile it, and correct any errors. Execute the program and confirm that the results are correct. See Figure 3-20.



```
A:\Chapter.03>java SetUpSite3
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
Serving you for 26 years

A:\Chapter.03>_
```

Figure 3-20 Output of the SetUpSite3 program with the calculateAge() method

Learning about Class Concepts

When you think in an object-oriented manner, everything is an object, and every object is a member of a class. You can think of any inanimate physical item as an object—your desk, your computer, and the building in which you live are all called objects in everyday conversation. You can also think of living things as objects—your houseplant, your pet fish, and your sister are objects. Events are also objects—the stock purchase you made, the mortgage closing you attended, or a graduation party that was held in your honor are all objects.

Everything is an object, and every object is a member of a more general class. Your desk is a member of the class that includes all desks, and your pet fish a member of the class that contains all fish. An object-oriented programmer would say that your desk is an instance of the Desk class and your fish is an instance of the Fish class. These statements represent **is-a relationships**, relationships that are correct only if said in the proper order. You can say, "My oak desk with the scratch on top *is a* Desk and my goldfish named Moby *is a* Fish." You can't say, "My Desk *is an* oak desk with a scratch on top or a Fish *is a* goldfish named Moby," because both a Desk and a Fish are much more general. The difference between a class and an object parallels the difference between abstract and concrete. An object is an **instantiation** of a class, or one tangible example of a class. Your goldfish, my guppy, and the zoo's shark each constitute one instantiation of the Fish class.

The concept of a class is useful because of its reusability. Objects inherit attributes from classes, and all objects have predictable attributes because they are members of certain classes. For example, if you are invited to a graduation party, you automatically know many things about the object (the party). You assume there will be a starting time, a certain number of guests, some quantity of food, and some kind of gifts. You understand what a party entails because of your previous knowledge of the Party class of which all parties are members. You don't know the number of guests, what food will be served, or what gifts will be received at this particular party, but you understand that because all parties have guests and refreshments, then this one must too. Similarly, you can also apply this thinking to the stock market. Even though every stock market purchase is unique, each stock purchase must have a dollar amount and a number of shares.

Tip

The data components of a class are often referred to as the instance variables of that class. Also, class object attributes are often called fields to help distinguish them from other variables you might use.

In addition to their attributes, class objects have methods associated with them, and every object that is an instance of a class is assumed to possess the same methods. For example, for all parties, at some point, you must set the date and time. You might name these methods setDate() and setTime(). Party guests need to know the date and time, and might use methods named getDate() and getTime() to find out the date and time of the party.

Your graduation party, then, might possess the identifier myGraduationParty. As a member of the Party class, myGraduationParty, like all parties, might have data members for the date and time methods setDate() and setTime(). When you use them, the setDate() and setTime() methods require arguments, or information passed to them. For example, myGraduationParty.setDate("May 12") and myGraduationParty.setTime("6 P.M.") invoke methods that are available for myGraduationParty and send it arguments. When you use an object and its methods, think of being able to send a message to the object to direct it to accomplish some task—you can tell the party object named myGraduationParty to set the date and time you request. Even though yourAnniversaryParty is also a member of the Party class, and even though it also has setDate() and setTime() methods, the arguments you send to yourAnniversaryParty will be different from those you send to myGraduationParty. Within any object-oriented program, you are continuously making requests to objects' methods, and often including arguments as part of those requests.

Additionally, some methods used in a program must return a message or value. If one of your party guests uses the getDate() method, the guest hopes that the method will respond with the desired information. Similarly, within object-oriented programs, methods are often called upon to return a piece of information to the source of the request.

For example, a method within a Payroll class that calculates federal withholding tax might return a tax figure in dollars and cents, and a method within an Inventory class might return true or false, depending on the method's determination of whether or not an item is at the reorder point.

There are two parts to object-oriented programming. First, you must create the classes of objects from which objects will be instantiated, and second, you must write other classes to use the objects (and their data and their methods). The same programmer does not need to accomplish these two tasks. Often, you will write programs that use classes created by others; similarly, you might create a class that others will use to instantiate objects within their own programs. You can call a program or class that instantiates objects of another prewritten class a **class client** or **class user**.

Tip

The System class that you used in Chapter 1 is an example of using a class that was written by someone else. You did not have to create it or its println() method; both were created for you by Java's creators.

Creating a Class

When you create a class, first you must assign a name to the class, and then you must determine what data and methods will be part of the class. Suppose you decide to create a class named Employee. One instance variable of Employee might be an employee number, and two necessary methods might be a method to set (or provide a value for) the employee number and another method to get (or retrieve) that employee number. To begin, you create a class header with three parts:

- An optional access modifier
- The keyword class
- Any legal identifier you choose for the name of your class

For example, a header for an Employee class is `public class Employee`. The keyword `public` is a class access modifier. You can use the following class access modifiers when defining a class: `public`, `final`, or `abstract`. If you do not specify an access modifier, access becomes `public`.

Public classes are accessible by all objects, which means that public classes can be **extended**, or used as a basis for any other class. The most liberal form of access is `public`. Public access means that if you develop a good Employee class, and some day you want to develop two more-specific classes, SalariedEmployee and HourlyEmployee, then you will not have to start from scratch. Each new class can become an extension of the original Employee class, inheriting its data and methods. The other access modifiers (or the omission of any access modifier) impose at least some limitations on extensibility. (You use the other access modifiers only under special circumstances.) You will use the public access modifier for most of your classes.

After writing the class header `public class Employee`, you write the body of the Employee class, containing its data and methods, between a set of curly braces. Figure 3-21 shows the shell for the Employee class.

```
public class Employee
{
  //Instance variables and methods go here
}
```

Figure 3-21 Employee class shell

You place the instance variables, or fields, for the Employee class as statements within the curly braces. For example, you can declare an employee number that will be stored as an integer simply as `int empNum;`. However, programmers frequently include an access modifier for each of the class fields and declare the empNum as `private int empNum;`.

The allowable field modifiers are `private`, `public`, `static`, and `final`. Most class fields are `private`, which provides the highest level of security. Private access means that no other classes can access a field's values, and only methods of the same class are allowed to set, get, or otherwise use private variables. Private access is sometimes called **information hiding**, and is an important component of object-oriented programs. A class's private data can be changed or manipulated only by a class's own methods, and not by methods that belong to other classes. In contrast, most class methods are not usually `private`. The resulting private data/public method arrangement provides a means for you to control outside access to your data—only a class's nonprivate methods can be used to access a class's private data. The situation is similar to hiring a public receptionist to sit in front of your private office and control which messages you receive (perhaps deflecting trivial or hostile ones) and which messages you send (perhaps checking your spelling, grammar, and any legal implications). The way in which the nonprivate methods are written controls how you use the private data.

Tip

The field modifiers are the same as the method modifiers with one addition—the `final` modifier. You will learn to use the `final` modifier in Chapter 4.

The entire class so far appears in Figure 3-22. It defines a public class named Employee, with one field, which is a private integer named empNum.

Next you will create a class to store information about event sites for Event Handlers Incorporated.

To create the class:

1. Open a new document in your text editor.

```
public class Employee
{
  private int empNum;
}
```

Figure 3-22 Employee class with one data field

2. Type the following class header and the curly braces to surround the class body:

   ```
   public class EventSite
   {
   }
   ```

3. Type **`private int siteNumber;`** between the curly braces to insert the private data field that will hold an integer site number for each event site used by the company.
4. Save the file as **EventSite.java** in the Chapter.03 folder on your Student Disk.
5. To ensure you have not made any typographical errors, compile the class by typing **`javac EventSite.java`** at the command-line prompt. If necessary, correct any errors, save your work, and then compile again. Do not execute the class.

Using Instance Methods

Besides data, classes contain methods. For example, one method you need for an Employee class that contains an empNum is the method to retrieve (or return) any Employee's empNum for use by another class. A reasonable name for this method is getEmpNum(), and its declaration is `public int getEmpNum()` because it will have public access, return an integer (the employee number), and possess the identifier getEmpNum(). Figure 3-23 shows the complete getEmpNum() method.

```
public int getEmpNum()
{
  return empNum;
}
```

Figure 3-23 The getEmpNum() method

The getEmpNum() method contains just one statement: the statement that accesses the value of the private empNum field.

Notice that, unlike the class methods you created earlier in this chapter, the getEmpNum() method does not employ the `static` modifier. The keyword `static` is used for class-wide methods, but not for methods that "belong" to objects. If you are creating a program with a main() method that you will execute to perform some task, then many of your methods will be static so you can call them from within main(). However, if you are creating a class from which objects will be instantiated, most methods will probably be non-static because you will associate the methods with individual objects. Methods used with object instantiations are called **instance methods**.

Tip: You can call class methods without creating an instance of the class. Instance methods require an instantiated object.

Next you will add an instance method to the EventSite2 class that will retrieve the value of an event site's number.

To add an instance method to the EventSite2 class:

1. Open the **EventSite.java** file in your text editor and change the class name to **EventSite2**.
2. Within the EventSite2 class's curly braces and after the declaration of the siteNumber field, enter the following getSiteNumber() method to return the site number to any calling class:

   ```
   public int getSiteNumber()
   {
     return siteNumber;
   }
   ```
3. Save the file as **EventSite2.java**.

When a class contains data fields, you want a means to assign values to the data fields. For an Employee class with an empNum field, you need a method with which to set the empNum. Figure 3-24 shows a method that sets the empNum. The method is a void method because there is no need to return any value to a calling program. The method receives an integer, locally called emp, to be assigned to empNum.

```
public static void setEmpNum(int emp)
{
  empNum = emp;
}
```

Figure 3-24 The setEmpNum() method

Next you will add a setSiteNumber() method to the EventSite2 class. This method takes an integer argument and assigns it to the siteNumber of an EventSite object.

To add a method to the EventSite3 class:

1. Open the **EventSite2.java** program, change the class name to **EventSite3**, then add the following method to the **EventSite3.java** file after the final curly brace for the getSiteNumber() method, but prior to the closing curly brace for the EventSite3 class:

```
public void setSiteNumber(int n)
{
  siteNumber = n;
}
```

The argument *n* represents any number sent to this method.

2. Save the file as **EventSite3.java**, compile it, and then correct any syntax errors. (You cannot run this file as a program.)

Declaring Objects

Declaring a class does not create any actual objects. A class is just an abstract description of what an object will be like if any objects are ever actually instantiated. Just as you might understand all the characteristics of an item you intend to manufacture long before the first item rolls off the assembly line, you can create a class with fields and methods long before you instantiate any objects which are members of that class.

A two-step process creates an object that is an instance of a class. First, you supply a type and an identifier, just as when you declare any variable, and then you allocate computer memory for that object. For example, you might define an integer as **`int someValue;`** and you might define an Employee as **`Employee someEmployee;`**, where someEmployee stands for any legal identifier you choose to represent an Employee.

When you declare an integer as **`int someValue;`**, you notify the compiler that an integer named someValue will exist, and you reserve computer memory for it at the same time. When you declare the someEmployee instance of the Employee class, you are notifying the compiler that you will use the identifier someEmployee. However, you are not yet setting aside computer memory in which the Employee named someEmployee might be stored—that is done only for primitive type variables. To allocate the needed memory, you must use the **new operator**. After you define someEmployee with the **`Employee someEmployee;`** statement, the statement that actually sets aside enough memory to hold a **`someEmployee = new Employee();`**.

You can also define and reserve memory for someEmployee in one statement, as in **`Employee someEmployee = new Employee();`**. In this statement, Employee is the object's type (as well as its class), and someEmployee is the name of the object. The equals sign is the assignment operator, so a value is being assigned to someEmployee. The **`new`** operator is allocating a new, unused portion of computer memory for someEmployee. The value that the statement is assigning to someEmployee is a memory address at which someEmployee is to be located. You do not need to be concerned with what the actual memory address is—when you refer to someEmployee, the compiler will locate it at the appropriate address for you—but someEmployee does need to know its own address.

Tip

Every object name is also a reference—that is, a computer memory location.

The last portion of the statement, `Employee()`, with its parentheses, looks suspiciously like a method name. In fact, it is the name of a method that constructs an Employee object. Employee() is a **constructor method**. You will write your own constructor methods later in this section, but when you don't write a constructor method for a class object, Java writes one for you, and the name of the constructor method is always the same as the name of the class whose objects it constructs.

Next you will instantiate an EventSite3 object.

To instantiate an object:

1. Open the **SetUpSite3.java** file from the Chapter.03 folder in your text editor. Change the class name to **SetUpSite4**.
2. Place the insertion point at the end of `int age` within the main() method, press **[Enter]** to start a new line, and then type **`EventSite3 oneSite = new EventSite3();`** to allocate memory for a new EventSite4 object named oneSite.
3. Save the file as **SetUpSite4.java** and then compile it. If necessary, correct any errors, and save and compile again.

After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. For example, if an Employee class method to change a salary is written using the code in Figure 3-25, and an Employee was declared with `Employee clerk = new Employee();`, then the clerk's salary can be changed to 350.00 with the call `clerk.changeSalary(350.00);`. The method changeSalary() is applied to the object clerk, and the argument 350.00 (a double type value) is passed to the method.

```
public void changeSalary(double newAmount)
{
  salary = newAmount;
}
```

Figure 3-25 The changeSalary() method

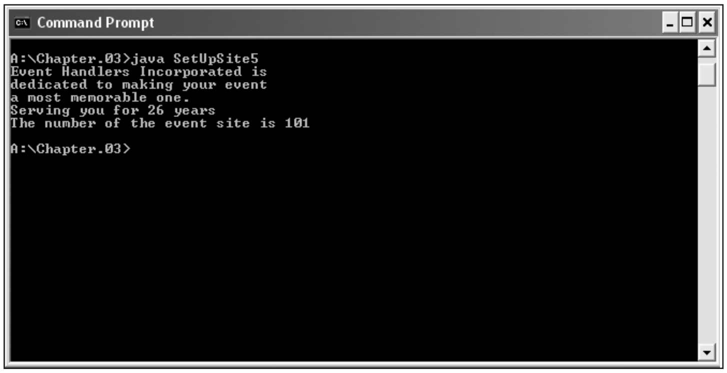
Within the same program, the statements `Employee secretary = new Employee();` and `secretary.changeSalary(420.00);` would apply the same changeSalary() method, but using a different argument value, to different objects that belong to the same class.

Next you will add calls to the getSiteNumber() and setSiteNumber() methods for the oneSite object member of the EventSite3 class.

To add the calls to the methods for the oneSite object member:

1. Open the file **SetUpSite4.java** in your text editor and change the class to **SetUpSite5**.

2. Just below the declaration for oneSite, to provide the SetUpSite5() method with a variable to hold any site number returned from the getSiteNumber() method, type **`int number;`**, and then press **[Enter]**.
3. Next call the method setSiteNumber() to set the site number for oneSite. Type **`oneSite.setSiteNumber(101);`**. The number in parentheses could be any integer number.
4. After the statement that prints the age of the company, `System.out.println("Serving you for " + age + " years");`, to call the getSiteNumber() method and assign its return value to the number variable, type **`number = oneSite.getSiteNumber();`**, and then press **[Enter]**.
5. To add a call to the println() method to display the value stored in number, type **`System.out.println("The number of the event site is " + number);`**.
6. Save the program file as **SetUpSite5.java** in the Chapter.03 folder on your Student Disk.
7. Compile the program by typing **`javac SetUpSite5.java`**. Correct any errors and compile again, if necessary.
8. Execute the program by typing **`java SetUpSite5`**. Your output should look like Figure 3-26.



```
A:\Chapter.03>java SetUpSite5
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
Serving you for 26 years
The number of the event site is 101

A:\Chapter.03>
```

Figure 3-26 Output of the SetUpSite5 program

ORGANIZING CLASSES

Most classes you create will have more than one data field and more than two methods. For example, in addition to requiring an employee number, an Employee needs a last name, a first name, and a salary, as well as methods to set and get those fields. Figure 3-27 shows how you could code the data fields for Employee.

```
public class Employee
{
  private int empNum;
  private String empLastName;
  private String empFirstName;
  private double empSalary;
//Methods will go here
}
```

Figure 3-27 Employee class with data fields

Although there is no requirement to do so, most programmers place data fields in some logical order at the beginning of a class. For example, the empNum is most likely used as a unique identifier for each employee (what database users often call a **primary key**), so it makes sense to list the employee number first in the class. An employee's last name and first name "go together," so it makes sense to store these two Employee components adjacently. Despite these common-sense rules, there is a lot of flexibility in how you position your data fields within any class.

Tip

A unique identifier is one that should have no duplicates within an application. For example, an organization might have many employees with the last name Johnson or a salary of $400.00, but there will be only one employee with employee number 128.

Because there are two String components in the current Employee class, they might be declared within the same statement, such as `private String empLastName, empFirstName;`. However, it is usually easier to identify each Employee field at a glance if the fields are listed vertically.

Even if the only methods created for the Employee class include one set method and one get method for each instance variable, eight methods are required. Consider an Employee record for most organizations and you will realize that many more fields are often required (such as address, phone number, hire date, number of dependents, and so on), as well as many more methods. Finding your way through the list can become a formidable task. For ease in locating class methods, many programmers store them in alphabetical order. Other programmers arrange values in pairs of "get" and "set" methods, an order that also results in functional groupings. Figure 3-28 shows how the complete class definition for an Employee might appear.

```
public class Employee
{
  private int empNum;
  private String empLastName;
  private String empFirstName;
  private double empSalary;
  public int getEmpNum()
  {
    return empNum;
  }
  public void setEmpNum(int num)
  {
    empNum = num;
  }
  public String getFirstName()
  {
    return empFirstName;
  }
  public void setFirstName(String name)
  {
    empFirstName = name;
  }
  public String getLastName()
  {
    return empLastName;
  }
  public void setLastName(String name)
  {
    empLastName = name;
  }
  public double getEmpSal()
  {
    return empSalary;
  }
  public void setEmpSal(double sal)
  {
    empSalary = sal;
  }
}
```

Figure 3-28 Employee class with data fields and methods

The Employee class is still not a particularly large class, and each of its methods is very short, but it is already becoming quite difficult to manage. It certainly can support some well-placed comments, as shown in Figure 3-29.

```
//Programmer: Joyce Farrell
//Date April 22, 2003
//Employee.java to hold employee data

public class Employee
{
  //private data members
  private int empNum;
  private String empLastName;
  private String empFirstName;
  private double empSalary;

  //getEmpNum method returns employee number
  public int getEmpNum()
  {
    return empNum;
  }
  //setEmpNum method returns employee number
  public void setEmpNum(int num)
  {
    empNum = num;
  }
  //... and so on
}
```

Figure 3-29 Employee class with data fields, methods, and comments

Tip: Although good program comments are crucial to creating understandable code, you will not be asked to include them in most examples in this book in an effort to save space.

To expand the EventSite3 class to contain data fields and methods:

1. Open the **EventSite3.java** file from the Chapter.03 folder in the text editor, and change the class name to **EventSite4**. Your program looks like Figure 3-30.

```
public class EventSite4
{
  private int siteNumber;
  public int getSiteNumber()
  {
    return siteNumber;
  }
  public void setSiteNumber(int n)
  {
    siteNumber = n;
  }
}
```

Figure 3-30 EventSite4.java class

You will add two new data fields to the EventSite4 class: a double to hold a usage fee for the site, and a String to hold the site manager's last name.

2. Position the insertion point at the end of the declaration of the `private int siteNumber;` variable, press **[Enter]** to start a new line, and then type **`private double usageFee;`** and **`private String managerName;`** on separate lines.

   You will also enter four new methods to set and get data from each of the two new fields. To ensure that the methods are easy to locate later, you will place them in alphabetical order within the class.

3. Position the insertion point after the end of the closing curly brace of the getSiteNumber() method, press **[Enter]** to start a new line, and then enter the following getUsageFee() method:

```
public double getUsageFee()
{
  return usageFee;
}
```

4. Position the insertion point at the end of the `private String managerName;` declaration, press **[Enter]** to start a new line, and then enter the following getManagerName() method:

```
public String getManagerName()
{
  return managerName;
}
```

5. Position the insertion point after the closing bracket of the getSiteNumber() method, press **[Enter]** to start a new line, and then enter the following setUsageFee() method:

```
public void setUsageFee(double amt)
{
  usageFee = amt;
}
```

6. Position the insertion point after the closing curly brace of the getUsageFee() method, press **[Enter]**, and then enter the following setManagerName() method:

```
public void setManagerName(String name)
{
  managerName = name;
}
```

7. Start a line above each of the methods, and add a comment describing the function of the method.

8. Save the file as **EventSite4.java** and compile it by typing the command **`javac EventSite4`**. If necessary, correct any errors, save the file, and then compile again.

You have created an EventSite4 class that contains both data and methods. However, no actual event sites exist yet. You must write a program that instantiates one or more EventSite objects to give actual values to the data fields for that object, and to manipulate the data in the fields using the class methods. Next you will create a program to test the new, expanded EventSite4 class.

To create the test program:

1. Open a new document in the text editor, and then enter the class that tests the new expanded EventSite4 class. The class should look like Figure 3-31.

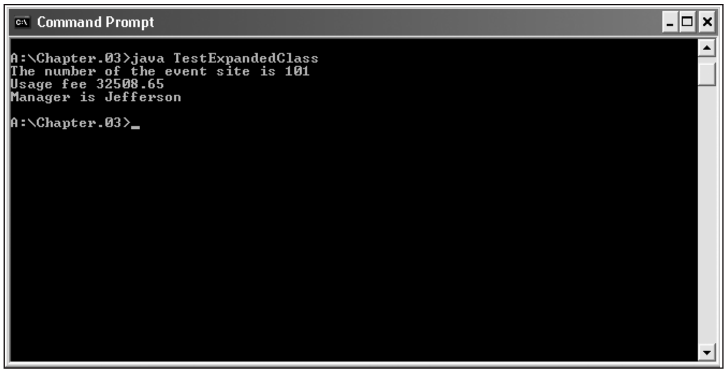
```
//Programmer: Joyce Farrell
//Date: April 22, 2003
//Program: TestexpandedClass
//Tests the expanded EventSite4 class
public class TestExpandedClass
{
  public static void main(String[] args)
  {
    EventSite4 oneSite = new EventSite4()
    oneSite.setSiteNumber(101);
    oneSite.setUsageFee(32508.65);
    oneSite.setManagerName("Jefferson");
    System.out.print("The number of the event site is ");
    System.out.println(oneSite.getSiteNumber());
    System.out.println("Usage fee "
       + oneSite.getUsageFee());
    System.out.println("Manager is "
       + oneSite.getManagerName());
  }
}
```

Figure 3-31 The TestExpandedClass class

Tip

You should get into the habit of documenting your programs with your name, today's date, and a brief explanation of the program. Your instructor might also ask you to insert additional information as comment text.

2. Save the file as **TestExpandedClass.java** in the Chapter.03 folder on your Student Disk. Compile the program and correct any errors, if necessary.
3. Execute the class with the command-line statement **`java TestExpandedClass`**. Your output should look like Figure 3-32.



```
A:\Chapter.03>java TestExpandedClass
The number of the event site is 101
Usage fee 32508.65
Manager is Jefferson

A:\Chapter.03>_
```

Figure 3-32 Output of the TestExpandedClass program

USING CONSTRUCTORS

When you create a class, such as Employee, and instantiate an object with a statement such as `Employee chauffeur = new Employee();`, you are actually calling a method named Employee() that is provided by the Java compiler. A **constructor method** is a method that establishes an object. The constructor method named Employee() establishes one Employee with the identifier chauffeur, and provides the following specific initial values to the Employee's data fields:

- Numeric fields are set to 0 (zero).
- Character fields are set to Unicode '\u0000'.
- The Boolean fields are set to `false`.
- The object type fields are set to null (or empty).

If you do not want an Employee's fields to hold these default values, or if you want to perform additional tasks when you create an Employee, then you can write your own constructor method. Any constructor method you write must have the same name as the class it constructs, and constructor methods cannot have a return type. For example, if every Employee has a starting salary of $300.00, then you could write the constructor method for the Employee class that appears in Figure 3-33. Any Employee instantiated will have a default empSal figure of 300.00.

```
Employee()
{
  empSal = 300.00;
}
```

Figure 3-33 Employee class constructor

You can write any statement in a constructor. Although you usually have no reason to do so, you could print a message from within a constructor or perform any other task. Next you will add a constructor to the EventSite4 class, and demonstrate that it is called when an EventSite4 object is instantiated.

To add a constructor to the EventSite4 class:

1. Open the **EventSite4.java** file in your text editor. Change the class name to **EventSite5**.
2. Place the insertion point at the end of the line containing the last field declaration `private String managerName;`, and then press **[Enter]** to start a new line.
3. Add the following constructor function that sets any EventSite siteNumber to 999 and any manager's name to "ZZZ" upon construction:

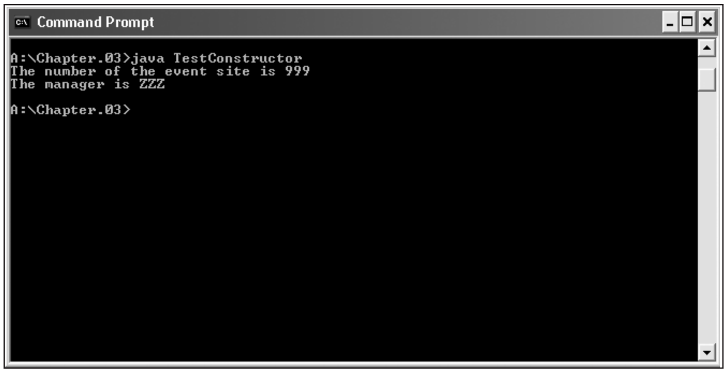
```
EventSite5()
{
  siteNumber = 999;
  managerName = "ZZZ";
}
```

4. Save the file as **EventSite5.java**, compile it, and correct any errors.
5. Open a new text file and create a test class named **TestConstructor** using the code shown in Figure 3-34.

```
public class TestConstructor
{
  public static void main(String[] args)
  {
    EventSite5 oneSite = new EventSite5();
    System.out.print("The number of the event site is ");
    System.out.println(oneSite.getSiteNumber());
    System.out.print("The manager is ");
    System.out.println(oneSite.getManagerName());
  }
}
```

Figure 3-34 TestConstructor class

6. Save the file as **TestConstructor.java** in the Chapter.03 folder on your Student Disk, compile the file, and correct any syntax errors.
7. Execute the program and confirm that it declares a oneSite object of type EventSite5, calls the constructor, and assigns the indicated initial values, as shown in Figure 3-35.



```
A:\Chapter.03>java TestConstructor
The number of the event site is 999
The manager is ZZZ

A:\Chapter.03>
```

Figure 3-35 Output of the TestConstructor program

Chapter Summary

- A method is a series of statements that carry out a task. Methods must include a declaration (or header or definition), an opening curly brace, a body, and a closing curly brace. A method declaration contains optional access modifiers, the return type for the method, the method name, an opening parenthesis, an optional list of method arguments, and a closing parenthesis.
- You place a method within the program that will use it, but not within any other method. If you place a method call within a class that does not contain the method, you must use the class name, followed by a dot, followed by the method. Some methods require a message or argument.
- When you write the method declaration for a method that can receive an argument, you need to include the type of the argument and a local name for the argument within the method declaration parentheses. You can call a method within a program using either a constant value or a variable as an argument.
- You can pass multiple arguments to methods by listing the arguments and separating them by commas within the call to the method. The arguments you send to the method must match both in number and in type the parameters listed in the method declaration.
- The return type for a method (the method's type) can be any Java type, including void. You use a return statement to send a value back to a program that calls a method.
- Objects inherit attributes from classes. Class objects have attributes and methods associated with them. Class instance methods that will be used with objects are not usually static. You can send messages to objects. Additionally, some methods used in a program must return a message or value.

- A class header contains an optional access modifier, the keyword class, and any legal identifier you choose for the name of your class. The instance variables, or fields, of a class are placed as statements within the class's curly braces.
- Declaring a class does not create any actual objects; you must instantiate any objects that are members of a class. To create an object that is an instance of a class, you supply a type and an identifier, and then you allocate computer memory for that object using the `new` operator.
- A constructor method establishes an object and provides specific initial values for the object's data fields. A constructor method always has the same name as the class of which it is a member. By default, numeric fields are set to 0 (zero), character fields are set to Unicode '\u0000', Boolean fields are set to false, and object type fields are set to null.

Review Questions

1. Methods must include all of the following except ______________.
   a. a declaration
   b. a call to another method
   c. curly braces
   d. a body
2. All method declarations contain ______________.
   a. the keyword `static`
   b. one or more access modifiers
   c. arguments
   d. parentheses
3. A public method named computeSum() is located in classA. To call the method from within classB, use the statement ______________.
   a. `computeSum(classB);`
   b. `classB(computeSum());`
   c. `classA.computeSum();`
   d. You cannot call computeSum() from within classB.
4. Which of the following method declarations is correct for a method named displayFacts() if the method receives an int argument?
   a. `public void int displayFacts()`
   b. `public void displayFacts(int)`
   c. `public void displayFacts(int data)`
   d. `public void displayFacts()`

5. The method with the declaration `public int aMethod(double d)` is a method type ________________.
   a. `int`
   b. `double`
   c. `void`
   d. You cannot determine the method type.
6. Which of the following is a correct call to a method declared as `double aMethod(char code)`?
   a. `double aMethod();`
   b. `double aMethod('V');`
   c. `aMethod(int 'M');`
   d. `aMethod('Q');`
7. A method is declared as `public void showResults(double d, int i)`. Which of the following is a correct method call?
   a. `showResults(double d, int i);`
   b. `showResults(12.2, 67);`
   c. `showResults(4, 99.7);`
   d. Two of the above answers are correct.
8. The method with the declaration `public char procedure(double d)` has a method type of ________________.
   a. `public`
   b. `char`
   c. `procedure`
   d. `double`
9. The method public `Boolean testValue(int response)` returns ________________.
   a. a Boolean value
   b. an integer value
   c. no value
   d. You cannot determine what is returned.
10. Which of the following could be the last legally coded line of a method declared as `public int getVal(double sum)`?
    a. `return;`
    b. `return 77;`
    c. `return 2.3;`
    d. Any of the above could be the last coded line of the method.

11. The data components of a class often are referred to as the ________________ of that class.
    a. access types
    b. instance variables
    c. methods
    d. objects
12. Class objects have both attributes and ________________.
    a. fields
    b. data
    c. methods
    d. instances
13. You send messages to an object through its ________________.
    a. fields
    b. methods
    c. classes
    d. data
14. A program or class that instantiates objects of another prewritten class is a(n) ________________.
    a. class client
    b. superclass
    c. object
    d. patron
15. The body of a class is written ________________.
    a. as a single statement
    b. within parentheses
    c. between curly braces
    d. as a method call
16. Most class fields are ________________.
    a. private
    b. public
    c. static
    d. final

17. The concept of allowing a class's private data to be changed only by a class's own methods is known as ________________.
    a. structured logic
    b. object orientation
    c. information hiding
    d. data masking
18. When you declare a variable, as in `double salary;`, you ________________.
    a. also must explicitly allocate memory for it
    b. need not explicitly allocate memory for it
    c. must explicitly allocate memory for it only if it is stored in a class
    d. can declare it to use no memory
19. If a class is named Student, then the class constructor name is ________________.
    a. any legal Java identifier
    b. any legal Java identifier that begins with S
    c. StudentConstructor
    d. Student
20. If you use the default constructor, ________________.
    a. numeric fields are set to 0 (zero)
    b. character fields are set to blank
    c. Boolean fields are set to true
    d. object type fields are set to 0 (zero)

Exercises

1. Name any device you use every day. Discuss how implementation hiding is demonstrated in the way this device works. Is it a benefit or a drawback to you that implementation hiding exists for methods associated with this object?
2. a. Create a class named Numbers whose main() method holds two integer variables. Assign values to the variables. Create two additional methods, sum() and difference(), that compute the sum of and difference between the values of the two variables, respectively. Each method should perform the computation and display the results. In turn, call each of the two methods from main(), passing the values of the two integer variables. Save the program as **Numbers.java** in the Chapter.03 folder on your Student Disk.
   b. Add a method named product() to the Numbers class. The product() method should compute the multiplication product of two integers, but not display the answer. Instead, it should return the answer to the calling main() program, which displays the answer. Save the program as **Numbers2.java** in the Chapter.03 folder on your Student Disk.

3. Create a class named Eggs. Its main() method holds an integer variable named numberOfEggs to which you will assign a value. Create a method to which you pass numberOfEggs. The method displays the eggs in dozens; for example, 50 eggs is 4 full dozen (with 2 eggs remaining). Save the program as **Eggs.java** in the Chapter.03 folder on your Student Disk.
4. Create a class named Monogram. Its main() method holds three character variables that hold your first, middle, and last initials, respectively. Create a method to which you pass the three initials and which displays the initials twice—once in the order first, middle, last, and a second time in traditional monogram style (first, last, middle). Save the program as **Monogram.java** in the Chapter.03 folder on your Student Disk.
5. Create a class named Exponent. Its main() method holds an integer value, and in turn passes the value to a method that squares the number and to a method that cubes the number. The main() method prints the results. Create the two methods that respectively square and cube an integer that is passed to them, returning the calculated value. Save the program as **Exponent.java** in the Chapter.03 folder on your Student Disk.
6. Create a class named Cube that displays the result of cubing a number. Pass a number to a method that cubes a number and returns the result. The display should execute within the main() method that calls the cube method. Save the program as **Cube.java** in the Chapter.03 folder on your Student Disk.
7. Create a program that displays the result of a sales transaction. The calculation requires three numbers. The first number represents the product price. The second number is the salesperson commission. These two numbers should be added together. The third value represents a customer discount; subtract this third number from the result of the addition. Create two classes. The first class, Transaction, contains the method to do the calculation. The three numbers are passed to this method by a statement in the other class. The display is performed in the class that calls the calculation method. Save the program as **Calculator.java** in the Chapter.03 folder on your Student Disk.
8. Write a program that displays the result of dividing two numbers and also displays any remainder. Do the calculation and display in the same method, which is a separate method from the main() method. Save the program as **Divide.java** in the Chapter.03 folder on your Student Disk.
9. a. Create a class named Pizza. Data fields include a String for toppings (such as pepperoni), an integer for diameter in inches (such as 12), and a double for price (such as 13.99). Include methods to get and set values for each of these fields. Save the class as **Pizza.java** in the Chapter.03 folder on your Student Disk.

   b. Create a class named TestPizza that instantiates one Pizza object and demonstrates the use of the Pizza set and get methods. Save this class as **TestPizza.java** in the Chapter.03 folder of your Student Disk.

10. a. Create a class named Student. A Student has fields for an ID number, number of credit hours earned, and number of points earned. (For example, many schools compute grade point averages based on a scale of 4, so a three-credit-hour class in which a student earns an A is worth 12 points.) Include methods to assign values to all fields. A Student also has a field for grade point average. Include a method to compute the grade point average field by dividing points by credit hours earned. Write methods to display the values in each Student field. Save this class as **Student.java** in the Chapter.03 folder on your Student Disk.
    b. Write a class named ShowStudent that instantiates a Student object from the class you created. Compute the Student grade point average, and then display all the values associated with the Student. Save the program as **ShowStudent.java** in the Chapter.03 folder on your Student Disk.
    c. Create a constructor method for the Student class you created. The constructor should initialize each Student's ID number to 9999 and his or her grade point average to 4.0. Write a program that demonstrates that the constructor works by instantiating an object and displaying the initial values. Save the program as **Student2.java**.
11. a. Create a class named Circle with fields named radius, area, and diameter. Include a constructor that sets the radius to 1. Also include methods named setRadius(), getRadius(), computeDiameter(), which computes a circle's diameter, and computeArea(), which computes a circle's area. (The diameter of a circle is twice its radius, and the area is 3.14 multiplied by the square of the radius.) Save the class as **Circle.java** in the Chapter.03 folder of your Student Disk.
    b. Create a class named TestCircle whose main() method declares three Circle objects. Using the setRadius() method, assign one Circle a small radius value and assign another a larger radius value. Do not assign a value to the radius of the third circle; instead, retain the value assigned at construction. Call computeDiameter() and computeArea() for each circle and display the results. Save the program as **TestCircle.java** in the Chapter.03 folder on your Student Disk.
12. a. Create a class named Checkup with fields that hold a patient number, two blood pressure figures (systolic and diastolic), and two cholesterol figures (LDL and HDL). Include methods to get and set each of the fields. Include a method named computeRatio() that divides LDL cholesterol by HDL cholesterol and displays the result. Include an additional method named ExplainRatio() that explains that HDL is known as "good cholesterol" and that a ratio of 3.5 or lower is considered optimum. Save the class as **Checkup.java** in the Chapter.03 folder of your Student Disk.
    b. Create a class named TestCheckup whose main() method declares four Checkup objects. Provide values for each field for each patient. Then display the values. Blood pressure numbers are usually displayed with a slash between the systolic and diastolic values. (Typical numbers are values such as 110/78 or 130/90.) With the cholesterol figures, display the explanation of the cholesterol ratio calculation. (Typical numbers are values such as 100 and 40 or 180 and 70.) Save the program as **TestCheckup.java** in the Chapter.03 folder on your Student Disk.

13. Write a program that displays employee IDs and first and last names of employees. Use two classes. The first class named Emp contains the employee data and separate methods to set the IDs and names. The other class creates objects for the employees and uses the objects to call the set methods. Create several employees and display their data. Save the program as **Employee.java** in the Chapter.03 folder on your Student Disk.
14. Write a program that displays an invoice of several items. It should contain the item name, quantity, price, and total cost on each line for the quantity and item cost. Use two classes. The first class Inv contains the item data and methods to get and set the item name, quantity, and price. The other class creates objects for the items and uses the objects to call the set and get methods. Save the program as **Invoice.java** in the Chapter.03 folder on your Student Disk.
15. Write a program that schedules several meetings for a meeting room. It should contain the day of the week, starting time, and ending time for each meeting. Use two classes. The first class contains the meeting data and methods to get and set the day of the week and starting and ending times. The other class creates objects for the meetings and uses the objects to call the set and get methods. Save both classes in the program as **RoomSchedule.java** in the Chapter.03 folder on your Student Disk.
16. Write a program that calculates and displays the weekly salary for an employee who earns $25 an hour, works 40 regular hours, 13 overtime hours, and earns time and one-half (wage * 1.5) for overtime hours worked. Create a separate method to do the calculation and return the result to be displayed. Save the program as **Salary.java** in the Chapter.03 folder on your Student Disk.
17. a. Write a program that calculates and displays the conversion of $57 into dollar-bill form—20's, 10's, 5's, and 1's. Create a separate method to do the calculation and display. Pass 57 as a variable to this method. Save the program as **Dollars.java** in the Chapter.03 folder on your Student Disk.

    b. In the Dollars.java program, alter the value of the variable that holds the amount of money. Run the program and confirm that the amount of each denomination calculates correctly.
18. Write a program that calculates and displays the amount of money you would have if you invested $1,000 at 5 percent interest for one year. Create a separate method to do the calculation and return the result to be displayed. Save the program as **Interest.java** in the Chapter.03 folder on your Student Disk.
19. a. Create a bank account named Account. The class should have one instance variable named balance. Write two constructors, one to set the value of balance to 0.0 when called, and a second that will receive a balance as a double value passed to the constructor. Write instance methods to add to, subtract from, and set the balance to 0.0. Write other instance methods as needed.

    b. Write a program to instantiate an Account object. With the Account object, open an account, add a deposit to the account, withdraw an amount from the account, and close the account. After each transaction, print the Account balance. Save the program as **TestAccount.java** in the Chapter.03 folder on your Student Disk.

20. Each of the following files saved in the Chapter.03 folder on your Student Disk has syntax and/or logical errors. In each case, determine and fix the problem. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugThree1.java will become FixDebugThree1.java.
    a. DebugThree1.java
    b. DebugThree2.java
    c. DebugThree3.java
    d. DebugThree4.java

CASE PROJECT

Event Handlers wants to develop an Employee program to set and retrieve employee ID numbers, employee salaries, and first and last employee names. The current list of employees and their data are as follows:

Table 3-1 Employee data

Employee Name	Employee ID	Employee Salary
Kim Yee	101	$40,000.00
John Reynolds	102	$55,000.00
Elena Gonzales	103	$50,500.00
Jim O'Shea	104	$75,000.00

The program will require a constructor that sets the names to "Unknown", ID's to "0", and Salaries to "0.00". As a programmer, your task is to write the necessary classes to accomplish the tasks given by the specifications.

CHAPTER

4

ADVANCED OBJECT CONCEPTS

In this chapter, you will:

- Understand blocks and scope
- Overload a method
- Learn about ambiguity
- Send arguments to constructors
- Overload constructors
- Learn about the `this` reference
- Work with constants
- Use automatically imported, prewritten constants and methods
- Use prewritten imported methods
- Learn about Gregorian calendars

Lynn Greenbrier, your mentor at Event Handlers Incorporated, pops her head into your cubicle on Monday morning. "How's the programming going?" she asks.

"I'm getting the hang of using objects," you tell her, "but I want to create lots of objects, and it seems like I will need so many methods for the classes that I create that it will be very hard to keep track of them." You pause a moment and add, "And all these set methods are driving me crazy. I wish an object could just start with values."

"Anything else bothering you?" Lynn asks.

"Well," you reply, "since you asked, shouldn't some objects and methods that are used by all kinds of programmers already be created for me? I can't be the first person who ever thought about taking a square root of a number or calculating a billing date for 10 days after service."

"You're in luck!" Lynn smiles. "Java's creators have already thought about these things. Let me tell you about some of the more advanced things you can do with your classes."

UNDERSTANDING BLOCKS AND SCOPE

Within any class or method, the code between a pair of curly braces is called a **block**. For example, the program shown in Figure 4-1 contains two blocks. The first block, or **outside block**, begins immediately after the method declaration and ends at the end of the method. The second block, or **inside block**, is contained within the second set of curly braces and contains three statements: the declaration of anotherNumber and two println() statements. The inside block is **nested** within the outside block. A block can exist entirely within another block, or entirely outside and separate from another block, but blocks can never overlap.

```
public static void methodWithTwoBlocks()
{
  int aNumber = 22;
    // aNumber comes into existence
  System.out.println("Number is " + aNumber);
  {
    int anotherNumber = 99;
    // anotherNumber comes into existence
    System.out.println("aNumber is " + aNumber);
    System.out.println("anotherNumber is " +
      anotherNumber);
  } // End of block - anotherNumber ceases to exist
  System.out.println("aNumber is " + aNumber);
} // End of outer block - aNumber ceases to exist
```

Figure 4-1 The methodWithTwoBlock() method

If you declare a variable in one program that you write, you cannot use that variable in another program. Similarly, when you declare a variable within a block, you cannot reference that variable outside the block. The portion of a program within which you can reference a variable is the variable's **scope**. A variable comes into existence, or **comes into scope**, when you declare it. A variable ceases to exist, or **goes out of scope**, at the end of the block in which it is declared.

Tip

Although you can create as many variables and blocks as you need within any program, it is not wise to do so without a reason. The use of unnecessary variables and blocks increases the likelihood of improper use of variable names and scope.

In the methodWithTwoBlocks() method shown in Figure 4-1, the variable aNumber exists from the point of its declaration until the end of the method. This means aNumber exists both in the outer block and within the inner block, and can be used anywhere in the method. The variable anotherNumber comes into existence within the inner block; anotherNumber ceases to exist when the inner block ends, and cannot be used beyond its block.

Figure 4-2 shows some invalid statements. The first assignment `aNumber = 75;` is invalid because aNumber has not been declared yet. Similarly, Invalid statement 2, `anotherNumber = 489;`, is invalid because it has not been declared yet. Statement 3 is also invalid because anotherNumber still has not been declared. After you declare anotherNumber, you can use it for the remainder of the block, but Invalid statement 4 is outside the block and anotherNumber has gone out of scope. The last statement in Figure 4-2, `aNumber = 29;`, will not work because it falls outside the block in which aNumber was declared; it actually falls outside the methodWithTwoBlocks() method.

```
public static void methodWithTwoBlocks()
{
   aNumber = 75;// Invalid statement
   int aNumber = 22;
   System.out.println("aNumber is " + aNumber);
   anotherNumber = 489;// Invalid statement 2
   {
      anotherNumber = 165;// Invalid statement 3
      int anotherNumber = 99;
      System.out.println("aNumber is " + aNumber);
      System.out.println("anotherNumber is " +
        anotherNumber);
   }
   System.out.println("aNumber is " + aNumber);
   System.out.println("anotherNumber is " +
        anotherNumber); // Invalid statement 4
}
anumber = 29;// Invalid statement 5
```

Figure 4-2 The methodWithTwoBlock() method with some invalid statements

Tip

You are not required to vertically align the opening and closing braces for a block, but your programs are much easier to read if you do.

Within a method, you can declare a variable with the same name multiple times, as long as each declaration is in its own, nonoverlapping block. For example, the two declarations of variables named someVar in Figure 4-3 are valid because each variable is contained within its own block. The first instance of someVar has gone out of scope before the second instance comes into scope.

```
public static twoDeclarations()
{
   { // Begin first block
     int someVar = 7;
     System.out.println(someVar);
   } // End first block
   { // Begin second block
     int someVar = 845;
     System.out.println(someVar);
   } // End second block
}
```

Figure 4-3 The twoDeclarations() method

You cannot declare the same variable name more than once within a block. For example, in Figure 4-4, the second declaration of aValue causes an error because you cannot declare the same variable twice within the outer block of the method. By the same reasoning, the third declaration of aValue is also invalid, even though it appears within a new block. The block that contains the third declaration is entirely within the outside block, so the first declaration of aValue has not gone out of scope.

```
public static methodWithRedeclarations()
{
   int aValue = 35;
   System.out.println(aValue);
   int aValue = 99; // Invalid - second declaration
   {
     int anotherValue = 58; // Valid
     int aValue = 99; // Invalid - third declaration
     // This block is inside the outer block
   }
}
```

Figure 4-4 Invalid methodWithRedeclarations()

If you declare a variable within a class, and use the same variable name within a method of the class, then the variable used inside the method takes precedence, or **overrides**, the first variable. For example, consider a class that holds Employee information including two integer fields, aNum and aDept, as shown in Figure 4-5.

Figure 4-5 shows an Employee class with two integers and two void methods. When a TestEmployee2 program instantiates an Employee object with a statement such as `Employee adminAssistant = new Employee();`, then either empMethod() or anotherEmpMethod() can be called using the adminAssistant object and the dot operator (`.`), as in `adminAssistant.empMethod()` or `adminAssistant.anotherEmpMethod()`.

```
public class Employee
{
   private int aNum = 44;
   private int aDept = 55;
   public void empMethod()
   {
     int aNum = 88;
     //aNum overrides the class variable aNum
     System.out.println("aNum is " + aNum;
     System.out.println("aDept is " + aDept;
   }
   public void anotherEmpMethod()
   {
     System.out.println("aNum is " + aNum;
     System.out.println("aDept is " + aDept;
   }
}
```

Figure 4-5 Employee2 class with an overriding variable

When the method call is `adminAssistant.empMethod();`, the output will indicate that aNum is 88 and aDept is 55. The empMethod() will use the local aNum valued at 88, but use the class aDept valued at 55. When the method call is `adminAssistant.anotherEmpMethod();`, the output will show that aNum is 44 and aDept is 55; in both cases, the class variables are used because they have not been overridden within anotherEmpMethod(). When you write programs, it is best to avoid confusing situations that arise when you give the same name to a class variable and a method variable. But, if you do use the same name, be aware that within the method, the method variable will override the class variable.

Next you will create a method with several blocks to demonstrate block scope.

To demonstrate block scope:

1. Start your text editor, and then open a new document, if necessary.
2. Type the header for a class named DemoBlock as **`public class DemoBlock`**. On the next three lines, type the opening curly brace (**`{`**), the main() method header, **`public static void main(String[] args)`**, and the main()'s opening curly brace (**`{`**).
3. On a new line that is indented one column, declare an integer by typing:

   ```
   int x = 1111;
   ```

4. On new, indented lines, type the following two println() statements:

   ```
   System.out.println("Demonstrating block scope");
   System.out.println("In first block x is " + x);
   ```

5. Begin a new block by typing an opening curly brace on the next line. Within the new block, declare another integer by typing **`int y = 2222;`**. Within this new block, type the following two statements to display the values of x and y:

```
System.out.println("In second block x is " + x);
System.out.println("In second block y is " + y);
```

6. End the block by typing a closing curly brace (**`}`**). On the next line, begin a new block with an opening curly brace. Within this new block, declare a new integer with the same name as the integer declared in the previous block by typing **`int y = 3333;`**

7. Enter two println() statements, a method call, and two more println() statements, as follows:

```
System.out.println("In third block x is " + x);
System.out.println("In third block y is " + y);
demoMethod();
System.out.println("After method x is " + x);
System.out.println("After method block y is " + y);
```

8. Close this block by typing a closing curly brace.

9. Type **`System.out.println("At the end x is " + x);`**, and then type a closing curly brace. This last statement in the program displays the value of x.

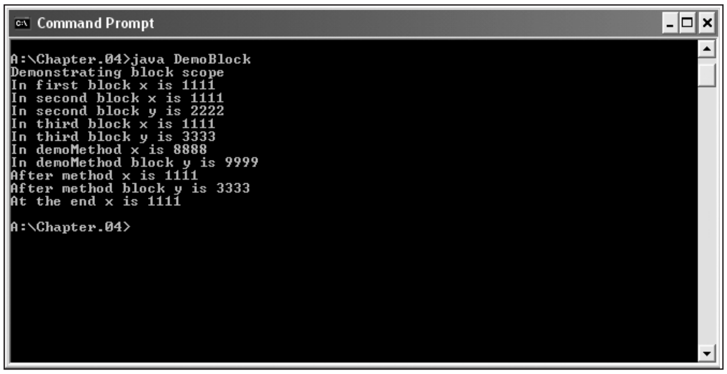
10. Finally, enter the following demoMethod() that creates its own x and y, assigns different values, and then displays them:

```
public static void demoMethod()
{
  int x = 8888, y = 9999;
  System.out.println("In demoMethod x is " + x);
  System.out.println("In demoMethod block y is " + y);
}
```

11. Type the final closing curly brace, and then save the file as **DemoBlock.java** in the Chapter.04 folder on your Student Disk. At the command prompt, compile the file by typing the command **`javac DemoBlock.java`**. If necessary, correct any errors and compile again.

12. Run the program by typing the command **`java DemoBlock`**. Your output should look like Figure 4-6.

It is important to understand the impact that blocks have on your variables. Once you understand the scope of variables, you can more easily locate the source of many errors within your programs.



```
A:\Chapter.04>java DemoBlock
Demonstrating block scope
In first block x is 1111
In second block x is 1111
In second block y is 2222
In third block x is 1111
In third block y is 3333
In demoMethod x is 8888
In demoMethod block y is 9999
After method x is 1111
After method block y is 3333
At the end x is 1111

A:\Chapter.04>
```

Figure 4-6 Output of the DemoBlock program

Tip: To gain a more complete understanding of blocks and scope, change the values of x and y in the program, and try to predict the exact output before resaving, recompiling, and rerunning the program.

Overloading a Method

Overloading involves using one term to indicate diverse meanings, or writing multiple methods with the same name, but with different arguments. When you use the English language, you overload words all the time. When you say "open the door," "open your eyes," and "open a computer file," you are talking about three very different actions using very different methods, and producing very different results. However, anyone who speaks English fluently has no trouble understanding your meaning because the verb *open* is understood in the context of the noun that follows it.

When you overload a Java method, you write multiple methods with a shared name. The compiler understands your meaning based on the arguments you use with the method. For example, suppose you create a class method to apply a simple interest rate to a bank balance. The method receives two double arguments—the balance and the interest rate—and displays the multiplied result. Figure 4-7 shows the method.

```
public static void simpleInterest(double bal, double rate)
{
   double interest;
   interest = bal * rate;
   System.out.println("Interest on " + bal + " at " +
      rate + " interest rate is " + interest);
}
```

Figure 4-7 The simpleInterest() method with two double arguments

Tip

The simpleInterest() method can receive integer arguments even though it is defined as needing double arguments because integers will be promoted or cast automatically to doubles, as you learned in Chapter 1.

When a program calls the simpleInterest() method and passes double values, as in `simpleInterest(1000.00, 0.04)`, the simple interest will be calculated correctly as four percent of $1000.00. Assume, however, that the interest rate passed to the simpleInterest() method comes from inconsistent user input. Some users who want to indicate an interest rate of four percent might type .04; others might type 4 and assume that they are typing four percent. When the simpleInterest() method is called with the arguments $1000.00 and .04, the interest is calculated correctly as 40.00. When the method is called using $1000.00 and 4, the interest is calculated incorrectly as 4000.00.

A solution for the conflicting use of numbers to represent parameter values is to overload the simpleInterest() method. For example, in addition to the simpleInterest() method shown in Figure 4-7, you could add the method shown in Figure 4-8.

```
public static void simpleInterest(double bal, int rate)
// Notice rate type
{
   double interest, rateAsPercent;
   rateAsPercent = rate/100.0;
   // Converts whole number rate to decimal equivalent
   interest = bal * rateAsPercent;
   System.out.println("Interest on " + bal + " at " +
     rate + " interest rate is " + interest);
}
```

Figure 4-8 The simpleInterest() method with a double and an integer argument

Tip

Note the rateAsPercent figure is calculated by dividing by 100.0, and not by 100. If two integers are divided, the result is a truncated integer; dividing by a double 100.0 causes the result to be a double. Alternately, you could use an explicit cast such as `(double)rate/100.00`.

If the method simpleInterest() is called using two double arguments, as in `simpleInterest(1000.00, .04)`, the first simpleInterest() method shown in Figure 4-7 will execute. However, if an integer is used as the second parameter in the call to simpleInterest(), as in `simpleInterest(1000.00, 4)`, then the method shown in Figure 4-8 will execute. The whole number rate figure will be divided by 100.0 correctly before it is used to determine the interest earned.

Of course, you could use methods with different names to solve the dilemma of producing an accurate simple interest figure—for example, simpleInterestRateUsingDouble() and simpleInterestRateUsingInt(). Using this approach requires that you place a decision within your program to determine which of the two methods to call, but it is more convenient to use one method name and then let the interpreter determine which method to use. Also, it is easier to remember one reasonable name for tasks that are functionally identical except for argument types.

Tip

You will learn about placing a decision within your program in Chapter 5.

Next you will overload methods to display event dates for Event Handlers Incorporated. The methods will take one, two, or three integer arguments. If there is one argument, it is the month, and the event is scheduled for the first day of the given month in the year 2003. If there are two arguments, they are the month and the day in the year 2003. Three arguments represent the month, day, and year.

Tip

In addition to creating your own class to store dates, you can use a built-in Java class to handle dates. You will learn about this class later in this chapter.

To overload an overloadDate() method to take one, two, or three arguments:

1. Open a new file in your text editor.
2. Create the following DemoOverload class, with three integer variables and three calls to an overloadDate() method:

```
public class DemoOverload
{
 public static void main(String[] args)
 {
    int month = 6, day = 24, year = 2003;
    overloadDate(month);
    overloadDate(month,day);
    overloadDate(month,day,year);
  }
```

3. Create the following overloadDate() method that requires one argument:

```
public static void overloadDate(int mm)
{
 System.out.println("Event date " + mm + "/1/2003");
}
```

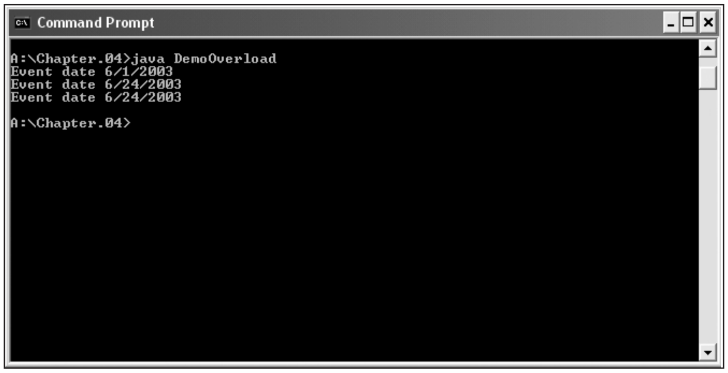
4. Create the following overloadDate() method that requires two arguments:

```
public static void overloadDate(int mm, int dd)
{
 System.out.println("Event date " + mm + "/" +
  dd + "/2003");
}
```

5. Create the following overloadDate() method that requires three arguments:

```
public static void overloadDate(int mm, int dd, int yy)
{
 System.out.println("Event date " + mm + "/" +
  dd + "/" + yy);
}
```

6. Type the closing curly brace for the DemoOverload class.
7. Save the file as **DemoOverload.java** in the Chapter.04 folder on your Student Disk.
8. Compile the program, correct any errors, recompile if necessary, and then execute the program. Figure 4-9 shows the output. Notice that whether you call the overloadDate() method using one, two, or three arguments, the date prints correctly because you have successfully overloaded the overloadDate() method.



```
A:\Chapter.04>java DemoOverload
Event date 6/1/2003
Event date 6/24/2003
Event date 6/24/2003

A:\Chapter.04>
```

Figure 4-9 Output of the DemoOverload program

Learning About Ambiguity

When you overload a method, you run the risk of creating an **ambiguous** situation—one in which the compiler cannot determine which method to use. For example, consider the simple method shown in Figure 4-10.

```
void simpMeth(double d)
{
   System.out.println("Method receives double parameter");
}
```

Figure 4-10 The simpMeth() method with a double argument

If you declare doubleValue as a double variable, and intValue as an int variable, then either method call **`simpMeth(doubleValue);`** or **`simpMeth(intValue);`** results in the output "Method receives double parameter". When you call the method with the double argument, the method works as expected. When you call the method with the integer argument, then the integer is cast as (or promoted to) a double, and the method also works.

Tip: Note that if the method with the declaration `void simpMeth(double d)` did not exist, but the declaration `void simpMeth(int i)` did exist, then the method call `simpMeth(doubleValue);` would fail. Although an integer can be promoted to a double, a double cannot become an integer. This makes sense if you consider the potential loss of information when a double value is reduced to an integer.

If you add a second overloaded simpMeth() method within a program that takes an integer parameter (as shown in Figure 4-11), then the output changes when you call **`simpMeth(intValue);`**. Instead of promoting an integer argument to a double, the compiler recognizes a more exact match for the method call that uses the integer argument, so it calls the version of the method that produces the output "Method receives integer parameter".

```
void simpMeth(int i)
{
   System.out.println("Method receives integer parameter");
}
```

Figure 4-11 The simpMeth() method with an integer argument

A more complicated and potentially ambiguous situation arises when the compiler cannot determine which of several versions of a method to use. Consider the following overloaded simpleInterest() method declarations:

```
public static void simpleInterest(double bal, double rate)
public static void simpleInterest(double bal, int rate)
  // Notice rate type
```

A call to simpleInterest() with two double arguments executes the first version of the method, and a call to simpleInterest() with a double and an integer argument executes the second version of the method. With each of these calls, the compiler can find an exact match for the arguments you send. However, if you call simpleInterest() using two integer arguments, as in `simpleInterest(300,6);`, an ambiguous situation arises because there is no exact match for the method call. Because two integers can be promoted to two doubles (thus matching the first version of the overloaded method), or just one integer can be promoted to a double (thus matching the second version), the compiler does not know which version of the simpleInterest() method to use and the program will not execute. You could argue that *int*, *int* is "closer" to *double*, *int* than it is to *double*, *double*, but the compiler does not make such decisions for you.

Tip

An overloaded method is not ambiguous on its own—it only becomes ambiguous if you create an ambiguous situation. A program containing a potentially ambiguous situation will run problem free if you do not make any ambiguous method calls.

It is important to note that you can overload methods correctly by providing different argument lists for methods with the same name. Methods with identical names that have identical argument lists, but different return types, are not overloaded—they are illegal. For example, `int aMethod(int x)` and `void aMethod(int x)` cannot coexist within a program. The compiler determines which of several versions of a method to call based on argument lists. When the method call `aMethod(17);` is made, the compiler will not know which method to execute because both methods take an integer argument.

Sending Arguments to Constructors

In Chapter 3, you learned that Java automatically provides a constructor method when you create a class. You also learned that you can write your own constructor method, and that you often do so when you want to ensure that fields within classes are initialized to some appropriate default value. Additionally, you can write constructor methods that receive arguments. Such arguments are often used for initialization purposes when the values that you want to assign to objects upon creation might vary.

For example, consider the Employee class with two data fields shown in Figure 4-12. Its constructor method assigns 999 to each potentially instantiated Employee's empNum. Any time an Employee object is created using a statement such as `Employee partTimeWorker = new Employee();`, even if no other data-assigning methods are ever used, you are ensured that the partTimeWorker Employee, like all Employees, will have an initial empNum of 999.

```
public class Employee
{
   private int empNum;
   private double empSalary;
   // Constructor method
   Employee()
   {
      empNum = 999
   }
   // Other methods go here
}
```

Figure 4-12 Employee class

Tip: You can use a setEmpNum() method to assign values to individual Employee objects after construction, but a constructor method assigns the values at the time of creation.

Alternately, you might choose to create Employees with initial empNums that differ for each Employee. To accomplish this within a constructor, you need to pass an employee number to the constructor. Figure 4-13 shows an Employee constructor that receives an argument. With this constructor, an argument is passed using a statement, such as **`Employee partTimeWorker = new Employee(881);`**. When the constructor executes, the integer within the method call is passed to Employee() and assigned to the empNum.

```
Employee(int num)
{
   empNum = num;
}
```

Figure 4-13 Employee constructor method with an argument

To demonstrate a constructor with an argument, you will use a new commented version of the EventSite5 class you created in Chapter 3.

To alter a constructor:

1. Open a new file in your text editor, and then enter the EventSite6 class shown in Figure 4-14. This file is similar to the EventSite5.java text file you created in Chapter 3, but comments have been added for clarity. Save the file as **EventSite6.java** in the Chapter.04 folder on your Student Disk.

```
public class EventSite6
{
  private int siteNumber;
  private double usageFee;
  private String managerName;
  //Constructor
  EventSite
  {
    siteNumber = 999;
    managerName = "ZZZ";
  }
  //getManagerName() gets managerName
  public String getManagerName()
  {
    return managerName;
  }
  //getSiteNumber() gets the siteNumber
  public int getSiteNumber()
  {
    return siteNumber;
  }
  //getUsageFee() gets the usageFee
  public double getUsageFee()
  {
    return usageFee;
  }
  //setManagerName() assigns a name to the manager
  public void setManagerName(String name)
  {
    managerName = name;
  }
  //setSiteNumber() assigns a siteNumber
  public void setSiteNumber(int n)
  {
    siteNumber = n;
  }
  //setUsageFee() assigns a value to the usage fee
  public void setUsageFee(double amt)
  {
    usageFee = amt;
  }
}
```

Figure 4-14 EventSite6.java class

2. Modify the existing constructor by typing the following code so that the constructor takes an argument for the site number and then assigns the argument value to the siteNumber field:

```
EventSite6(int siteNum)
{
  siteNumber = siteNum;
  managerName = "ZZZ";
}
```

3. Save the file and then compile and correct any errors.
4. Open a new text file to create a short program to demonstrate the constructor at work by typing the following code:

```
public class DemoConstruct
{
 public static void main(String[] args)
 {
  EventSite6 aSite = new EventSite6(678);
  System.out.println("Site number is "
    + aSite.getSiteNumber());
  }
}
```

5. Save the file as **DemoConstruct.java** in the Chapter.04 folder, and then compile and test the program. The site number (678) should be assigned to the aSite object.

OVERLOADING CONSTRUCTORS

If you create a class from which you instantiate objects, Java automatically provides you with a constructor. Unfortunately, if you create your own constructor, the automatically created constructor no longer exists. Therefore, once you create a constructor that takes an argument, you no longer have the option of using the automatic constructor that requires no arguments.

Fortunately, as with any other method, you can overload constructors. Overloading constructors provides you with a way to create objects with or without initial arguments, as needed. For example, in addition to using the provided constructor method shown in Figure 4-14, you can create the second constructor method for the Employee class shown in Figure 4-15. When both constructors reside within the Employee class, you have the option of creating an Employee object, either with or without an initial empNum value. When you create an Employee object with `Employee aWorker = new Employee();`, the constructor with no arguments is called and the Employee object receives an initial empNum value of 999. When you create an Employee object with `Employee anotherWorker = new Employee(7677);`, the constructor that requires an integer is used, and the anotherWorker Employee receives an initial empNum of 7677.

```
Employee()
{
   empNum = 999;
}
```

Figure 4-15 Employee constructor method with no argument

Similarly, if you want to pass values to initialize field values for siteNumber and managerName, you can create a constructor that requires two arguments. You can use the arguments to initialize field values, but you can also use arguments for any other purpose. For example, you could use the presence or absence of an argument simply to determine which of two possible constructors to call, yet not make use of the argument within the constructor method. As long as the constructor argument lists differ, there is no ambiguity in which constructor method to call.

Next you will overload the EventSite6 constructor to take either no arguments, in which case the site number is 999, or to take an argument that is the site number.

To overload the EventSite6 constructor:

1. In your text editor, open the **EventSite6.java** text file from the Chapter.04 folder on your Student Disk, and save it as **EventSite7.java**. Change the class name to **`EventSite7`**.
2. Position the insertion point at the end of the comment `// Constructor`, type **`s`** to make the word *Constructors*, and then press **[Enter]** to start a new line.
3. Above the existing constructor that requires an argument, add the new overloaded constructor that requires no argument by typing the following:

   ```
   EventSite7()
   {
    siteNumber = 999;
    managerName = "ZZZ";
   }
   ```
4. Rename the old EventSite6 constructor to **EventSite7**.
5. Save the file, compile, and correct any errors.
6. In your text editor, open the **DemoConstruct.java** file from the Chapter.04 folder on your Student Disk, and change the class name to **DemoConstruct2**.
7. Change the statement `EventSite6 aSite =  new EventSite6(678);`, to **`EventSite7 aSite = new EventSite7(678);`**, position the insertion point at the end of this statement and then press **[Enter]** to start a new line.
8. Create a new EventSite7 with no constructor argument by typing **`EventSite7 anotherSite = new EventSite7();`**
9. Position the insertion point after the println() statement that displays the site number of `aSite, System.out.println("Site number is " +  aSite.getSiteNumber());`, and then press **[Enter]** to start a new line. Then type the following statement to print the site number of anotherSite:

   ```
   System.out.println("Another site number is "
      + anotherSite.getSiteNumber());
   ```

10. Save the program, compile, and test the program. The two site numbers should print as 678 and 999.
11. Close your text editor.

LEARNING ABOUT THE `this` REFERENCE

When you start creating classes from objects that you instantiate, the classes can become large very quickly. Besides data fields, each class can have many methods, including several overloaded versions. If you instantiate many objects of a class, the computer memory requirements can become substantial. Fortunately, it is not necessary to store a separate copy of each variable and method for each instantiation of a class.

Usually you want each instantiation of a class to have its own data fields. If an Employee class contains fields for employee number, name, and salary, every individual Employee object will need a unique number, name, and salary values. However, when you create a method for the Employee class, any Employee object can use the same method. Whether the method performs a calculation, sets a field value, or constructs an object, the instructions are the same for each instantiated object. Therefore, you store just one copy of a method that all instantiated objects use.

When you use an object method, you use the object name, a dot, and the method name—for example, **`aWorker.getEmpNum();`**. When you refer to the aWorker.getEmpNum() method, you are referring to the general, shared Employee class getEmpNum() method; aWorker has access to the method because aWorker is a member of the Employee class. However, within the getEmpNum() method, when you access the empNum *field*, you access aWorker's private, individual copy of the empNum field. Because many Employees might exist, but just one copy of the method exists no matter how many Employees there are, when you call **`aWorker.getEmpNum();`**, the compiler must determine *whose* copy of empNum should be returned by the single getEmpNum() method.

The compiler accesses the correct object's field because you implicitly pass to the getEmpNum method a reference to aWorker. This reference is called the **`this`** reference and is a reserved word in Java. For example, the two getEmpNum() methods shown in Figure 4-16 perform identically. The first method simply uses the **`this`** reference without you being aware of it; the second method uses the **`this`** reference explicitly.

Tip

When you pass a reference, you pass a memory address.

Usually you neither want nor need to refer to the **`this`** reference within the methods you write, but the **`this`** reference is always there, working behind the scenes, so that the data field for the correct object can be accessed.

```
public void getEmpNum()
{
  return empNum;
}
public void getEmpNum()
{
  return this.empNum;
}
```

Figure 4-16 The getEmpNum() methods with implicit and explicit `this` references

Tip: Recall that methods associated with individual objects are instance methods.

In Chapter 3, you learned that most methods you create within a class are nonstatic—methods that you associate with individual objects. You also created static methods. For example, the main() method in a program and the method's main() calls without an object reference are static. These methods do not have a `this` reference because they have no object associated with them; therefore, they are called **class methods**.

You can also create **class variables**, which are variables that are shared by every instantiation of a class. For example, you might have a company ID number that is the same for all Employee objects. You can add a static class variable to the class definition, as shown in Figure 4-17. Also shown in figure 4-17 is a simple method to display the employee number along with the employee's COMPANY_ID.

```
public class Employee
{
  static private int COMPANY_ID = 12345;
  private int empNum;
  private double empSalary;
  Employee(int num)
  // Constructor requiring employee number
    empNum = num;
  }
  public void showCompanyID()
  {
    System.out.println("Worker " + empNum
      + " has company ID " + COMPANY_ID);
  }
  // Other class methods can go here
}
```

Figure 4-17 Employee class with a static ID number field

No matter how many Employee objects are eventually instantiated, each will refer to the single COMPANY_ID field. For example, if two Employees are created with `Employee firstWorker = new Employee(444);` and `Employee secondWorker = new Employee(777);`, when you write the statement `firstWorker.showCompanyID()`, its output is `Worker 444 has company ID_ 12345`, and when you write `secondWorker.showCompanyID();`, the statement's output is `Worker 777 has COMPANY_ID 12345`. The different workers have individual IDs, but the same company ID.

Additionally, if you change the value of COMPANY_ID in the Employee class, the value changes for all class instantiations. Therefore, besides values such as a company ID, good candidates for static class variables are fields such as a legal minimum wage or a maximum number of hours that an employee is allowed to work in a single week. When such values change for one employee, they change uniformly for all employees.

WORKING WITH CONSTANTS

In Chapter 2, you learned to create literal constants within a program. A literal constant is a fixed value that does not change, such as the literal string "First Java program." Variables, on the other hand, do change. When you declare `int empNum;`, you expect that the value stored in empNum will be different at different times or for different employees.

Sometimes, however, a variable or data field should be constant; that is, it should not be changed during the execution of a program. This is known as a **constant variable**. While the concept of a constant variable is somewhat of an oxymoron, there are situations where using a constant variable is reasonable. For example, the value for a company ID is fixed, so you do not want any methods to alter the company ID value while a program is running. To prevent alteration, insert the keyword `final` in the company ID declaration. Then the name COMPANY_ID becomes a **symbolic constant**, which indicates that when you compile any program that uses an object that contains the COMPANY_ID, the field has a `final`, unalterable value. By convention, constant fields are written using all uppercase letters. The compiler does not require using uppercase identifiers for constants, but using uppercase identifiers helps you distinguish symbolic constants from variables. For readability, you can insert underscores between words in symbolic constants.

Tip

Mathematical constants are good candidates for receiving final status. For example, when PI is defined as `static final double PI = 3.14159;`, it appropriately becomes a constant that should never take on any other value. A fixed sales tax rate `static final double SALES_TAX = 0.075;` remains fixed for every use within a program.

Tip: You can use the keyword `final` with methods or classes. When used in this manner, `final` indicates limitations placed on inheritance. You will learn more about inheritance as you become more proficient at object-oriented programming.

You cannot change the value of a symbolic constant after declaring it; any attempt to do so will result in a compiler error. You must initialize a constant with a value; this makes sense when you consider that a constant cannot be changed later. If a constant does not receive a value upon creation, it can never receive a value. Figure 4-18 shows a typical declaration of a constant.

```
public class Employee
{
   static final private int COMPANY_ID = 12345;
   // Rest of class goes here
```

Figure 4-18 Employee class with the symbolic constant COMPANY_ID

A constant always has the same value within a program, so you might wonder why you cannot use the actual, literal value. For example, why not code 12345 when you need the company ID rather than going to the trouble of creating the COMPANY_ID symbolic constant? There are at least three good reasons to use the symbolic constant rather than the literal one:

- The number 12345 is more easily recognized as the company ID if it is associated with an identifier such as COMPANY_ID.
- If the company ID changes, you would change the value of COMPANY_ID at one location within your program—where the constant is defined—rather than searching for every use of 12345 to change it to a different number. Also, being able to make the change at one location saves you valuable programming time.
- Even if you are willing to search for every instance of 12345 in a program to change it to the new company ID value, you might inadvertently change the value to one that is being used for something else, such as an employee's employee number or salary.

Next you will create a class variable to hold the location of the company headquarters for Event Handlers Incorporated. The location of the company headquarters is an ideal candidate for a class variable. Because the headquarters location is the same for every event no matter where the actual event is held, the value for the headquarters location should be stored just once, but every EventSite7 object should have access to the information.

To create a class variable for the EventSite7 class:

1. In your text editor, open the **EventSite7.java** text file from the Chapter.04 folder on your Student Disk, and then change the class name to **EventSite8**. Save the file as **EventSite8. java**.
2. Position the insertion point after the opening curly brace of the class, and then press **[Enter]** to start a new line.
3. Type the class variable:

   `static final public String HEADQUARTERS = "Crystal Lake, IL";`

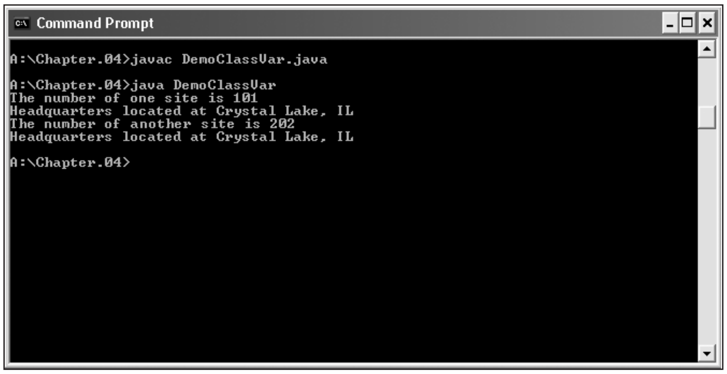
Tip: A static variable can be either `public` or `private`. If the variable is `private`, then you must write a method in your class to access it.

4. Change the `EventSite7()` constructor to **`EventSite8()`**, and then change the `EventSite7(int siteNum)` constructor to **`EventSite8(int siteNum)`**.
5. Save the file and compile.
6. Start a new file in your text editor, and then create the demonstration program named **DemoClassVar** shown in Figure 4-19. This program shows the headquarters location is the same for all EventSites.

```
public class DemoClassVar
{
   public static void main(String[] args)
   {
      EventSite8 oneSite = new EventSite8();
      EventSite8 anotherSite = new EventSite8();
      oneSite.setSiteNumber(101);
      anotherSite.setSiteNumber(202);
      System.out.print("The number of one site is ");
      System.out.println(oneSite.getSiteNumber());
      System.out.print("Headquarters located at ");
      System.out.println(oneSite.HEADQUARTERS);
      System.out.print("The number of another site is ");
      System.out.println(anotherSite.getSiteNumber());
      System.out.print("Headquarters located at ");
      System.out.println(anotherSite.HEADQUARTERS);
   }
}
```

Figure 4-19 DemoClassVar program

7. Save the file as **DemoClassVar.java** in the Chapter.04 folder. Compile and test the program. Figure 4-20 shows the program's output.



```
A:\Chapter.04>javac DemoClassVar.java

A:\Chapter.04>java DemoClassVar
The number of one site is 101
Headquarters located at Crystal Lake, IL
The number of another site is 202
Headquarters located at Crystal Lake, IL

A:\Chapter.04>
```

Figure 4-20 Output of the DemoClassVar program

Using Automatically Imported, Prewritten Constants and Methods

There are many times when you need to create classes from which you will instantiate objects. You can create an Employee class with fields appropriate for describing employees and their functions, and an Inventory class with fields appropriate for whatever type of item it is that you manufacture. There are, however, many classes that a wide variety of programmers need. Rather than having each Java programmer "reinvent the wheel," the creators of Java created nearly 500 classes for you to use in your programs.

You already used several of the prewritten classes without being aware of it. System, Character, Boolean, Byte, Short, Integer, Long, Float, and Double are actually classes from which you can create objects. These classes are stored in a **package**, or a **library of classes**, which is simply a folder that provides a convenient grouping for classes. There are many Java packages containing classes that are available only if you explicitly name them within your program, but the group of classes that contains the previously listed classes is used so frequently that it is available automatically to every program you write. The package that is implicitly imported into every Java program is named **java.lang**. The classes it contains are the **fundamental classes**, or basic classes, as opposed to the **optional classes** that must be explicitly named.

Tip: You will begin to import optional classes explicitly later in this chapter.

The class java.lang.Math contains constants and methods that you can use to perform common mathematical functions. All of the constants and methods in the Math class are `static`—they are class variables and class methods. A commonly used constant is PI.

Within the Math class, the declaration for PI is `public final static double PI = 3.14159265358979323846;`. Notice that PI is:

- `public`, so any program can access it directly
- `final`, so it cannot be changed
- `static`, so only one copy exists
- `double`, so it holds a large floating-point value

Tip

In geometry, PI is an approximation of the value of the ratio of the circumference of a circle to its diameter.

Tip

Another useful constant is E, which represents the base of natural logarithms. Its definition is `public final static double E = 2.7182818284590452354;`.

You can use the value of PI within any program you write by referencing the full package path in which PI is defined; for example `areaOfCircle = java.lang.Math.PI * radius * radius;`. However, the Math class is imported automatically into your programs, so if you simply reference `Math.PI`, Java will recognize this code as a shortcut to the full package path. Therefore, the preferred (and simpler) statement is `areaOfCircle = Math.PI * radius * radius;`.

In addition to constants, there are many useful methods available within the Math class. For example, the Math.max() method returns the larger of two values, and the method Math.abs() returns the absolute value of a number. The statement `largerValue = Math.max(32, 75)` results in largerValue assuming the value 75, and the statement `posVal = Math.abs(-245);` results in posVal assuming the value 245. Table 4-1 lists some common Math class methods.

Table 4-1 Common Math class methods

Method	Meaning
abs(x)	Absolute value of *x*
acos(x)	Arccosine of *x*
asin(x)	Arcsine of *x*
atan(x)	Arctangent of *x*
atan2(x,y)	Theta component of the polar coordinate (r,theta) that corresponds to the Cartesian coordinate x,y
ceil(x)	Smallest integral value not less than *x* (ceiling)

Table 4-1 Common Math class methods (continued)

Method	Meaning
cos(x)	Cosine of *x*
exp(x)	Exponent, where *e* is the base of the natural logarithms
floor(x)	Largest integral value not greater than *x*
log(x)	Natural logarithm of *x*
max(x,y)	Larger of *x* and *y*
min(x,y)	Smaller of *x* and *y*
pow(x,y)	*x* raised to the *y* power
random()	Random double number between 0.0 and 1.0
rint(x)	Closest integer to *x* (*x* is a double, and the return value is expressed as a double)
round(x)	Closest integer to *x* (where *x* is a float or double, and the return value is an integer or long)
sin(x)	Sine of *x*
sqrt(x)	Square root of *x*
tan(x)	Tangent of *x*

Tip

Because all constants and methods in the Math class are classwide, there is no need to create an instance. You cannot instantiate objects of type Math because the constructor for the Math class is private and your programs cannot access the constructor.

Unless you are a mathematician, you won't use many of these Math class methods, and it is unwise to do so unless you understand their purposes. For example, because it is illegal to take the square root of a negative number, the method call `imaginaryNumber = Math.sqrt(-12);` causes a compiler error and does not execute.

Next you will use the Math class to perform some basic calculations.

To write a program that uses some Math class methods:

1. Open a new file in your text editor. Type the DemoMath class header **`public class DemoMath`**. On a new line, type the opening curly brace for the class, and then press **[Enter]**.
2. Type the main() method header **`public static void main(String[] args)`**, press **[Enter]**, type the opening curly brace for the main() method, and then press **[Enter]** to create a new line.
3. Create a double variable named val by typing **`double val = 26.9;`**, and then press **[Enter]**.

4. Type the following statement that displays the value on the screen:
   `System.out.println("The value is " + val);`

5. On separate lines, type the following statements to demonstrate the Math class methods:

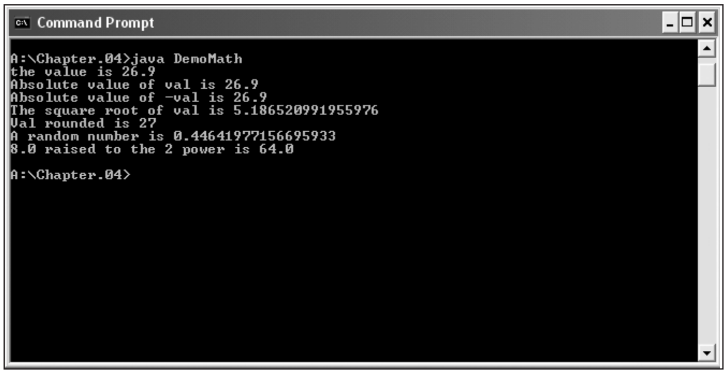
```
System.out.print("Absolute value of val is ");
System.out.println(Math.abs(val));
System.out.print("Absolute value of -val is ");
System.out.println(Math.abs(-val));
System.out.print("The square root of val is ");
System.out.println(Math.sqrt(val));
System.out.print("Val rounded is ");
System.out.println(Math.round(val));
System.out.print("A random number is ");
System.out.println(Math.random());
System.out.print("8.0 raised to the 2 power is ");
System.out.println(Math.pow(8.0, 2));
```

Tip

The expression -val means "negative val." The minus sign (-) used in this manner is a unary or single-argument operator. You will learn more about unary operators in Chapter 5.

6. Add closing curly braces for the main() method and for the class.

7. Save the program as **DemoMath.java** in the Chapter.04 folder on your Student Disk, compile the program, run it, and then compare your results to Figure 4-21.



```
A:\Chapter.04>java DemoMath
the value is 26.9
Absolute value of val is 26.9
Absolute value of -val is 26.9
The square root of val is 5.186520991955976
Val rounded is 27
A random number is 0.44641977156695933
8.0 raised to the 2 power is 64.0

A:\Chapter.04>
```

Figure 4-21 Output of the DemoMath program

8. Add additional statements that demonstrate any of the other Math methods that you might use in your programs. Save, compile, and test the program again.

Using Prewritten Imported Methods

Java contains hundreds of classes, only a few of which—such as java.lang—are included automatically in the programs you write. To use any of the other prewritten classes, you must use one of three methods:

- Use the entire path with the class name.
- Import the class.
- Import the package which contains the class you are using.

For example, the java.util class package contains useful methods that deal with dates and times. Within this package, a class named Date is defined. You can instantiate an object of type Date from this class by using the full class path, as in `java.util.Date myAnniversary = new java.util.Date();`. Alternately, you can shorten the declaration of myAnniversary to `Date myAnniversary = new Date();` by including `import java.util.Date;` as the first line in your program. An import statement allows you to abbreviate lengthy class names by notifying the Java program that when you use Date, you mean the java.util.Date class. You must place any import statement you use before any executing statement in your program. That is, you can have a blank line or a comment line—but nothing else—prior to an import statement.

Tip

Date is not a reserved word; it is a class you are importing. If you do not want to import the java utility's Date class, you are free to write your own Date class.

An alternative to importing a class is to import an entire package of classes. You can use the asterisk (*) as a **wildcard symbol** to represent all the classes in a package. Therefore, the import statement `import java.util.*;` imports the Date class and any other java.util classes as well. There is no disadvantage to importing the extra classes, and you will most commonly see the wildcard method in professionally written Java programs.

Tip

You cannot use the Java language wildcard exactly like a DOS or UNIX wildcard because you cannot import all the Java classes with `import java.*;`. The Java wildcard works only with specific packages such as `import java.util.*;` or `import java.lang.*;`.

Tip

Notice that the import statement ends with a semicolon. The import statement does not move the entire imported class or package into your program as its name implies. Rather, it simply notifies the program that you will be using the data and method names that are part of the imported class or package.

The Date class has several constructors. For example, if you construct a Date object with five integer arguments, they become the year, month, day, hour, and minutes. A Date object constructed with three integer arguments assumes the arguments to be the year, month, and day, and the time is set to midnight. The constructor that takes no argument assigns the current moment to a Date object. The current moment is the number of milliseconds that have elapsed since midnight, January 1, 1970. Therefore, the statement `Date myAnniversary = new Date()` assigns a value that is a very large 12- or 13-digit number to the myAnniversary variable. You can retrieve this number with a method named getTime(). The statement `System.out.println("Milliseconds since  1/1/70 are " + myAnniversary.getTime());` results in the output `Milliseconds since 1/1/70 are 1066233611927` when the program is run at midnight on October 15, 2003.

Tip: If you set the hours in a Date object, a 24-hour clock is assumed—for example, 13 is 1 P.M.

Although it is interesting, the number of milliseconds elapsed since 1970 is not a useful piece of information for most people. Fortunately, the Date class does contain a variety of methods such as setMonth(), getMonth(), setDay(), getDay(), setYear(), and getYear(), which supply more-useful information. The program shown in Figure 4-22 shows the values of two dates being set and retrieved.

```
import java.util.*;
public class DemoDate
{
   public static void main(String[] args)
   {
      Date toDay = new Date();
      Date birthDay = new Date(82,6,14);
      System.out.println(toDay);
      System.out.print("Current month is ");
      System.out.println(toDay.getMonth());
      System.out.print("Current day is ");
      System.out.println(toDay.getDate());
      System.out.print("Current year is ");
      System.out.println(toDay.getYear());
      System.out.print("Birth month is ");
      System.out.println( birthDay.getMonth());
      System.out.print("Birth day is ");
      System.out.println(birthDay.getDate());
      System.out.print("Birth year is ");
      System.out.println(birthDay.getYear());
   }
}
```

Figure 4-22 DemoDate program

You can perform arithmetic using dates. For example, if toDay is declared to hold today's date with `Date toDay = new Date();`, then you can use the following code to find out the due date of a bank certificate that matures in 180 days by adding 180 to the day part of the Date object:

```
toDay.setDate(toDay.getDate() + 180);
System.out.println("In 180 days it will be " + toDay);
```

Tip The compiler will interpret an incorrect date, such as March 32, as being April 1. This makes many calculations with dates easier. For example, if you bill a customer on August 30 and allow 10 days for payment, you can add 10 to the billing day, and the compiler will understand August 40 to be September 9.

Tip For information about time, including how leap years and leap seconds are calculated, go to the U.S. Naval Observatory Web site at *http://tycho.usno.navy.mil*.

Any year that you use with these Date class methods is a value that is 1900 less than the actual year. For example, 82 means 1982 and 105 means 2005. The month is a value from 0 through 11; January is 0, February is 1, and so on. You must be aware of this value organization when analyzing the meaning of a date.

Next you will use the Date class by declaring some Date variables and keeping track of the length of time it takes for the program to run.

To write a program that uses the Date class:

1. Open a new file in your text editor.
2. For the first line in the file, type **`import java.util.*;`**, press **[Enter]**, and then indent the line two spaces.
3. Begin a DemoDate2 class with the header **`public class DemoDate2`**. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
4. On the new line, indent two more spaces, and then type the following main() class header: **`public static void main(String[] args)`**. On a new line, enter the opening curly brace for the main() method, and then press **[Enter]**.
5. Declare a variable named startTime, and then assign it the current time by typing **`Date startTime = new Date();`**.
6. Declare another variable to hold the day your Java programming class began, for example, **`Date classStart = new Date(103,7,25);`** (where 103,7,25 in this example is August 25, 2003). Don't forget that the current year is 1900 less than the actual year and that the months are numbered 0 through 11.

7. Display the current date and the class start date by typing the following:

```
System.out.println("The current date is " + startTime);
System.out.println("The class started on " + classStart);
```

8. Save the file as **DemoDate2.java** in the Chapter.04 folder on your Student Disk.

Now enter a statement to print the time it takes to run this program. You will create a new endTime object that will hold the current date and time of its creation. Depending on the speed of the computer processor you are using, this time should be a few hundred milliseconds later than it was when the program started. The calculation involves using the getTime() method for the endTime and startTime objects and displaying the difference between the two values.

To use the getTime() method:

1. Open **DemoDate2.java**, if necessary, and then change the class name to **DemoDate3.java**.
2. Position the insertion point after the last print statement, press **[Enter]**, and then type the following code to include the getTime() method:

```
Date endTime = new Date();
System.out.print("Time elapsed is ");
System.out.print(endTime.getTime() -
 startTime.getTime());
System.out.println(" milliseconds");
```

3. Add the closing curly brace for the main() method as well as the closing curly brace for the program.
4. Save the program as **DemoDate3.java**, and then compile and test the program.

When you compile the DemoDate3.java program, you might receive the following error from the compiler: `DemoDate3.java uses or overrides a deprecated API. Recompile with "-deprecation" for details. 1 warning.` This warning indicates that your program compiled successfully. A deprecated API simply indicates that your program uses something that has been improved in subsequent versions of Java. If you want to see information about the methods that are deprecated, then you can recompile the DemoDate3 program using `javac -deprecation DemoDate3.java`.

5. Add some extra println() statements to the program, and save, compile, and run the program again. Does the program take longer to execute?

Learning About Gregorian Calendars

The preceding Help text indicates that Date is a deprecated API. Most of the methods in the Date class have been replaced by the GregorianCalendar class to support the Gregorian calendar, the calendar used in most of the western world. There are seven constructors for GregorianCalendar objects. The default creates a calendar with the current date and time in the default locale. You can use other constructors to specify the year, month, day, hour, minute, and second. You create a calendar object with the default constructor `GregorianCalendar calendar = new GregorianCalendar();`. To calculate time in milliseconds, you can use the getTimeInMillis() method, as in `calendar.getTimeInMillis()`.

Information such as the day, month, and year can be retrieved from a GregorianCalendar object by using a class get() method, and then specifying what you want as an argument. All values returned are of type int. For example, you could get the day of the year with the statement `int dayOfYear = calendar.get(calendar.DAY_OF_YEAR);`. Some of the possible arguments to the get() method are shown in Table 4-2.

Table 4-2 Some possible returns from the GregorianCalendar get() method

Arguments	Values returned by get()
DAY_OF_YEAR	A value of 1 to 366
DAY_OF_MONTH	A value from 1 to 31
DAY_OF_WEEK	SUNDAY, MONDAY, …, SATURDAY, corresponding to values of 1 to 7
YEAR	The current year, for example, 2003
MONTH	JANUARY, FEBRUARY, …, DECEMBER, corresponding to values of 0 to 11
HOUR	A value of 1 to 12 being the current hour in the A.M. or P.M.
AM_PM	A.M. or P.M., which correspond to values of 0 to 1
HOUR_OF_DAY	A value of 0 to 23
MINUTE	The current minute in the current hour, a value of 0 to 59
SECOND	The second in the current minute, a value of 0 to 59
MILLISECOND	The millisecond in the current second, a value of 0 to 999

Next you will construct a program using the GregorianCalendar class and using some of the arguments to the GregorianCalendar get() method.

To write a program that uses the GregorianCalendar class:

1. Open a new file in your text editor.
2. For the first line in the file, type **`import java.util.*;`**, press **[Enter]**, and then indent the line two spaces.
3. Begin by typing a header class **`public class Birthdate`**. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.

4. On the new line, indent two more spaces, and then type the following main class header: **public static void main(String[] args)**. Type the opening curly brace for the main() method, and then press **[Enter]**.
5. Declare integer values to hold a birthdate year, month, and day. Place each on a separate line by pressing **[Enter]** after each integer declaration:

```
int ayear = 1940;
int amonth = 0;//month is 0 to 11
int aday = 31;
```

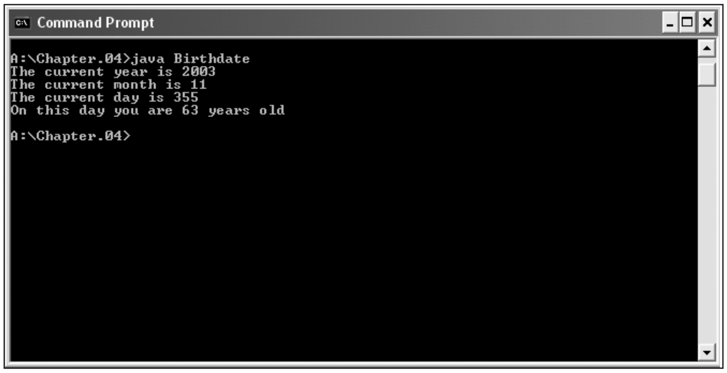
6. Create a new GregorianCalendar object acalendar as **GregorianCalendar acalendar = new GregorianCalendar(ayear,amonth,aday);**.
7. Press **[Enter]** and create a second new GregorianCalendar bcalendar as **`GregorianCalendar bcalendar = new GregorianCalendar();`**.
8. Press **[Enter]**, and then create three separate calls to the get() method, storing each value returned in a separate integer variable byear, bmonth, and bday:

```
int byear = bcalendar.get(bcalendar.YEAR);
int bmonth = bcalendar.get(bcalendar.MONTH);
int bday = bcalendar.get(bcalendar.DAY_OF_WEEK);
```

9. Display the current year, month, and day by typing the following:

```
System.out.println("The current year is " + byear);
System.out.println("The current month is " + bmonth);
System.out.println("The current day is " + bday);
System.out.println("On this day you are " +(byear -
 ayear)+ " years old");
```

10. Finally, put the closing curly braces on separate lines to end the main() method and the Birthday class.
11. Save the file as **Birthdate.java** in the Chapter.04 folder on your Student Disk. Compile and run the program. The output is shown in Figure 4-23. Note that your results may differ from the output shown in Figure 4-23.



```
A:\Chapter.04>java Birthdate
The current year is 2003
The current month is 11
The current day is 355
On this day you are 63 years old

A:\Chapter.04>
```

Figure 4-23 Output of the Birthdate program

Chapter Summary

- A variable's scope is the portion of a program within which you can reference that variable. A variable comes into scope (comes into existence) when you declare it, and goes out of scope (ceases to exist) at the end of the block in which it is declared.
- A block is the code between a pair of curly braces. You can nest blocks within other blocks. Within a method, you can declare a variable with the same name multiple times as long as each declaration is in its own, nonoverlapping block. Within nested blocks, you cannot declare the same variable name more than once. If you declare a variable within a class and use the same variable name within a method of the class, then the variable used inside the method takes precedence, or overrides, the first variable.
- Overloading involves writing multiple methods with the same name but different argument lists. Methods that have identical argument lists but different return types are not overloaded; they are illegal.
- Constructor methods can receive arguments and be overloaded. If you explicitly create a constructor for a class, the automatically created constructor no longer exists.
- You store just one copy of a method for use with each object. You store separate copies of data fields for each object. The compiler accesses the correct object's data fields because you implicitly pass a `this` reference to class methods. Static methods do not have a `this` reference because they have no object associated with them. Static methods are also called class methods.
- Static class variables are those variables that are shared by every instantiation of a class.
- After a program is compiled, literal constants never change. Also, the values stored in symbolic constants never change. You create a symbolic constant by inserting the keyword `final` before a variable name. By convention, constant fields are written using all uppercase letters. A constant must be initialized with a value.
- Java contains nearly 500 prewritten classes which are stored in a package, which is simply a folder that provides a convenient grouping for classes. The package that is implicitly imported into every Java program is named java.lang. The classes it contains are the fundamental classes, as opposed to the optional classes, which must be explicitly named.
- The class java.lang.Math contains constants and methods that can be used to perform common mathematical functions. All of the constants and methods in the Math class are static—they are class variables and class methods. Common useful Math class methods include those used for finding an absolute value, taking a square root, and rounding. To use a prewritten class other than java.lang, you must use the entire path with the class name, import the class, or import the package that contains the class.

- An import statement allows you to abbreviate lengthy class names by notifying the Java program that when you use class names you are referring to those within the imported class. Any import statement you use must be placed before any executing statement in your program. An alternative to importing a class is to import an entire package of classes. To do so, you can use the asterisk (*) as a wildcard symbol to represent all the classes in a package.
- The Date class has several constructors: one that takes no argument and assigns the current moment to a Date object; and others that take the date, or the date and time. The current moment is the number of milliseconds that have elapsed since midnight, January 1, 1970. You can retrieve this number with the getTime() method. The Date class contains a variety of other methods, such as setMonth(), getMonth(), setDay(), getDay(), setYear(), and getYear(), which supply more-useful information.
- The GregorianCalendar class is the calendar generally used in the western world. The GregorianCalandar class has seven constructors and a number of get() methods to define and manipulate dates and time.

Review Questions

1. The code between a pair of curly braces in a method is a ________________.
   a. function
   b. block
   c. brick
   d. sector
2. When a block exists within another block, the blocks are ________________.
   a. structured
   b. nested
   c. sheltered
   d. illegal
3. The portion of a program within which you can reference a variable is the variable's ________________.
   a. range
   b. space
   c. domain
   d. scope

4. You can declare a variable with the same name multiple times ________________.
   a. within a statement
   b. within a block
   c. within a method
   d. never
5. If you declare a variable within a class, and declare and use the same variable name within a method of the class, ________________.
   a. the variable used inside the method takes precedence
   b. the class variable takes precedence
   c. they become the same variable with the same memory address
   d. an error will occur
6. A method variable will ________________ a class variable with the same name.
   a. acquiesce to
   b. destroy
   c. override
   d. alter
7. Overloaded methods must have the same ________________.
   a. name
   b. number of arguments
   c. argument names
   d. type of argument
8. If a method is written to receive a double argument, and you pass an integer to the method, then the method will ________________.
   a. work correctly; the integer will be promoted to a double
   b. work correctly; the integer will remain an integer
   c. execute; but any output will be incorrect
   d. not work; an error message will be issued
9. A constructor ________________ arguments.
   a. can receive
   b. cannot receive
   c. must receive
   d. can receive a maximum of 10
10. A constructor ________________ overloaded.
   a. can be
   b. cannot be

c. must be

d. is always automatically

11. Usually you want each instantiation of a class to have its own copy of ________________.

a. the data fields

b. the class methods

c. both of the above

d. none of the above

12. If you create a class and instantiate two objects, you usually store ________________ for use with the objects.

a. one copy of each method

b. two copies of the same method

c. two different methods

d. data only, not methods

13. The `this` reference ________________.

a. can be used implicitly

b. must be used implicitly

c. must not be used implicitly

d. must not be used

14. Methods that you associate with individual objects are ________________.

a. `private`

b. `public`

c. `static`

d. `nonstatic`

15. Variables that are shared by every instantiation of a class are ________________.

a. class variables

b. `private` variables

c. `public` variables

d. illegal

16. The keyword `final` in a variable declaration indicates ________________.

a. the end of the program

b. a static field

c. a symbolic constant

d. that no more variables will be declared in the program

17. Java classes are stored in a folder or ________________.
    a. packet
    b. package
    c. bundle
    d. gaggle
18. Which of the following statements determines the square root of a number and assigns it to the variable s?
    a. `s = sqrt(number);`
    b. `s = Math.sqrt(number);`
    c. `number = sqrt(s);`
    d. `number = Math.sqrt(s);`
19. A GregorianCalandar object can be created with one of ________________ constructors.
    a. two
    b. four
    c. seven
    d. nine
20. The get() method using the DAY_OF_WEEK argument returns ________________.
    a. SUNDAY to SATURDAY
    b. a value from 0 to 6
    c. a value from 1 to 7
    d. a value of 1 to 6

EXERCISES

1. a. Create a class named Commission that includes three variables: a double sales figure, a double commission rate, and an integer commission rate. Create two overloaded methods named computeCommission(). The first method takes two double arguments representing sales and rate, multiplies them, and then displays the results. The second method takes two arguments: a double sales figure and an integer commission rate. This method must divide the commission rate figure by 100.0 before multiplying by the sales figure and displaying the commission. Supply appropriate values for the variables, and write a main() method that tests each overloaded method. Save the program as **Commission.java** in

the Chapter.04 folder on your Student Disk, and then compile and test the program.

b. Add a third overloaded method to the Commission program you created in Exercise 1a. The third overloaded method takes a single argument representing sales. When this method is called, the commission rate is assumed to be 7.5 percent and the results are displayed. To test this method, add an appropriate call in the Commission program's main() method. Save the program as **Commission2.java** in the Chapter.04 folder on your Student Disk, and then compile and test it.

2. Create a class named Pay that includes five double variables that hold hours worked, rate of pay per hour, withholding rate, gross pay, and net pay. Create three overloaded computeNetPay() methods. Gross pay is computed as hours worked, multiplied by pay per hour. When computeNetPay() receives values for hours, pay rate, and withholding rate, it computes the gross pay and reduces it by the appropriate withholding amount to produce the net pay. When computeNetPay() receives two arguments, the withholding rate is assumed to be 15 percent. When computeNetPay() receives one argument, the withholding rate is assumed to be 15 percent, and the hourly rate is assumed to be 4.65. Write a main() method that tests all three overloaded methods. Save the program as **Pay.java** in the Chapter.04 folder on your Student Disk.
3. a. Create a class named Household that includes data fields for the number of occupants and the annual income, as well as methods named setOccupants(), setIncome(), getOccupants(), and getIncome() that set and return those values, respectively. Additionally, create a constructor that requires no arguments and automatically sets the occupants field to 1 and the income field to 0. Save this file as **Household.java** in the Chapter.04 folder on your Student Disk. Create a program named TestHousehold that demonstrates that each method works correctly. Save the file as **TestHousehold.java** in the Chapter.04 folder on your Student Disk.

   b. Create an additional overloaded constructor for the Household class you created in Exercise 3a. This constructor receives an integer argument and assigns the value to the occupants field. Add any needed statements to TestHousehold to ensure that the overloaded constructor works correctly, save it, and then test it.

   c. Create a third overloaded constructor for the Household class you created in Exercises 3a and 3b. This constructor receives two arguments, the values of which are assigned to the occupants and income fields, respectively. Alter the TestHousehold program to demonstrate that each version of the constructor works properly. Save the program, and then compile and test it.
4. Create a class named Box that includes integer data fields for length, width, and height. Create three constructors that require one, two, and three arguments, respectively. When one argument is used, assign it to length, assign zeros to height and width, and print "Line created". When two arguments are used, assign them to length and width, assign zero to height, and print "Rectangle created". When three arguments are used, assign them to the three variables and print "Box created".

Save this file as **Box.java** in the Chapter.04 folder of your Student Disk. Create a program named TestBox that demonstrates that each method works correctly. Save the test file as **TestBox.java** in the Chapter.04 folder on your Student Disk.

5. What is the result when you compile and run the following code? Why?

```
class Scope
{
 int scopeInt = 1;
 void scopeDisplay()
 {
   int scopeInt = 10;
   System.out.println("scopeInt = " + scopeInt);
}
 public static void main(String[] args)
{
   Scope scopeExercise = new Scope();
   scopeExercise.scopeDisplay();
 }
 }
```

6. a.What is the result when you compile and run the following code? Why?

```
class Overload
{
 public static void main(String[] args)
 {
   Overload overloadExercise = new Overload();
   overloadExercise.methodOv();
   overloadExercise.methodOv(6.1, 3);
 }
 void methodOv()
 {
   System.out.println("no arguments");
 }
 void methodOv(double dblArg, int intArg)
 {
   System.out.println("dblArg = " + dblArg + "intArg = " +
      intArg);
  }
}
```

b. What happens when you compile and run the program shown in Exercise 6a if you replace the line `overloadExercise.methodOv(6.1, 3);` with `overloadExercise.methodOv(6, 3);`, and why?

c. What happens if you change the program shown in Exercise 6a as follows, and why?

```
class Overload
{
   public static void main(String[] args)
```

```
     {
      Overload overloadExercise = new Overload();
      overloadExercise.methodOv(6.1, 3.2);
     }
     void methodOv(double dblArg, float fltArg)
     {
      System.out.println("dblArg = " + dblArg + " fltArg = "
   + fltArg);
      }
     void methodOv(float fltArg, double dblArg)
     {
      System.out.println("dblArg = " + dblArg + " fltArg = "
         + fltArg);
     }
   }
```

d. If the program shown in Exercise 6c results in a compile error, how would you fix the program so it compiles and runs successfully?

7. Create a class named Shirt with data fields for collar size and sleeve length. Include a constructor method that takes arguments for each field. Also include a String class variable named material and initialize it to "cotton". Write a program named TestShirt to instantiate three Shirt objects with different collar sizes and sleeve lengths, and then display all the data, including material, for each shirt. Save both the **Shirt.java** and **TestShirt.java** programs in the Chapter.04 folder of your Student Disk.

8. Create a class named CheckingAccount with data fields for an account number and a balance. Include a constructor method that takes arguments for each field. Include a double class variable that holds a value for the minimum balance required before a monthly fee is applied to the account. Set the minimum balance to 200.00. Write a program named **TestAccount** in which you instantiate two CheckingAccount objects and display the account number, balance, and minimum balance without fee for both accounts. Save both the **CheckingAccount.java** and **TestAccount.java** programs in the Chapter.04 folder on your Student Disk.

9. Write a Java program to determine the answers for each of the following:

   a. the square root of 30

   b. the sine and cosine of 100

   c. the value of the floor, ceiling, and round of 44.7

   d. the larger and the smaller of the character K and the integer 70

   Save the file as **MathTest.java** in the Chapter.04 folder on your Student Disk.

10. Write a program to calculate how many milliseconds it is from today until the first day of next summer (assume that this date is June 21). Use the Date class. Save the file as **Summer.java** in the Chapter.04 folder on your Student Disk.

11. Write a program to calculate how many days it is from today until the end of the current year. Use the Date class. Save the file as **YearEnd.java** in the Chapter.04 folder on your Student Disk.
12. What is the result when you compile and run the following code, and why?

```
public class MathEx6
{
  public static void main(String[] args)
  {
    System.out.println(Math.round(1.49));
    System.out.println(Math.round(1.50));
    System.out.println(Math.round(-1.49));
    System.out.println(Math.round(-1.50));
   }
}
```

13. What is the result when you compile and run the following code, and why?

```
public class MathEx13
{
 public static void main(String[] args)
 {
   System.out.println(Math.ceil(1.49));
   System.out.println(Math.ceil(1.50));
   System.out.println(Math.ceil(-1.49));
   System.out.println(Math.ceil(-1.50));
  }
}
```

14. What is the result when you compile and run the following code, and why?

```
public class MathEx14
{
 public static void main(String[] args)
 {
   System.out.println(Math.floor(1.49));
   System.out.println(Math.floor(1.50));
   System.out.println(Math.floor(-1.49));
   System.out.println(Math.floor(-1.50));
 }
}
```

15. Modify the Employee class shown in Figure 4-17 by changing the class name to EmployeeWithDate. Then change the showCompanyID() method so it shows the current date, in addition to the employee number and company ID. Save the file as **EmployeeWithDate.java** in the Chapter.04 folder on your Student Disk. Then write a program that creates and displays two or more EmployeeWithDate objects. Save this new program as **UseEmployeeWithDate.java** in the Chapter.04 folder on your Student Disk.

16. Write a program to calculate how many milliseconds it is from today until the first day of next summer (assume that this date is June 21). Use the GregorianCalendar class. Save the file as **Summer2.java** in the Chapter.04 folder on your Student Disk.
17. Write a program to calculate how many days it is from today until the end of the current year. Use the GregorianCalendar class. Save the file as **YearEnd2.java** in the Chapter.04 folder on your Student Disk.
18. Each of the following files in the Chapter.04 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, save DebugThree1.java as FixDebugFour1.java.
    a. DebugFour1.java
    b. DebugFour2.java
    c. DebugFour3.java
    d. DebugFour4.java

Case Project

The Pool Associates operates a business that offers a variety of services to the general public who own swimming pools. Each year, Pool Associates cleans local pools and fills those pools when needed. Because swimming pools require a different amount of time to service, your job is to write a program that calculates the amount of time it will take to fill the pools with water. Then a reasonable estimate of fill-up time can be made before the job begins. This will enable Pool Associates to charge for pool fill-up on the basis of estimated hours. The table below gives some necessary parameters for estimating the fill-up time for a small pool and a large pool. Calculate the fill-up time for a small pool and a large pool. A small pool is considered 20 by 12 by 4 feet, and a large pool 30 by 20 by 10 feet. Save the program as **Swimming.java** in the Chapter.04 folder on your Student Disk.

Problem Parameters
Pool Volume L*W*D
Pool Capacity L*W*D*CAPACITY
Time to Fill L*W*D*CAPACITY/(RATE_OF_FLOW*60)
RATE_OF_FLOW 50.0 gal/min
CAPACITY 7.5 gal/cubic foot
L – pool length
W – pool width
D – pool depth

CHAPTER

5 INPUT AND SELECTION

In this chapter, you will:

- ♦ Accept keyboard input
- ♦ Use the JOptionPane class for GUI input and output
- ♦ Draw flowcharts
- ♦ Make decisions with the `if` and `if...else` structures
- ♦ Use compound statements in an `if` or `if...else` structure
- ♦ Nest `if` and `if...else` statements
- ♦ Use AND and OR operators
- ♦ Use the `switch` statement
- ♦ Use the conditional and NOT operators
- ♦ Understand precedence

Lynn Greenbrier asks, "Why are you frowning?"

"It's fun writing programs," you tell her, "but I don't think my programs can do much yet. When I use programs written by other people, I can respond to questions and make choices. In addition, other people's programs keep running for a while—the programs I write finish as soon as they start."

"You're disappointed because the programs you've written so far simply carry out a sequence of steps," Lynn says. "You need to make your programs interactive by accepting user input. To be able to do this, you need to learn about decision-making structures."

PREVIEWING THE CHOOSEMANAGER PROGRAM USING THE EVENT CLASS

To preview the ChooseManager program using the Event class:

1. In your text editor, open the **Chap5Event.java** file from the Chapter.05 folder on your Student Disk and examine the code. This file contains a class definition for a class that stores information about events that Event Handlers Incorporated will handle. You will create a similar class file in this chapter.
2. At the command line, compile the **Chap5Event.java** file using the command **`javac Chap5Event.java`**.
3. Open the **Chap5ChooseManager.java** file from the Chapter.05 folder on your Student Disk and examine the code. This file contains a program that will demonstrate prompting the user for input and creating objects based on the input.
4. At the command line, compile the **Chap5ChooseManager.java** file using the command **`javac Chap5ChooseManager.java`**.
5. Execute the program by typing the command **`java Chap5ChooseManager`**. At the prompt to enter C, P, or N, ignore the directions, and enter an invalid letter. Do this as many times as you like—the program will continue to prompt you until you enter a valid letter. Then enter **C**, **P**, or **N** to see the name of the manager and the minimum charge assigned to your event. You will create a similar program in this chapter.

ACCEPTING KEYBOARD INPUT

In Chapters 1 through 4 of this book, you wrote programs that created objects, performed mathematical calculations, and produced output. A shortcoming of these programs is that when you write the program you must know the values with which you want to work. It is far more useful to provide values to your program at **run time**, that is, while the program is executing. A program that accepts values at run time is **interactive** because it exchanges communications, or interacts, with the user. Providing values during the execution of a program requires input; the simplest form of input to use is keyboard entry from the program's user.

You already have used the System class and its out object and println() method to produce output. The **in object** is similar; it has access to a method named read() that retrieves data from the keyboard. Figure 5-1 shows a program that accepts simple user input.

The DemoInput class shown in Figure 5-1 has just one method—a main() method. At the end of the line containing the main() method header is the phrase `throws Exception`. The main() methods you have written that use `System.out.println();` have not required this phrase, but programs you write using `System.in.read();` do. An **exception** is an error situation. Errors are the "exception to the rule." Unfortunately, when a program user provides input, all sorts of error situations can arise. For example, the keyboard might be disconnected or the user might enter the wrong type of data. As you become a better Java programmer, you learn to handle these exceptional situations by writing code to take appropriate action, such as issuing detailed messages that explain the problem to the user. For now, however, you can let the compiler handle the problem by **throwing the exception**, or passing the error to the operating system. The code `throws Exception` after the main() method header accomplishes this; a program that reads keyboard input will not compile without this phrase.

```
public class DemoInput
{
  public static void main(String[] args) throws Exception
  {
    char userInput;
    System.out.println("Please enter a character ");
    userInput = (char)System.in.read();
    System.out.println("You entered " + userInput);
  }
}
```

Figure 5-1 DemoInput program

Tip

You write `Exception` with an uppercase E because it is a class name. Classes, by convention, begin with uppercase letters.

In Figure 5-1, a character named userInput is declared inside the main() method of the DemoInput program. The string "Please enter a character " prints on the screen. A message requesting user input commonly is called a **prompt** because it prompts or coaches the user to enter an appropriate response.

Tip

You are not required to supply a prompt every time there is user input, but you almost always will want to do so. Unless you supply a prompt, your user will see a blank screen and won't know how to proceed.

The statement `userInput = (char)System.in.read();` in the DemoInput program accomplishes three separate tasks:

- The method call `System.in.read();` gets the input from the keyboard. The read() method accepts a byte and returns an integer.
- The cast `(char)` converts the returned integer into a character.
- The assignment `userInput =` assigns the converted character to the variable userInput.

At first, it might not make sense that `System.in.read();` returns an integer value. However, Java's creators chose to have `System.in.read();` behave this way for the following reasons:

- To the computer, all values are integers because computers store input (and everything else) as a series of 0s and 1s. The character A, for example, is stored in Unicode as 0000 0000 0100 0001, which can also be expressed as '\u0041' or decimal 65. You learned (in Chapter 2) that the ASCII code is an 8-bit code. Unicode is a 16-bit code that represents a much larger character set that includes special and international characters.
- The System.in.read() method must return a value to indicate that no input is available. For example, when you use System.in.read() to read records from a disk file, at some point the compiler reaches the end of the file and no more input is available. Java's creators decided that the System.in.read() method should return the value -1 when the compiler reaches the end of a file. To accomplish this, the read() method must have a return type of int.

The final statement in the DemoInput program shown in Figure 5-1, `System.out.println("You entered " + userInput);`, **echoes**, or repeats, the userInput character. When you write interactive programs, it is a good idea to echo the input so the user can visually confirm that the data is correct.

When you run the DemoInput program, the prompt appears on the screen. The program will not proceed any further until you type a character and press [Enter]. The read() method accepts precisely one byte of input. Therefore, you cannot enter a floating-point number or a string of characters.

Next you will write a simple program that accepts three bytes of user input and echoes them.

To write a program that accepts and echoes user input:

1. Start your text editor, and then open a new text file.
2. Type the class header for a UsersInitials class, **`public class UsersInitials`**, press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
3. Type the header for the main() method, **`public static void main(String[] args) throws Exception`**, press **[Enter]**, type the opening curly brace for the main() method, and then press **[Enter]** again.
4. Type the following declarations for three character variables: **`char firstInit, middleInit, lastInit;`**.
5. On new lines, prompt the user for three initials by typing the following:
   ```
   System.out.println("Please enter your three initials.");
   System.out.println
      ("Do not use periods or spaces between initials.");
   System.out.println("Press Enter when you're done.");
   ```

The instruction "Do not use periods or spaces between initials." is important because you will write the program to accept only three characters from the keyboard. If a user enters A.B.C., then six characters were entered—three letters and three periods. The first letter would become firstInit, the first period would become secondInit, and the second letter would become thirdInit. There would be no room to store the second period, the third letter, or the last period.

Tip

It is customary to type a println() statement all on one line. The println() statement sometimes appears on two lines in this book due to printing limitations.

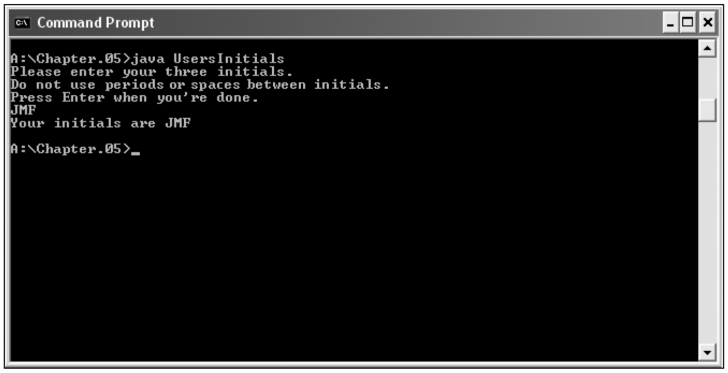
6. On new lines, type the following code to read each of the three initials into the appropriate variables:

```
firstInit = (char)System.in.read();
middleInit = (char)System.in.read();
lastInit = (char)System.in.read();
```

7. On a new line, type the following code to write the statements that will echo the three initials to the screen:

```
System.out.println("Your initials are " + firstInit +
   middleInit + lastInit);
```

8. On new lines, type the two closing curly braces that respectively close the main() method and the UsersInitials class.
9. Save the file as **UsersInitials.java** in the Chapter.05 folder on your Student Disk, and then compile and run the program. When you are prompted for three initials, enter any three characters and confirm that they are echoed to the screen correctly. Your output should look like Figure 5-2.



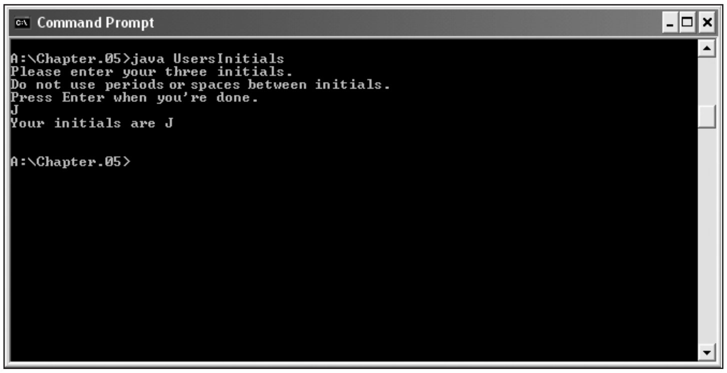
```
A:\Chapter.05>java UsersInitials
Please enter your three initials.
Do not use periods or spaces between initials.
Press Enter when you're done.
JMF
Your initials are JMF

A:\Chapter.05>_
```

Figure 5-2 Output of the UsersInitials program

The UsersInitials program works correctly as long as the user follows directions by entering three initials and pressing [Enter] only once after typing all three initials. However, just as if the user types periods between initials, a problem also occurs if the user presses [Enter] after typing each initial, as you will see next.

To demonstrate that the user should not press [Enter] after typing each initial:

1. Run the UsersInitials program again at the command prompt. When you see the prompt to enter your initials, type an initial and then press **[Enter]**. The program will terminate before you can type the second initial. The output will display only one initial, as shown in Figure 5-3.



```
A:\Chapter.05>java UsersInitials
Please enter your three initials.
Do not use periods or spaces between initials.
Press Enter when you're done.
J
Your initials are J

A:\Chapter.05>
```

Figure 5-3 Output of the UsersInitials program when the user presses [Enter] after the first initial

The problem occurs because when you use read() to accept a character from the keyboard, every key you press—including [Enter]—is accepted, one at a time. When you type your first initial, it is correctly stored in the firstInit variable. When you press [Enter] after entering the first initial, the value for [Enter] is stored in two bytes—the middleInit and the lastInit variables. When all three variables display on the screen, you see the first initial and the insertion point on a new line below the initial. The insertion point advances a line because the middleInit and lastInit together hold the [Enter] value.

The values for the two bytes occupied by [Enter] are '\u000D' and '\u000A', or decimal 13 and 10.

You can deal with this input problem by being very specific in your instructions to the user and insisting that the user type all three initials before pressing [Enter]. Alternately, you can ask the user for one initial at a time, and take care of [Enter] yourself. You can absorb the extra [Enter] key after each initial by reading it in with two read() method calls, and then not storing the bytes anywhere, as you will see next.

Tip

Depending on the version of JDK you are using, you might require only one extra read() statement to absorb [Enter]. For now, use one or two read() statements so all programs work correctly with your compiler. Later in this chapter, you will learn to use dialog boxes to eliminate the [Enter] problem.

To eliminate the [Enter] problem in the UsersInitials program:

1. In the UsersInitials.java text file, change the class name to **UsersInitials2**. Delete the following three lines of code that prompt the user for initials:
```
System.out.println("Please enter your three initials.");
System.out.println
   ("Do not use periods or spaces between initials.");
System.out.println("Press Enter when you're done.");
```
2. Replace the deleted lines with the following single statement: **`System.out.print("Enter your first initial and press Enter. ");`**.
3. Position the insertion point at the end of the read() statement that reads the firstInit variable, press **[Enter]** to start a new line, and then type the following statements to read in the two [Enter] bytes without storing them: **`System.in.read(); System.in.read();`**.
4. Press **[Enter]**, and then type the following prompt for the second initial on the new line: **`System.out.print("Enter your second initial and press Enter. ");`**.
5. Position the insertion point at the end of the statement that reads the middleInit variable, press **[Enter]** to start a new line, and then type the following statements to read [Enter] pressed after the second initial, and to prompt for the third initial:
```
System.in.read();
System.in.read();
System.out.print
   ("Enter your third initial and press Enter. ");
```

Tip

You might choose to place a final `System.in.read(); System.in.read();` statement after the statement that reads the third initial, to discard its [Enter]. Because the program doesn't accept any more input after reading the third initial, these extra read() statements will not affect program execution. However, if you add extra read() statements to absorb the last [Enter], the [Enter] following the third initial already will be discarded if you add additional input steps to this program later.

6. Save the program as **UsersInitials2.java**, compile, and run the program. Respond to each prompt by typing your initial and then pressing **[Enter]**. Your output should display your three initials correctly.

Using the JOptionPane Class for GUI Input and Output

Often referred to as **Swing components**, the classes found in the javax.swing package define GUI elements and provide alternatives to the System.in.read() and System.out.println() methods found in the java.lang package. The Swing classes are part of a more general set of GUI programming capabilities that are collectively referred to as the **Java Foundation Classes**, or **JFC** for short. JFC includes Swing component classes and selected classes from the java.awt package.

Tip

Swing will be introduced in greater detail starting in Chapter 9. In this chapter, only a few components that deal with input and output are demonstrated.

To access the Swing components used in this chapter, it is necessary to import the javax.swing package using `import javax.swing.*;`. Recall that you learned that the asterisk(*) is used as a wildcard symbol to represent all the classes in a package.

The Swing component **JOptionPane** can be used to create standard dialog boxes. These dialog boxes are small windows that ask a question, warn a user, or provide brief important user messages. As such, they provide a GUI interface to communicate with the user, as opposed to the nonwindowed standard input and output methods presented so far. These dialog boxes also provide methods that automatically handle input and output.

Three standard dialog boxes of the JOptionPane class are:

- InputDialog—prompts the user for text input
- MessageDialog—displays a user message
- ConfirmDialog—asks the user a question, with buttons for Yes, No, and Cancel responses

Input Dialog Boxes

An **input dialog box** asks a question and uses a text field for entering a response. You can create an input dialog box using the **showInputDialog() method**. There are two components or arguments with this method, the parent component and the string component. The string component is composed of a string or icon to be displayed in the dialog box. When no parent component is used, the keyword `null` is substituted.

The input dialog method returns a string that represents a user's response. In Chapter 2 you learned that a String is an object that can hold more than one character. In the example, `String firstName = "Audrey";`, the statement stores the name Audrey as a string in a variable named firstName. The ShowInputDialog() method returns a string. For example, Figure 5-4 shows an input dialog box created with the statement `String response = JOptionPane.showInputDialog(null, "Enter your first name");`. If the user types "Audrey", then clicks the OK button or presses [Enter] on the keyboard, the response string will contain "Audrey".

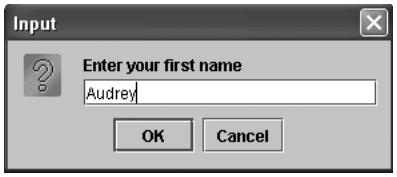


Figure 5-4 Input dialog box containing user input

Tip: When the keyword `null` is used as the first argument in the showInputDialog() method, it can be omitted entirely. Thus, `String response = JOptionPane.showInputDialog("Enter your first name");` is also correct.

An overloaded version of the showInputDialog() method contains more options; it allows the programmer flexibility in controlling the appearance of the input dialog box. The **showInputDialog() method with four arguments** can be used to display a title in the dialog box title bar and a message that describes the type of dialog box. The first two arguments (or components) are the same as in the shorter method, and the last two arguments are as follows:

- The title to be displayed in the title bar
- A class variable describing the type of dialog box—ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE, or WARNING_MESSAGE

When the statement `JOptionPane.showInputDialog(null,"What is your area code?:", "Enter area code", JOptionPane.QUESTION_MESSAGE);` is executed, it displays the input dialog box shown in Figure 5-5. Note that the title bar displays "Enter area code," and the dialog box shows a question mark icon.

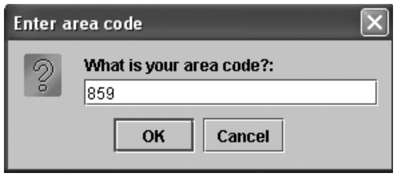
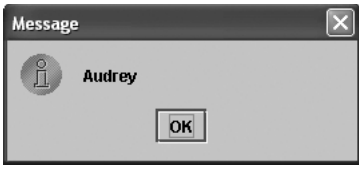


Figure 5-5 Input dialog box with four arguments

Message Dialog Boxes

The **message dialog box** uses a simple window to display information. A message dialog box is created with the **showMessageDialog() method**. The arguments are the same as with the input dialog box—the parent component (if any) and the string component composed of a string or icon to display in the box. Unlike the input dialog box, the message dialog box does not return any user response. For example, if a string variable named response holds "Audrey", then the statement `JOptionPane.showMessageDialog(null,response);` displays "Audrey" as shown in Figure 5-6.



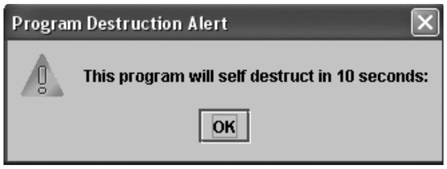
Audrey

Figure 5-6 The resulting message dialog box

You can also create a message dialog box with more options. The **showMessageDialog() method with four arguments** is used to display a title in the dialog box title bar and a message that best describes its message type. The first two arguments are the same as the shorter method, and the last two arguments are as follows:

- The title to be displayed in the title bar
- A class variable describing the type of dialog box—ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION_MESSAGE, and WARNING_MESSAGE

When the statement `JOptionPane.showMessageDialog(null,"This program will self destruct in 10 seconds:","Program Destruction Alert", JOptionPane.WARNING_MESSAGE);` is executed, it displays a message dialog box as shown in Figure 5-7. Note that the title bar displays Program Destruction Alert, and the dialog box shows a warning sign with an exclamation mark.



This program will self destruct in 10 seconds:

Figure 5-7 Message dialog box with four arguments

Confirm Dialog Boxes

An easy way to create a confirm dialog box which displays the options Yes, No, and Cancel, is with the **showConfirmDialog() method**. Like the previous dialog boxes, the two arguments are the same—the parent component (if any) and the string component composed of a string or icon to display in the box. When you use `null` as the first argument to showConfirmDialog(), the dialog box is centered on the screen. When you use the showConfirmDialog() method, it returns one of three possible integer values, each a class variable of JOptionPane:YES_OPTION, NO_OPTION, and CANCEL_OPTION. To set an integer named option equal to the class variable NO_OPTION, you write the statement `int option = JOptionPane.NO_OPTION;`.

The statement `JOptionPane.showConfirmDialog(null, "Are you ready to quit");` displays the dialog box shown in Figure 5-8. The program will not terminate, however, without a statement to stop the program. To stop the program, you can call the exit() method from the System class with an argument of 0, for example, `System.exit(0);`. When you include this statement at the end of your dialog box programs they terminate correctly.

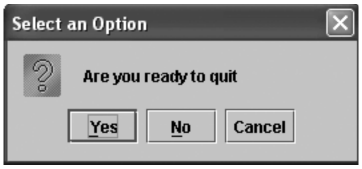
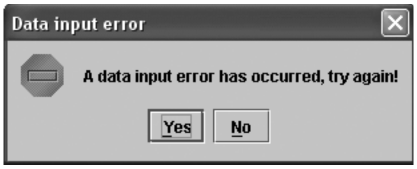


Figure 5-8 The resulting confirm dialog box

You can also create a confirm dialog box with five components. The first two arguments are the same as for the previous input dialog box and message dialog box examples. The last three components represent:

- The title to be displayed in the title bar
- An integer that indicates which option button will be shown (It should be equal to the class variables YES_NO_CANCEL_OPTION or YES_NO_OPTION.)
- An integer that describes the kind of dialog box, using the class variables ERROR_MESSAGE, INFORMATION_MESSAGE, PLAIN_MESSAGE, QUESTION MESSAGE, or WARNING_MESSAGE

When the statement `JOptionPane.showConfirmDialog(null,"A data input error has occurred, try again!", "Data input error", JOptionPane.YES_NO_OPTION, JOptionPane.ERROR_MESSAGE);` is executed, it displays an input dialog box as shown in Figure 5-9. Note that the title bar displays Data Input Error, the YES and NO buttons appear, and the dialog box shows the error message, "A data input error has occurred, try again!" It also displays the ERROR_MESSAGE type with an octagonal icon.



Data input error

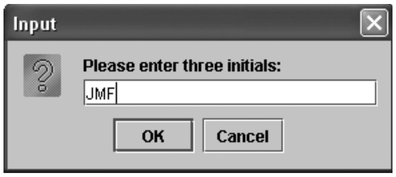
A data input error has occurred, try again!

Figure 5-9 Confirm dialog box with five arguments

Next you will use the dialog box methods to create a class comparable to the UserInitials class presented earlier in this chapter.

To write a program that accepts and displays user input:

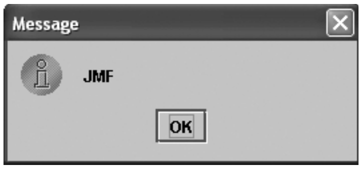
1. Start your text editor, open a new file, type the import statement, **`import javax.swing.*;`**, and then press **[Enter]**.
2. Type the class header for a DialogInitials class, **`public class DialogInitials`**, press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
3. Type the header for the main() method, **`public static void main(String[] args)`**, press **[Enter]**, type the opening curly brace for the main() method, and then press **[Enter]** again.
4. Type the statement to create an input dialog box and store user input in a String variable named response; **`String response = JOptionPane.showInputDialog(null, "Please enter three initials:");`**, and then press **[Enter]**.
5. Type the statement to create a message dialog box to display the user's input stored in the String response; **`JOptionPane.showMessageDialog(null, response);`**, and then press **[Enter]**.
6. Type the statement to create a confirm dialog box that asks the user to confirm that the initials entered are correct: **`JOptionPane.showConfirmDialog (null, "Are the initials correct?");`**.
7. Add the statement to close the program: **`System.exit(0);`**. Press **[Enter]**, type the closing curly braces for the main() method, press **[Enter]** a second time, and then type the closing curly brace for the class.
8. Save the file as **DialogInitials** in the Chapter.05 folder on your Student Disk, and then compile and run the program. When you are prompted for three initials, enter any three letters and confirm that they are displayed correctly. Finally, click the **YES** button when the message "Are the initials correct?" appears. The sequence of input, output, and confirmation of input are shown in Figures 5-10 through 5-12.


Input

Please enter three initials:

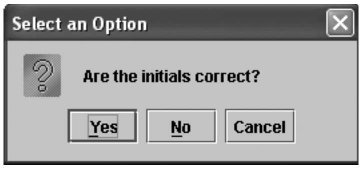
JMF

Figure 5-10 Input dialog box with initials entered


Message

JMF

Figure 5-11 Message dialog box


Select an Option

Are the initials correct?

Figure 5-12 Confirm dialog box

Tip: Examples of the capabilities of dialog boxes introduced in this chapter will be expanded and used as programming structures for decision making in this and future chapters.

DRAWING FLOWCHARTS

This section is not a thorough discussion of flowcharting. Instead, it is a brief introduction, so you can use flowcharts as a visual aid in the next sections of this chapter.

When computer programmers write programs, they seldom simply sit down at a keyboard and begin typing. Programmers must plan the complex portions of programs using paper and pencil tools. Programmers often use **pseudocode**, or lists of tasks that must be accomplished, to help them plan a program's logic. Using pseudocode requires that you write down the steps needed to accomplish a given task. You write pseudocode in English; you concentrate on the logic required, and not the syntax used in any programming language. As a matter of fact, a task you pseudocode does not have to be computer related. If you have ever written a list of things you must accomplish during a day (for example; 1. Wash car, 2. Study for test, 3. Buy birthday gift for Mom), then you have written pseudocode. A **flowchart** is similar to pseudocode, but you write the steps in diagram form, as a series of shapes connected by arrows.

Tip You learned the difference between a program's logic and its syntax in the Running a Program section of Chapter 1.

Some programmers use a variety of shapes to represent different tasks in their flowcharts, but you can draw simple flowcharts that express very complex situations using just rectangles and diamonds. You use a rectangle to represent any unconditional step, and a diamond to represent any decision. For example, Figure 5-13 shows a flowchart of a day's tasks.

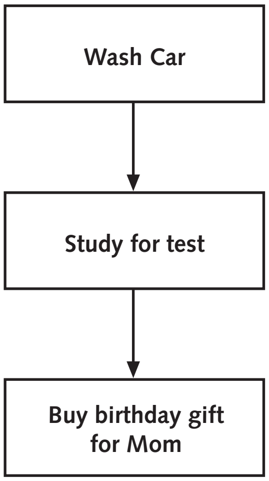


Figure 5-13 Flowchart of a day's tasks

Sometimes your days don't consist of a series of unconditional tasks—some tasks may or may not occur based on decisions you make. Using diamond shapes, flowchart creators draw paths to alternate courses of action emanating from the sides of the diamonds. Figure 5-14 shows a flowchart of a day's tasks in which some tasks are based on decisions.

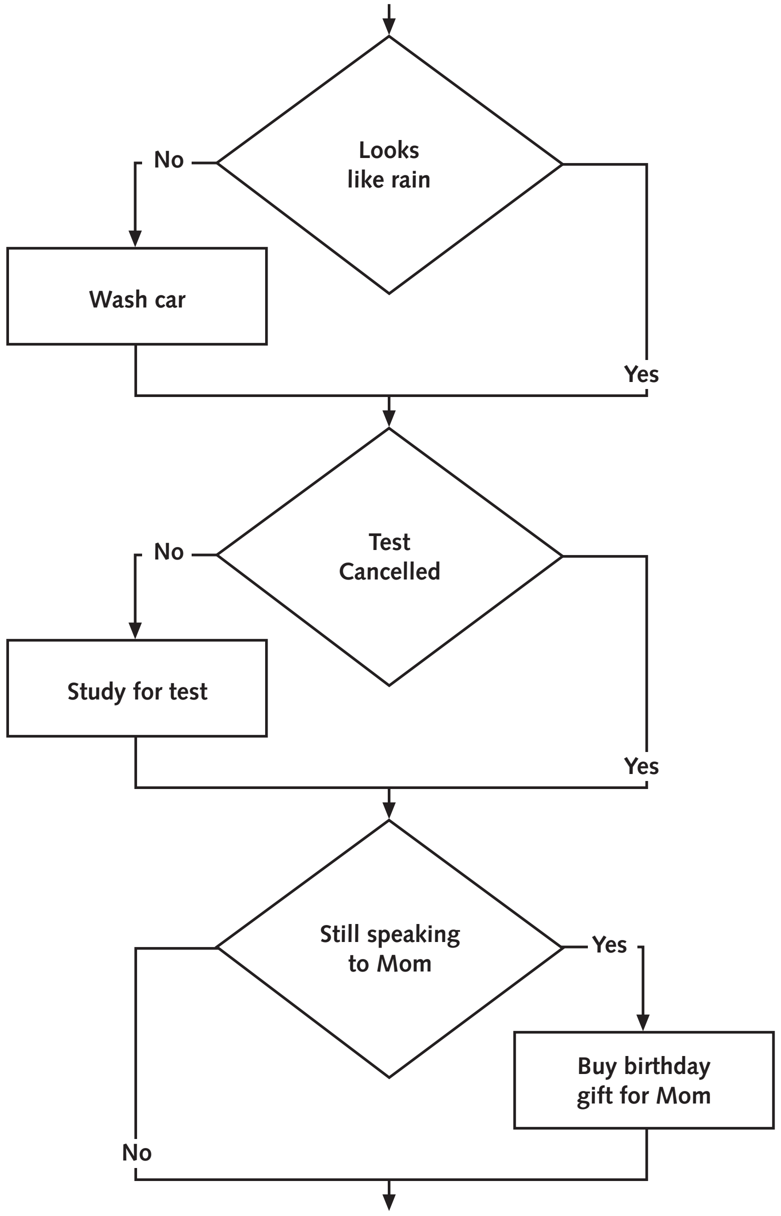


Figure 5-14 Flowchart of a day's tasks with decisions

Making Decisions with the `if` and `if...else` Structures

You can already write a program that produces different output based on input; for example, a user who types JFK into the UsersInitials program receives different output than a user who types FDR. Additionally, after you learn to write programs that can accept input, you gain a powerful new capability—you can alter the events that occur within a program based on user input. Now you can make decisions.

Making a **decision** involves choosing between alternate courses of action based on some value within a program. For example, the program that produces your paycheck can make decisions about the proper amount to withhold for taxes, the program that guides a missile can alter its course, and a program that monitors your blood pressure during surgery can determine when to sound an alarm. Making decisions is what makes computer programs seem "smart."

The value upon which a decision is made is always a Boolean value, which in turn is always one of two values—`true` or `false`. Figure 5-15 shows the logic of the decision structure. Recall from Figure 5-14 that each decision structure was couched in a yes and no answer, rather than a Boolean `true` or `false` value.

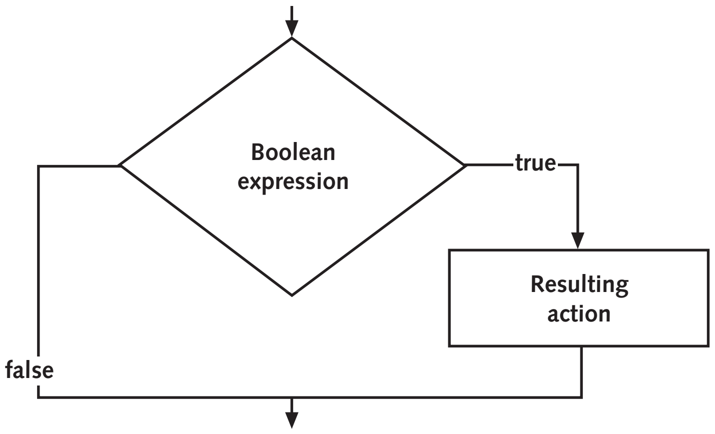


Figure 5-15 Decision structure

One statement you can use to make a decision is the **`if` statement**. For example, you can store a value in an integer variable named someVariable, and then print the value of someVariable when the user wants to see it. As you can see in Figure 5-16, you can prompt the user to enter Y or N (for "Yes" or "No") and store the response in a character variable.

```
char userResponse;
int someVariable = 512;
System.out.println
  ("Do you want to see the value of someVariable?");
System.out.println("Enter Y for yes or N for no");
char userResponse = (char)System.in.read();
```

Figure 5-16 Storing a user's response

The following is the `if` statement that makes the decision to print. Note that the double equals sign (==) is used to determine equality.

```
if(userResponse == 'Y')
  System.out.println
   ("The value of someVariable is " + someVariable);
```

Tip: Remember that you reference character values using single quotation marks.

If the userResponse variable holds the value 'Y,' then the Boolean value of the expression `userResponse == 'Y'` is `true`, and the subsequent println() statement will execute. If the value of the expression `userResponse == 'Y'` is `false`, then the println() statement will not execute. The `userResponse == 'Y'` expression will be `false` if userResponse holds anything other than 'Y,' including 'y,' 'N,' 'n,' 'A,' or any other value.

The Boolean expression `(userResponse == 'Y')` must appear within parentheses.

You are not required to leave a space between the keyword `if` and the opening parentheses, but if you do, the statement is easier to read and is less likely to be confused with a method call. Also, there is no semicolon at the end of the first line of the `if` statement `if(userResponse == 'Y')` because the statement does not end there. The statement ends after the println() call, so that is where you type the semicolon. You also could type the same statement on one line and execute it in the same manner. However, the two-line format is more conventional and easier to read, so you will usually type `if` and the Boolean expression on one line, press [Enter], and then indent a few spaces before coding the action that will occur if the Boolean expression evaluates as `true`. Be careful—when you use the two-line format, do not type a semicolon at the end of the first line, as in the following example:

```
if(userResponse == 'Y');
// Notice the incorrect semicolon here
 System.out.println("The value of someVariable is " +
 someVariable);
```

When this `if` expression is evaluated, the statement ends if it evaluates as `true`. Whether the expression evaluates as `true` or `false`, execution continues with the next independent statement that prints someVariable. In this case, because of the incorrect semicolon, the `if` statement accomplishes nothing.

Another very common programming error occurs when a programmer uses a single equals sign rather than the double equals sign when attempting to determine equivalency. The expression `userResponse = 'Y'` does not compare userResponse to 'Y.' Instead, it attempts to assign the value 'Y' to the userResponse variable. When the expression is part of an `if` statement, this assignment is illegal. The confusion arises in part because the single equals sign is used within Boolean expressions in `if` statements in many other programming languages, such as COBOL, Pascal, and BASIC. Adding to the confusion, Java programmers use the word *equals* when speaking of equivalencies. For example, you might say, "If userResponse *equals* 'Y'…" rather than "If userResponse is *equivalent* to 'Y'…"

An alternative to using a Boolean expression, such as `userResponse == 'Y'`, is to store the Boolean expression's value in a Boolean variable. For example, if userSaidYes is a Boolean variable, then `userSaidYes = (userResponse == 'Y');` compares userResponse to 'Y' and stores `true` or `false` in userSaidYes. Then you can write the `if` as `if(userSaidYes)...`. This adds an extra step to the program, but makes the `if` statement more similar to an English statement.

The `if...else` Structure

Consider the following statement:

```
if(userResponse == 'Y')
System.out.println("The value of someVariable is "
 + someVariable);
```

Such a statement is sometimes called a **single-alternative `if`** because you only perform an action based on one alternative, which is the case when userResponse is 'Y.' Often you require two options for the next course of action, or a **dual-alternative `if`**. For example, if the user does not respond 'Y' to a prompt, you might want to print a message that at least acknowledges that the response was received. The **`if...else` statement** provides the mechanism to perform one action when a Boolean expression evaluates as `true`, and performs a different action when a Boolean expression evaluates as `false`. Figure 5-17 shows the logic for the `if...else` structure. Figure 5-18 shows an example of the `if...else` structure coded in Java. In Figure 5-18, the value of someVariable is printed when userResponse is equivalent to 'Y.' When userResponse is any other value, the program prints the message "Too bad."

Tip

You can code an `if` without an `else`, but it is illegal to code an `else` without an `if`.

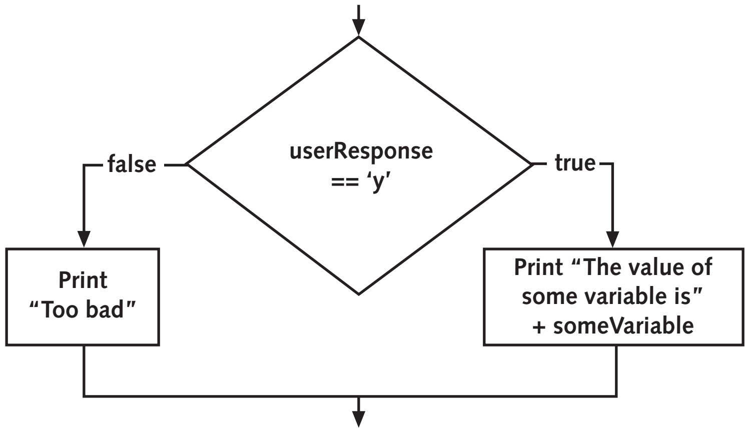


Figure 5-17 The `if...else` structure

```
if(userResponse == 'Y')
  System.out.println("The value of someVariable is"
    + someVariable);
else
  System.out.println("Too bad.");
```

Figure 5-18 A dual-alternative `if` structure

Tip: The indentation shown in the `if...else` example shown in Figure 5-18 is not required, but is standard usage. You vertically align the keyword `if` with the keyword `else`, and then indent the action statements that depend on the evaluation.

When you execute the code shown in Figure 5-18, only one of the println() statements will execute; the one that executes depends upon the evaluation of `(userResponse == 'Y')`. Each println() statement is a complete statement, so each statement ends with a semicolon.

Next you will start writing a program for Event Handlers Incorporated that determines which of several employees will be assigned to manage a client's scheduled event. To begin, you will prompt the user to answer a question about the event type, and then the program will display the name of the manager who handles such events. There are two event types: corporate events, handled by Dustin Britt; and private events, handled by Carmen Lindsey.

To write a program that chooses between two managers:

1. Open a new text file, and then enter the code to choose a manager:

```
public class ChooseManager
{
 public static void main(String[] args) throws Exception
  {
```

2. On a new line, declare a variable that will hold the type of event by typing **`char eventType;`**.
3. On a new line, type the three-line prompt that explains what is expected of the user:

```
System.out.println
 ("Enter type of event you are scheduling");
System.out.println("C for a corporate event");
System.out.println("P for a private event");
```

4. On a new line, type the statement that reads in the type of event:

```
eventType = (char)System.in.read();.
```

5. On a new line, type the print() statement that explains the output:

```
System.out.print("The manager for this event will be ");.
```

6. Code the `if...else` structure to determine which of two managers will be assigned to the event.

```
if(eventType == 'C')
 System.out.println("Dustin Britt");
else
 System.out.println("Carmen Lindsey");
```

7. Type the two closing curly braces to end the main() method and the ChooseManager class.
8. Save the program as **ChooseManager.java** in the Chapter.05 folder on your Student Disk, then compile and run the program. Confirm that the program selects the correct manager when you choose C for a corporate event or P for a private event.

USING COMPOUND STATEMENTS IN AN `if` OR `if...else` STRUCTURE

Often there is more than one action to take following the evaluation of a Boolean expression within an `if` statement. For example, you might want to print several separate lines of output or perform several mathematical calculations. To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. For example, the program segment shown in Figure 5-19 determines whether an employee has worked more than 40 hours in a single week, and if so, the program computes regular and overtime salary, and then prints the results.

Tip

When you create a block, you do not have to place multiple statements within it. It is perfectly legal to block a single statement.

```
if(hoursWorked > 40)
{
  regularPay = 40 * rate;
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
    // Time and a half for hours over 40
  System.out.println("Regular pay is " + regularPay);
  System.out.println("Overtime pay is " + overTimePay);
} // The if structure ends here
```

Figure 5-19 An `if` statement with multiple dependent statements

If you compare Figures 5-19 and 5-20, you will see that in Figure 5-19, the regularPay calculation, the overTimePay calculation, and the println() statement are executed only when `hoursWorked > 40` is `true`. In Figure 5-20, the curly braces are omitted. Within the program in Figure 5-20, when `hoursWorked > 40` is `true`, regularPay is calculated and the `if` expression ends. The next three statements that compute overTimePay and print the results always execute every time the program runs, no matter what value is stored in hours. These last three statements are not dependent on the `if` statement; they are independent, stand-alone statements. The indentation might be deceiving; it looks as though four statements depend on the `if` statement, but indentation does not cause statements following an `if` statement to be dependent. Rather, curly braces are required if the four statements must be treated as a block.

```
if(hoursWorked > 40)
  regularPay = 40 * rate;  // The if structure ends here
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
  System.out.println("Regular pay is " + regularPay);
  System.out.println("Overtime pay is " + overTimePay);
```

Figure 5-20 `if` statement with a single dependent statement

The code shown in Figure 5-20 might not compile if regularPay is not assigned a value—the compiler will recognize that you are attempting to print the value of regularPay without calculating it. However, if you have assigned a value to regularPay, you can compile the program, but the output still will not be what you intended. Within the code segment shown in Figure 5-20, if hoursWorked is greater than 40, then the program properly calculates both regular and overtime pay. Because `hoursWorked > 40` is `true`, the regularPay calculation is made. The overTimePay calculation and the println() statements will execute as well because they are just statements that always execute and do not depend on the `if` statement.

However, in Figure 5-20, if the hoursWorked value is 40 or less—30, for example—then the regularPay calculation will not execute (it executes only `if(hoursWorked > 40)`), but the next three independent statements will execute. The variable regularPay will hold whatever value you have previously assigned to it—0.0, for example—and the program will calculate the value of overTimePay as a negative number (because 30 - 40 results in -10). Therefore, the output will be incorrect.

Just as you can block statements to depend on an `if`, you can also block statements to depend on an `else`. Figure 5-21 shows an `if` structure with two dependent statements and an `else` with two dependent statements. The program executes the final two println() statements without regard to the hoursWorked variable's value; the println() statements are not part of the `if` structure.

```
if(hoursWorked > 40)
{
  regularPay = 40 * rate;
  overTimePay = (hoursWorked - 40) * 1.5 * rate;
  // Time and a half for hours over 40
}
else
{
  regularPay = hours * rate;
  overTimePay = 0.0;
}
System.out.println("Regular pay is " + regularPay);
System.out.println("Overtime pay is " + overTimePay);
```

Figure 5-21 An `if...else` statement with multiple dependent statements

Next you will create an Event class. Each Event object includes two data fields: the type of event, and the base price Event Handlers charges per hour for the event type. The Event class also contains a constructor method and get methods for the two fields.

To create the Event class:

1. Open a new text file, and type the class header for the Event class, **`public class Event.`** Press **[Enter]** and then type the opening curly brace for the class on the new line.
2. Type the following declarations for two data fields to hold the type of event and the minimum hourly rate that Event Handlers charges:

   ```
   private char eventType;
   private double eventMinRate;
   ```

3. Type the following constructor for the Event class. The constructor will require two arguments with which you will fill the two data fields.

   ```
   public Event(char event, double rate)
   {
    eventType = event;
    eventMinRate = rate;
   }
   ```

4. On new lines, type the following two get methods that return the field values:

   ```
   public char getEventType()
   {
    return eventType;
   }
   public double getEventMinRate()
   {
    return eventMinRate;
   }
   ```

5. Type the closing curly brace for the class.
6. Save the file as **Event.java** in the Chapter.05 folder on your Student Disk, then compile the file and correct any errors.

Now that you have created an Event class, you can modify the ChooseManager program to perform multiple tasks based on user input. You will display a message to indicate which manager is assigned to the event, and you will also instantiate a unique Event object, with different minimum rates to charge based on the type of event.

To modify the ChooseManager program:

1. Open the **ChooseManager.java** text file from the Chapter.05 folder on your Student Disk, and change the class name to **ChooseManager2**. You will declare two constants to hold the corporate hourly rate and the private hourly rate. If these hourly rates change in the future, they will be easy to locate at the top of the file, where you can change their values.
2. Position the insertion point just after the opening curly brace for the ChooseManager2 class, press **[Enter]** to start a new line, and then type the following two constants:

   ```
   static final double CORP_RATE = 75.99;
   static final double PRI_RATE = 47.99;
   ```

   Within the main() method of the ChooseManager2 class, you will define an Event object named anEvent. You do not want to construct the Event object until you discover whether it will be a corporate or private event; you simply want to declare it now.
3. Position the insertion point to the right of the statement that declares the eventType character variable, press **[Enter]** to start a new line, and then type the event declaration as **`Event anEvent;`**.
4. Next type the following lines to modify the `if...else` structure that currently prints a manager's name, so that the `if` and `else` each control a block of two statements. The first statement in each block still prints the manager's name. The second statement constructs an appropriate Event object.

   ```
   if(eventType == 'C')
   {
   ```

```
  System.out.println("Dustin Britt");
  anEvent = new Event(eventType, CORP_RATE);
}
else
{
  System.out.println("Carmen Lindsey");
  anEvent = new Event(eventType, PRI_RATE);
}
```

5. To confirm that the event was constructed properly, type the following two println() statements immediately after the closing brace for the `if...else` structure:

```
System.out.println("Event type is " + anEvent.
getEventType());
System.out.println("Minimum rate charged is $" +
anEvent.getEventMinRate());
```

6. Save the program as **ChooseManager2.java**. Compile and run the program several times with different input at the prompt. Confirm that the output shows that the event has the correct manager, type, and rate based on how you respond to the prompt (with C or P).

NESTING `if` AND `if...else` STATEMENTS

Within an `if` or an `else` statement, you can code as many dependent statements as you need, including other `if` and `else` statements. Just as spoons are nested inside each other in a drawer, statements with an `if` inside another `if` commonly are called **nested `if` statements**. Nested `if` statements are particularly useful when two conditions must be met before some action is taken.

For example, suppose you want to pay a $50 bonus to a salesperson only if the salesperson sells more than three items that total more than $1,000 in value during a specified time. Figure 5-22 shows the logic for this situation. Figure 5-23 shows the code to solve the problem.

Notice there are no semicolons in the code shown in Figure 5-23 until after the `bonus = 50;` statement. The expression `itemsSold > 3` is evaluated. If this expression is `true`, then the program evaluates the second Boolean expression `(totalValue > 1000)`. If that expression is also `true`, then the bonus assignment executes and the `if` structure ends.

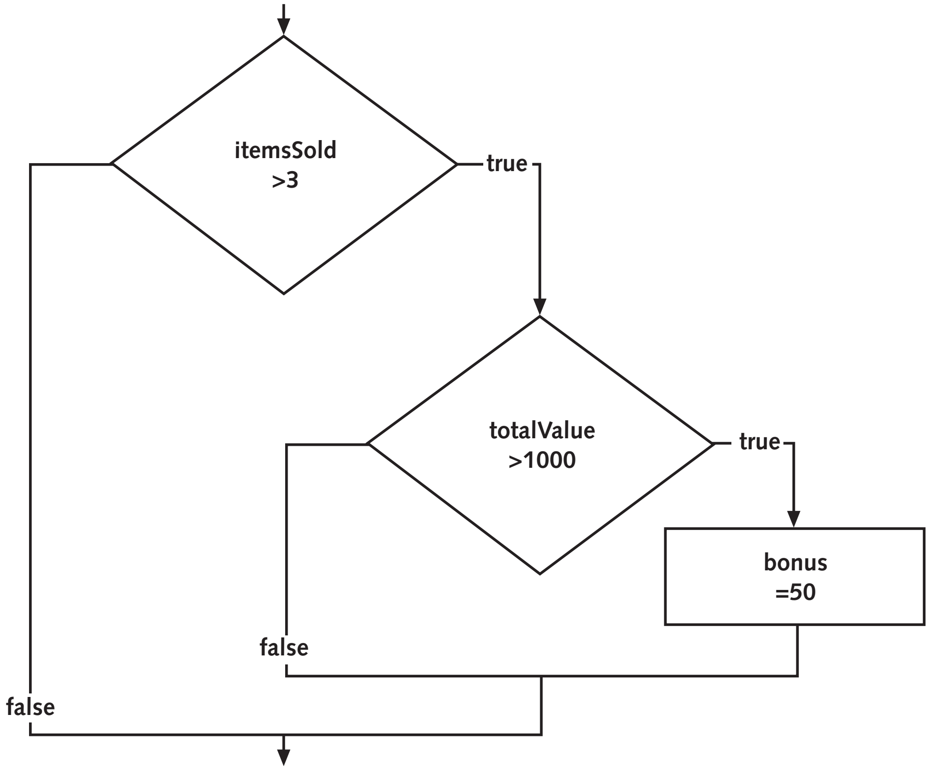


Figure 5-22 Nested `if` structure

```
if(itemsSold > 3)
  if(totalValue > 1000)
    bonus = 50;
```

Figure 5-23 Nested `if` statement

Using AND and OR Operators

For an alternative to nested `if` statements, you can use the **AND operator** within a Boolean expression to determine whether two expressions are both `true`. The AND operator is written as two ampersands (&&). For example, the code shown in Figure 5-24 works exactly the same as the code shown in Figure 5-23. The itemsSold variable is tested, and if it is greater than 3, then the totalValue is tested. If totalValue is greater than $1000, then the bonus is set to $50.

```
if(itemsSold > 3 && totalValue > 1000)
  bonus = 50;
```

Figure 5-24 Using the AND operator

You are never required to use the AND operator because using nested `if` statements always achieves the same result, but using the AND operator often makes your code more concise, less error prone, and easier to understand.

It is important to note that when you use the AND operator, you must include a complete Boolean expression on each side of the && operator. If you want to set a bonus to $400 if a saleAmount is both over $1000 and under $5000, the correct statement is `if(saleAmount > 1000 && saleAmount < 5000) bonus = 400;`. Even though the saleAmount variable is used on both sides of the AND expression, the statement `if(saleAmount > 1000 && < 5000)...` is incorrect and will not compile.

With the AND operator, both Boolean expressions must be `true` before the action in the statement can occur. You can use the **OR operator**, which is written as ||, when you want some action to occur, even if only one of two conditions is true. For example, if you want to give a bonus of $200 if a salesperson satisfies at least one of two conditions—selling more than 100 items or selling any number of items that total more than $3000 in value—then you can write the code using either of the ways shown in Figure 5-25. Figure 5-26 shows the program logic.

Tip

A common use of the OR operator is to decide to take action whether a character variable is uppercase or lowercase, as in `if(selection == 'A' || selection == 'a') ....` The subsequent action occurs whether the selection variable holds an uppercase or lowercase A.

```
// Using two ifs
if(itemsSold > 100)
  bonus = 200;
else if(totalValue > 3000)
  bonus = 200;
// Using the OR operator
if(itemsSold > 100 || totalValue > 3000)
  bonus = 200;
```

Figure 5-25 Code using two `if` statements and the OR operator

Sometimes situations arise in which there are more than two possible courses of action to take. Consider a situation in which salespeople can receive one of three possible commission rates based on their sales. For example, a sale totaling $1001 or more earns the salesperson an eight percent commission, a sale totaling $500 to $1000 earns six percent of the sale amount, and any sale totaling $499 or less earns five percent. Using three separate `if` statements to test single Boolean expressions results in some incorrect commissions. Examine the code shown in Figure 5-27.

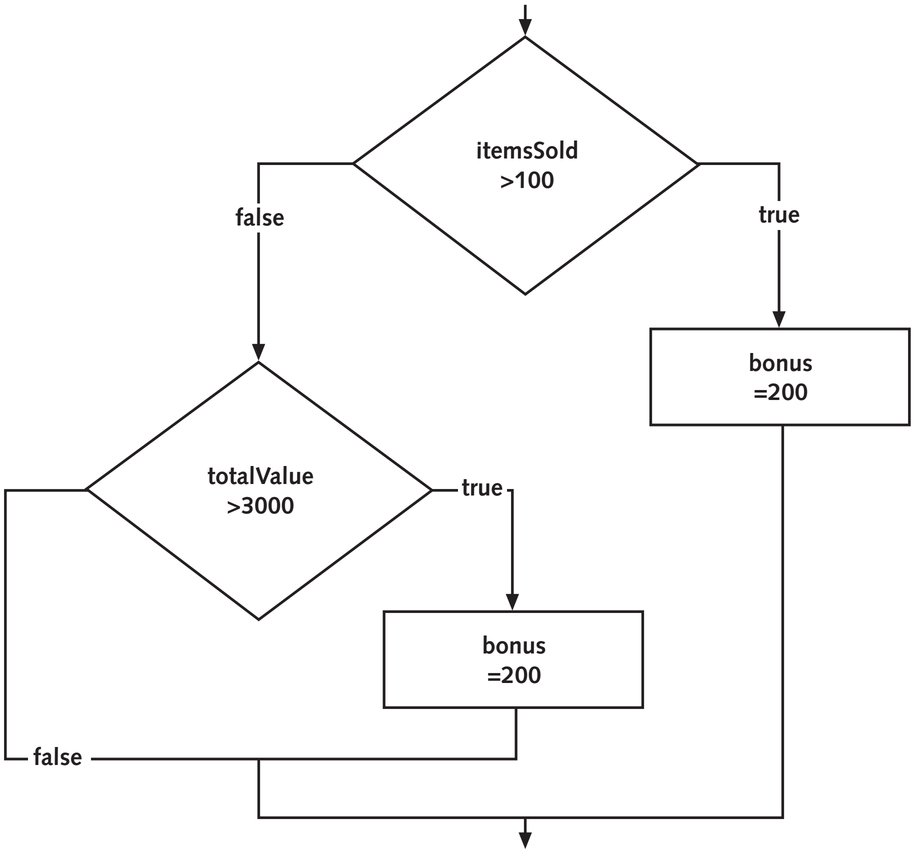


Figure 5-26 Diagram of the OR logic

```
if(saleAmount > 1000)
  commRate = .08;
if(saleAmount > 500)
  commRate = .06;
if(saleAmount <= 500)
  commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-27 Incorrect assignment of three commissions

Tip As long as you are dealing with whole dollar amounts, the expression `if(saleAmount > 1000)` can be expressed just as well as `if(saleAmount >= 1001)`. Additionally, `if(1000 < saleAmount)` and `if(1001 <= saleAmount)` have the same meaning. Use whichever has the clearest meaning for you.

Using the code shown in Figure 5-27, if a saleAmount is $5000, the first `if` statement executes. The Boolean expression `(saleAmount > 1000)` evaluates as `true`, and .08 is correctly assigned to commRate. However, when a saleAmount is $5000, the next `if` expression, `(saleAmount > 500)`, also evaluates as `true`, so the commRate, which was eight percent, is incorrectly reset to six percent.

A partial solution to this problem is to use an `else` statement following the `if(saleAmount >= 1000)` expression, as shown in Figure 5-28.

```
if(saleAmount > 1000)
  commRate = .08;
else if(saleAmount > 500)  // Notice the else
  commRate = .06;
if(saleAmount <= 500)
  commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-28 Inefficient assignment of three commissions

Tip You can place and indent the `if` following an `else`, but a program with many nested `if...else` combinations soon grows very long and "deep," and with indentations, later statements in the nest would move farther and farther to the right on the page. For easier-to-read code, Java programmers commonly place each `else` and its subsequent `if` on the same line.

With the new code in Figure 5-28, when the saleAmount is $5000, the expression `(saleAmount > 1000)` is `true` and the commRate becomes eight percent. When the saleAmount is not greater than $1000, the `else` statement executes and correctly sets the commRate to six percent.

The code shown in Figure 5-28 works, but it is somewhat inefficient. When the saleAmount is any amount over $500, either the first `if` sets commRate to eight percent for amounts over $500, or its `else` sets commRate to six percent for amounts over $500. The Boolean value tested in the next statement, `if(saleAmount <= 500)`, is always `false`. Rather than unconditionally asking `if(saleAmount <= 500)`, it's easier to use an `else`. If the saleAmount is not over $1000 and it is also not over $500, it must, by default, be less than or equal to 500. Figure 5-29 shows this improved logic and Figure 5-30 shows its code. Within a nested `if...else`, it is most efficient to ask the most likely question first. In other words, if you know that most saleAmount values are over 1000, compare saleAmount to that value first. If, however, you know that most saleAmounts are small, you should ask `if(saleAmount <= 500)` first.

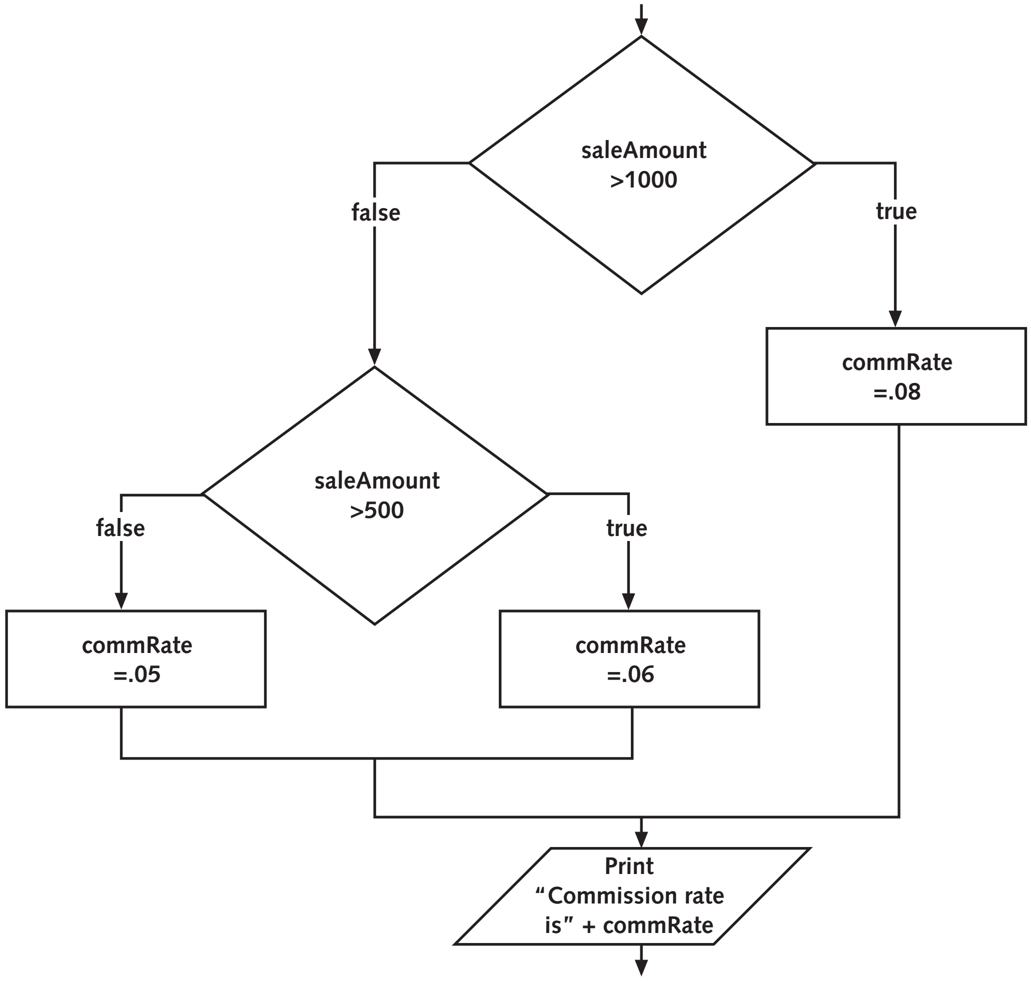


Figure 5-29 Improved OR logic

```
if(saleAmount >= 1000)
  commRate = .08;
else if(saleAmount > 500)
  commRate = .06;
else commRate = .05;
System.out.println("Commission rate is " + commRate);
```

Figure 5-30 Correct assignment of three commissions

Currently, the ChooseManager2 program identifies an event as a corporate event and assigns an appropriate manager and rate when the user enters C at the event-type prompt. When the user enters any other character, the event is considered to be private. Next you will improve the ChooseManager2 program so that if the user does not enter either C or P, an Event object with an "invalid" X-type is instantiated.

To improve the ChooseManager2 program:

1. If necessary, start your text editor and open the **ChooseManager2.java** file from the Chapter.05 folder on your Student Disk, and then change the class name to **ChooseManager3**.
2. Change the `if...else` structure that tests the eventType so that it becomes a nested `if...else` with three possibilities. When the user inputs anything other than C or P, display an error message and create an Event object with a code 'X' for the eventType and a rate of 0.0 for the minEventRate.

Tip: Remember that the Event constructor that you created earlier in this chapter requires both a character and a double argument.

```
if(eventType == 'C')
{
 System.out.println("Dustin Britt");
 anEvent = new Event(eventType, CORP_RATE);
}
else if(eventType == 'P')
{
 System.out.println("Carmen Lindsey");
 anEvent = new Event(eventType, PRI_RATE);
}
else
{
 System.out.println("Invalid entry!");
 anEvent = new Event('X',0.0);
}
```

3. Save the program as **ChooseManager3.java**, compile it, and then run the program several times to confirm that user responses of C or P result in valid Event objects, and that other responses result in an error message and an event of type X.

USING THE `switch` STATEMENT

By nesting a series of `if` and `else` statements, you can choose from any number of alternatives. For example, suppose you want to print a student's class year based on a stored number. Figure 5-31 shows the program.

```
if(year == 1)
  System.out.println("Freshman");
else if(year == 2)
  System.out.println("Sophomore");
else if(year == 3)
  System.out.println("Junior");
else if(year == 4)
  System.out.println("Senior");
else System.out.println("Invalid year");
```

Figure 5-31 Multiple alternatives

An alternative to the series of nested `if` statements is to use the `switch` statement. The **`switch` statement** is useful when you need to test a single variable against a series of exact integer or character values. The `switch` structure uses four keywords:

- `switch` starts the structure and is followed immediately by a test expression enclosed in parentheses
- `case` is followed by one of the possible values for the test expression and a colon
- `break` optionally terminates a switch structure at the end of each case
- `default` optionally is used prior to any action that should occur if the test variable does not match any case

Tip

You are not required to list the case values in ascending order as shown here. It is most efficient to list the most common case first, instead of the case with the lowest value.

```
switch(year)
{
  case 1:
    System.out.println("Freshman");
    break;
  case 2:
    System.out.println("Sophomore");
    break;
  case 3:
    System.out.println("Junior");
    break;
  case 4:
    System.out.println("Senior");
    break;
  default:
    System.out.println("Invalid year");
}
```

Figure 5-32 Sample `case` structure

Figure 5-32 shows the `case` structure used to print the four school years.

The `switch` structure shown in Figure 5-32 begins by evaluating the year variable shown in the `switch` statement. If the year is equal to the first `case` value, which is 1, then the statement that prints "Freshman" will execute. The `break` statement bypasses the rest of the `switch` structure, and execution continues with any statement after the closing curly brace of the `switch` structure.

If the year variable is not equivalent to the first `case` value of 1, then the next case value is compared, and so on. If the year variable does not contain the same value as any of the `case` statements, then the `default` statement or statements execute.

You can leave out the `break` statements in a `switch` structure. However, when you omit the `break`, if the program finds a match for the test variable, then all the statements within the `switch` statement from that point forward will execute. For example, if you omit each `break` statement in the code shown in Figure 5-32, when the year is 3, the first two cases will be bypassed, but "Junior," "Senior," and "Invalid year" all will print. You should intentionally omit the `break` statements if you want all subsequent cases to execute once the test variable is matched.

You are never required to use a `switch` structure; you can always achieve the same results with nested `if` statements. The `switch` structure is simply convenient to use when there are several alternate courses of action which depend on a single integer or character variable. Additionally, it makes sense to use `switch` only when there are a reasonable number of specific matching values to be tested. For example, if every sale amount from $1 to $500 requires a five percent commission, it is not reasonable to test every possible dollar amount using the following code:

```
switch(saleAmount)
{
 case 1:
  commRate = .05;
  break;
 case 2:
  commRate = .05;
  break;
 case 3:
  commRate = .05;
  break;
 // ...and so on for several hundred more cases
}
```

Because 500 different dollar values result in the same commission, one test—`if(saleAmount <= 500)`—is far more reasonable than listing 500 separate cases.

Next you will modify the ChooseManager3 program to account for a new type of event. Besides corporate and private, there will be special rates for nonprofit organizations. The entered code for nonprofit events will be N, and Robin Armanetti will be the manager assigned to these events. You will convert your nested `if` statements to a `switch` structure.

To convert the ChooseManager3 decision-making process to a switch structure:

1. Open the **ChooseManager3.java** file and change the class name to **ChooseManager4**. Position the insertion point at the end of the statement that declares the constant for PRI_RATE, press **[Enter]** to start a new line, and then type the constant for NON_PROF_RATE, **`static final double NON_PROF_RATE = 40.99;`**.

2. To the list of current prompts, add the following prompt to tell the user to enter N for nonprofit organization events:

   ```
   System.out.println("N for non-profit event");
   ```

3. Delete the `if...else` statements that presently are used to determine whether the user entered C or P, and then replace them with the following `switch` structure:

   ```
   switch(eventType)
   {
    case 'C':
     System.out.println("Dustin Britt");
     anEvent = new Event(eventType, CORP_RATE);
     break;
    case 'P':
     System.out.println("Carmen Lindsey");
     anEvent = new Event(eventType, PRI_RATE);
     break;
    case 'N':
     System.out.println("Robin Armanetti");
     anEvent = new Event(eventType, NON_PROF_RATE);
     break;
    default:
     System.out.println("Invalid entry!");
     anEvent = new Event('X',0.0);
   }
   ```

Tip

Remember from Chapter 2 that characters are stored as integers. That is why they are allowed as the case variables in a `switch` statement.

4. Save the file as **ChooseManager4.java**, compile, and test the program. Make sure the correct output appears when you enter C, P, N, or some other value as keyboard input.

USING THE CONDITIONAL AND NOT OPERATORS

Java provides one more way to make decisions. The **conditional operator** requires three expressions separated with a question mark and a colon, and it is used as an abbreviated version of the `if...else` structure. As with the `switch` structure, you are never required to use the conditional operator; it is simply a convenient shortcut. The syntax of the conditional operator is `testExpression ? true Result : false Result;`.

The first expression, testExpression, is a Boolean expression that is evaluated as `true` or `false`. If it is `true`, then the entire conditional expression takes on the value of the expression following the question mark (`trueResult`). If the value of the testExpression is `false`, then the entire expression takes on the value of `falseResult`. For example, suppose that you want to assign the smallest price to a sale item. Let the variable *a* be the advertised price and the variable *b* be the discounted price on the sale tag. The expression for assigning the smallest cost is `smallerNum = (a < b) ? a : b;`. When evaluating the expression `a < b`, where *a* is less than *b*, the entire conditional expression takes the value of *a*, which then is assigned to smallerNum. If *a* is not less than *b*, then the expression assumes the value of *b*, and *b* is assigned to smallerNum.

You use the **NOT operator**, which is written as the exclamation point (!), to negate the result of any Boolean expression. Any expression that evaluates as `true` becomes `false` when preceded by the NOT operator, and accordingly, any `false` expression preceded by the NOT operator becomes `true`.

For example, suppose a monthly car insurance premium is $200 if the driver is age 25 or younger, and $125 if the driver is age 26 or older. Each of the following `if` statements (which have been placed on single lines for convenience) correctly assigns the premium values:

```
if(age <= 25)  premium = 200;    else premium = 125;
if(!(age <= 25)) premium = 125;   else premium = 200;
if(age >= 26) premium = 125;   else premium = 200;
if(!(age >= 26)) premium = 200;    else premium = 125;
```

The statements with the NOT operator are somewhat harder to read, particularly because they require the double set of parentheses, but the result of the decision-making process is the same in each case. Using the NOT operator is clearer when the value of a Boolean variable is tested. For example, a variable initialized as `Boolean oldEnough = (age >= 25);` can become part of the relatively easy-to-read expression `if(!oldEnough)...`.

UNDERSTANDING PRECEDENCE

You learned in Chapter 2 that operations have higher and lower precedences. For example, within an arithmetic expression, multiplication and division are always performed prior to addition or subtraction. Table 5-2 shows the precedence of the operators you have used so far.

Table 5-1 Operator precedence for operators used so far

Precedence	Operator(s)	Symbol(s)
Highest	Multiplication, division	`* / %`
	Addition, subtraction	`+ -`
	Relational	`> < >= <=`
	Equality	`== !=`
	Logical AND	`&&`
	Logical OR	`||`
	Conditional	`?:`
Lowest	Assignment	`=`

In general, the order of precedence agrees with common algebraic usage. For example, in any mathematical expression such as `x = a + b`, the arithmetic is done first and the assignment is done last, as you would expect. The relationship of && and || might not be as obvious. Consider the program segment shown in Figure 5-33 and try to predict its output.

```
int tickets = 4;
int age = 40;
char gender = 'F';
if(tickets > 3 || age < 25 && gender == 'M')
  System.out.println("Do not insure");
if((tickets > 3 || age < 25) && gender == 'M')
  System.out.println("Bad risk");
```

Figure 5-33 Demonstrating AND and OR operator precedence

With the first `if` statement, the AND operator takes precedence over the OR operator, so `age < 25 && gender == 'M'` is evaluated first. The value is `false` because age is not less than 25 and gender is not 'M.' So the expression is reduced to "tickets > 3" or `false`. Because the value of the tickets variable is greater than 3, the entire expression is `true`, and "Do not insure" is printed.

Tip: Even though the AND operator is evaluated first in the expression `age < 25 && gender == 'M' || tickets > 3`, you can add extra parentheses, as in `(age < 25 && gender == 'M') || tickets > 3`. The outcome is the same, but the intent is clearer to someone reading your code.

In the second `if` statement shown in Figure 5-33, parentheses have been added so the OR operator is evaluated first. The expression `tickets > 3 || age < 25` is `true` because tickets is greater than 3. So the expression evolves to `true && gender == 'M'`. Because gender is not 'M,' the value of the entire expression is `false`, and the "Bad risk" statement does not print. The following two conventions are important to keep in mind:

- The order in which you use operators makes a difference.

- You can always use parentheses to change precedence or make your intentions clearer.

Chapter Summary

- An interactive program accepts values at run time. When you write interactive programs, it is often a good idea to echo the input so the user can confirm visually that the data entered is correct. The message that requests user input commonly is called a prompt.
- An exception is an error situation. You can let the compiler handle the problem by throwing the exception, or you can pass the error to the operating system. When the System.in.read() method accepts input from the keyboard it `throws Exception`. The method System.in.read() accepts a byte from the keyboard and returns an integer value.
- The JOptionPane component, part of the javax.swing package, can be used to create standard dialog boxes. These dialog boxes are small windows that ask a question, warn a user, or provide brief important user messages. As such, they provide a GUI interface to communicate with the user as opposed to the nonwindowed standard input and output methods presented thus far.
- Three standard dialog boxes of the JOptionPane class are: an input dialog box that prompts the user for text input, a message dialog box that displays a user message, and a confirm dialog box that asks the user a question using buttons for Yes, No, and Cancel responses.
- Making a decision involves choosing between two alternate courses of action based on some value within a program. You can use the `if` statement to make a decision based on a Boolean expression that evaluates as `true` or `false`. If the Boolean expression enclosed in parentheses within an `if` statement is `true`, then the subsequent statement or block will execute.
- A single-alternative `if` performs an action based on one alternative; a dual-alternative `if`, or `if...else`, provides the mechanism for performing one action when a Boolean expression evaluates as `true`. When a Boolean expression evaluates as `false`, a different action occurs.
- To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. Within an `if` or an `else` statement, you can code as many dependent statements as you need, including other `if` and `else` statements. Nested `if` statements are particularly useful when two conditions must be met before some action occurs.
- You can use the AND operator (&&) within a Boolean expression to determine whether two expressions are both `true`. You use the OR operator (||) when you want to carry out some action even if only one of two conditions is `true`.
- You use the `switch` statement to test a single variable against a series of exact integer or character values.

- The conditional operator requires three expressions, a question mark, and a colon, and it is used as an abbreviated version of the `if...else` statement.
- You use the NOT operator (!) to negate the result of any Boolean expression.
- Operator precedence makes a difference. You can always use parentheses to change precedence or make your intentions clearer.

Review Questions

1. Which of the following is typically used in a flowchart to indicate a decision?
   a. square
   b. rectangle
   c. diamond
   d. oval
2. A message that requests user input is commonly called a ________.
   a. coach
   b. prompt
   c. hint
   d. port
3. Which of the following is not a type of `if` statement?
   a. single-alternative `if`
   b. dual-alternative `if`
   c. double-alternative `if`
   d. nested `if`
4. Standard dialog boxes of the JOptionPane class include ________.
   a. InputDialog—Prompts the user for text input
   b. MessageDialog—Displays a user message
   c. ConfirmationDialog—Asks the user a question, with buttons for Yes, No, and Cancel responses
   d. all of the above
5. The JOptionPane class contains methods to create a(n) ________.
   a. input dialog box
   b. message dialog box
   c. confirm dialog box
   d. all the above

6. An easy way to create a Yes, No, Cancel dialog box is with the ______________.
   a. `showConfirmDialog()` method
   b. `showInputDialog()` method
   c. `showMessDialog()` method
   d. none of the above
7. A decision is based on a(n) ______________ value.
   a. Boolean
   b. absolute
   c. definitive
   d. convoluted
8. The value of `(4 > 7)` is ______________.
   a. 4
   b. 7
   c. `true`
   d. `false`
9. Assuming the variable q has been assigned the value 3, which of the following statements prints `XXX`?
   a. `if(q > 0) System.out.println("XXX");`
   b. `if(q > 7); System.out.println("XXX");`
   c. Both of the above statements print `XXX`.
   d. Neither of the above statements prints `XXX`.
10. What is the output of the following code segment?

```
t = 10;
if(t > 7)
{
 System.out.println("AAA");
 System.out.println("BBB");
}
```

   a. `AAA`
   b. `BBB`
   c. `AAA`
      `BBB`
   d. nothing

11. When you code an `if` statement within another `if` statement, as in `if(a > b) if(c > d) x = 0;`, then the `if` statements are ________________.
    a. notched
    b. nestled
    c. nested
    d. sheltered
12. The operator that combines two conditions into a single Boolean value that is true when both of the conditions are `true` is ________________.
    a. $$
    b. !!
    c. ||
    d. &&
13. The operator that combines two conditions into a single Boolean value that is true when at least one of the conditions is `true` is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
    a. $$
    b. !!
    c. ||
    d. &&
14. Assuming a variable f has been initialized to 5, which of the following statements sets g to 0?
    a. `if(f > 6 || f == 5) g = 0;`
    b. `if(f < 3 || f > 4) g = 0;`
    c. `if(f >= 0 || f < 2) g = 0;`
    d. All of the above statements set g to 0.
15. Which if the following groups has the lowest operator precedence?
    a. Relational
    b. Equality
    c. Addition
    d. Logical OR
16. You can use the ________________ statement to terminate a `case` in a `switch` structure.
    a. `switch`
    b. `end`
    c. `case`
    d. `break`

17. The `switch` argument within a `switch` structure requires a(n) ________________.
    a. integer value
    b. character value
    c. double value
    d. integer or character value
18. Assuming a variable w has been assigned the value 15, then the statement `w == 15 ? x = 2 : x = 0;` assigns ________________.
    a. 15 to w
    b. 2 to x
    c. 0 to x
    d. nothing
19. Assuming a variable y has been assigned the value 6, then the value of `!(y < 7)` is ________________.
    a. 6
    b. 7
    c. `true`
    d. `false`
20. Assuming `a = 5` and `b = 9`, then the value of `a > 0 &&  b < 10 || b > 1` is ________________.
    a. 5
    b. 9
    c. `true`
    d. `false`

EXERCISES

In the following exercises, save each program that you create in the Chapter.05 folder on your Student Disk.

1. a. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Save the program as **Password.java** in the Chapter.05 folder on your Student Disk.

   b. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Test the first character. If it is B, issue a message that the password is valid; otherwise issue a message that the password is not valid. Save the program as **Password.java** in the Chapter.05 folder on your Student Disk.

c. Write a program that prompts the user for a four-character password, accepts four characters, and then echoes the characters to the screen. Test all four characters. If the characters spell BOLT, then issue a message that the password is valid; otherwise issue a message that the password is not valid. Save the program as **PasswordC.java** in your Chapter.05 folder on your Student Disk.

2. a. Write a program for a furniture company. Ask the user to choose P for pine, O for oak, or M for mahogany. Show the price of a table manufactured with the chosen wood. Pine tables cost $100, oak tables cost $225, and mahogany tables cost $310. Save the program as **Furniture.java** in the Chapter.05 folder on your Student Disk.

   b. Add a prompt to the program you wrote in Exercise 2a to ask the user to specify a large (L) or a small (S) table. Add $35 to the price of any large table. Save the program as **FurnitureSizes.java** in the Chapter.05 folder on your Student Disk.

3. Write a program for a college's admissions office. Create variables to store a student's numeric high school grade point average (for example, 3.2) and an admission test score. Print the message "Accept" if the student has any of the following:

   - A grade point average of 3.0 or above and an admission test score of at least 60
   - A grade point average below 3.0 and an admission test score of at least 80

   If the student does not meet either of the qualification criteria, print "Reject." Save the program as **Admission.java** in the Chapter.05 folder on your Student Disk.

4. Write a program that stores an hourly pay rate and hours worked. Compute gross pay (hours times rate), withholding tax, and net pay (gross pay minus withholding tax). Withholding tax is computed as a percentage of gross pay based on the following:

Gross Pay	Withholding Percentage
Up to and including 300.00	10
300.01 and up	12

   Save the program as **Payroll.java** in the Chapter.05 folder on your Student Disk.

5. a. Write a program that stores two integers and allows the user to enter a character. If the character is A, add the two integers. If it is S, subtract the second integer from the first; if it is M, multiply the integers. Display the results of the arithmetic. Save the program as **Calculate.java** in the Chapter.05 folder on your Student Disk.

   b. Modify the Calculate program so the user also can enter a D for divide. If the second number is zero, then display an error message; otherwise divide the first number by the second and display the results. Save the program as **Calculate2.java** in the Chapter.05 folder on your Student Disk.

6. a. Write a program for a lawn-mowing service. The lawn-mowing season lasts 20 weeks. The weekly fee for mowing a lot under 400 square feet is $25. The fee for a lot 400 square feet or more but under 600 square feet is $35 per week. The fee for a lot 600 square feet or over is $50 per week. Store the values in

the length and width variables and then print the weekly mowing fee, as well as the seasonal fee. Save the program as **Lawn.java** in the Chapter.05 folder on your Student Disk.

b. To the Lawn program created in 6a, add a prompt that asks the user whether the customer wants to pay A) once, B) twice, or C) 20 times per year. If the user enters A for once, the fee for the season is simply the seasonal total. If the customer requests two payments, each payment is half the seasonal fee plus a $5 service charge. If the user requests 20 separate payments, add a $3 service charge per week. Print the payment amount. Save the program in the Chapter.05 folder on your Student Disk as **Lawn2**.

7. a. Write a program that compares your checking account balance with your savings account balance (two doubles). Assign values to both variables, compare them, and then display either "Checking is higher" or "Checking is not higher". Save the program as **Balance.java** in the Chapter.05 folder on your Student Disk.

   b. Write a program so that it compares your checking account balance and your savings account balance to less than zero. If both statements are true, then display the message "Both accounts in the red". If the first balance is less than the second balance, and the first balance is greater than or equal to zero, then display the message "Both accounts in the black". Save the program as **Balance2.java** in the Chapter.05 folder on your Student Disk.

8. Write a program that asks a user to input an initial. Display the full name of an employee who matches the initial: A is Armando, B is Bruno, and Z is Zachary. All other entries should cause a "No such employee" message to display. Save the program as **PickEmployee.java** in the Chapter.05 folder on your Student Disk.

9. Write a program that asks the user to type a digit from the keyboard. If the character entered is not a digit, display an error message. Save the program as **GetDigit.java** in the Chapter.05 folder on your Student Disk.

10. Write a program that stores an IQ score. If the score is a number less than 0 or greater than 200, issue an error message; otherwise, issue an "above average", "average", or "below average" message for scores over, at, or under 100, respectively. Save the program as **IQ.java** in the Chapter.05 folder on your Student Disk.

11. Write a program for a college's admissions office. Create variables that store a numeric high school grade point average (for example, 3.2) and an admission test score. Print the message "Accept" if the student has any of the following:

    - A grade point average of 3.6 or above and an admission test score of at least 60
    - A grade point average of 3.0 or above and an admission test score of at least 70
    - A grade point average of 2.6 or above and an admission test score of at least 80
    - A grade point average of 2.0 or above and an admission test score of at least 90

    If the student does not meet any of the qualifications, print "Reject." Save the program as **Admission2.java** in the Chapter.05 folder on your Student Disk.

12. Write a program that stores an employee's hourly pay rate and hours worked. Compute gross pay (hours times rate), withholding tax, and net pay (gross pay minus withholding tax). Withholding tax is computed as a percentage of gross pay based on the following:

Gross Pay	Withholding Percentage
0 to 300.00	10
300.01 to 400.00	12
400.01 to 500.00	15
500.01 and over	20

Save the program as **Payroll2.java** in the Chapter.05 folder on your Student Disk.

13. Write a program that recommends a pet for a user based on the user's lifestyle. Prompt the user to enter whether he or she lives in an apartment, house, or dormitory (A, H, or D) and the number of hours the user is home during the average day. The user will select an hour category from a menu: A) 18 or more; B) 10 to 17; C) 8 to 9; D) 6 to 7; or E) 0 to 5. Print your recommendation based on the following:

Residence	Hours Home	Recommendation
House	18 or more	Pot bellied pig
House	10 through 17	Dog
House	Fewer than 10	Snake
Apartment	10 or more	Cat
Apartment	Fewer than 10	Hamster
Dormitory	6 or more	Fish
Dormitory	Fewer than 6	Ant farm

Save the program as **PetAdvice.java** in the Chapter.05 folder on your Student Disk.

14. Write a program that declares two ints named myNumberOfSiblings and yourNumberOfSiblings. Display an appropriate message to indicate whether your friend has more, fewer, or the same number of siblings as you. Display the number of siblings whether the `if` statement is `true` or not. Save the program as **Siblings.java** in the Chapter.05 folder on your Student Disk.

15. Write a program that compares the number of college credits you have earned with the number of college credits earned by a classmate or friend. Display an appropriate message to indicate whether your classmate has earned more, fewer, or the same number of credits as you. Display the number of college credits whether the `if` statement is `true` or not. Save the program as **Credits.java** in the Chapter.05 folder on your Student Disk.

16. Write a program that displays a menu of three items in a store, with a price for each item. Include characters a, b, and c so the user can select a menu item. Prompt the user to choose an item using the character that corresponds to the item. After the user makes the first selection, show a prompt to ask if another selection will be made. The user should respond Y or N to this prompt (for yes or no). If the user types N,

display the cost of the item. If the user types Y, allow the user to select another item and then display the total cost of the two items. Use the `switch` statement to check the menu selection. Save the program as **Store.java** in the Chapter.05 folder on your Student Disk.

17. Write a program using input, message, and dialog boxes to accept users' first and last names. Display the first and last names as they are entered. Prompt the user to verify that the names entered are correct. Use an `if...else` structure with a confirm dialog box to determine if the user clicks the No or Cancel options. Print appropriate messages if the user clicks No or cancel. Exit the program. Save the program as **DemoDialog.java** in the Chapter.05 folder on your Student Disk.
18. Write a program using the input dialog box that asks the question. "What is your zip code?" and displays "Enter Your Zip Code" in the title bar. The type of the dialog box should be QUESTION_MESSAGE. The program should display the zip code entered. Save the program as **ZipDialog.java** in the Chapter.05 folder on your Student Disk.
19. Write a program using the confirm dialog box that asks "Error reading file. Do you want to try again?". The confirm dialog box title should read "File input error". The user should see an error message icon and have a Yes or No option. Save the program as **ErrorDialog.java** in the Chapter.05 folder on your Student Disk.
20. Write a message dialog box that displays the message, "This program has finished installing." The message dialog box title should read "Installing Program". Save the program in the Chapter.05 folder on your Student Disk as **InstallDialog**.
21. Each of the following files in the Chapter.05 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, save DebugFive1.java as FixDebugFive1.java.
    a. DebugFive1.java
    b. DebugFive2.java
    c. DebugFive3.java
    d. DebugFive4.java

Case Project

Widget Company runs a small factory that makes several types of nuts and bolts. They employ factory workers that are paid one of three hourly rates depending on skill level: (a) 7.00, (b) 10.00, or (c) 12.00. Each factory worker can work: (a) 40 hours, (b) 45 hours, and (c) 50 hours per week. All hours over 40 are paid at double time.

Jack Smith, the factory manager, wants you to write an interactive Java payroll program that will calculate the gross pay for a factory worker. Hours worked and hourly pay rate are to be entered from the keyboard. Once the figures are entered for an employee, the program prints out: (1) the hours worked, (2) the hourly pay rate, (3) the regular pay for 40 hours, and (4) the overtime pay. The class name is pay.java. Save the program as **Pay.java** in the Chapter.05 folder on your Student Disk.

CHAPTER

6

LOOPING

In this chapter, you will:

- ♦ Learn about the loop structure
- ♦ Use a `while` loop
- ♦ Use shortcut arithmetic operators
- ♦ Use a `for` loop
- ♦ Learn how and when to use a `do...while` loop
- ♦ Learn about nested loops

Still having problems?" asks Lynn Greenbrier, who is watching intently as you code a new program.

"My programs can finally respond to questions and make choices," you reply. "But I still feel like something is missing."

"It's probably because you want to do the same task more than once without writing the procedure more than one time," Lynn suggests. "Let me introduce you to the control structure called looping."

Previewing the EvenInt Program

To preview the EvenInt program:

1. In your text editor, open the **Chap6EvenInt.java** file from the Chapter.06 folder on your Student Disk and examine the code. This file contains a definition for a class that loops through all integers from 1 to 100, and during looping prints each integer and all the integers by which it can be evenly divided. You will create a similar class file in this chapter.
2. At the command line, compile the **Chap6EvenInt.java** file using the command **`javac Chap6EvenInt.java`**.
3. Execute the program by typing the command **`java Chap6EvenInt`**. Press **[Enter]** repeatedly to see the evenly divisible numbers. They are displayed twenty at a time.

Learning About the Loop Structure

If making decisions is what makes programs seem smart, then looping is what makes programs seem powerful. A **loop** is a structure that allows repeated execution of a block of statements. Within a looping structure, a Boolean expression is evaluated. If it is `true`, then a block of statements, called the **loop body**, executes and the Boolean expression is evaluated again. As long as the expression is `true`, the statements in the loop body continue to execute. When the Boolean evaluation is `false`, the loop ends. Figure 6-1 shows a diagram of the logic of a loop.

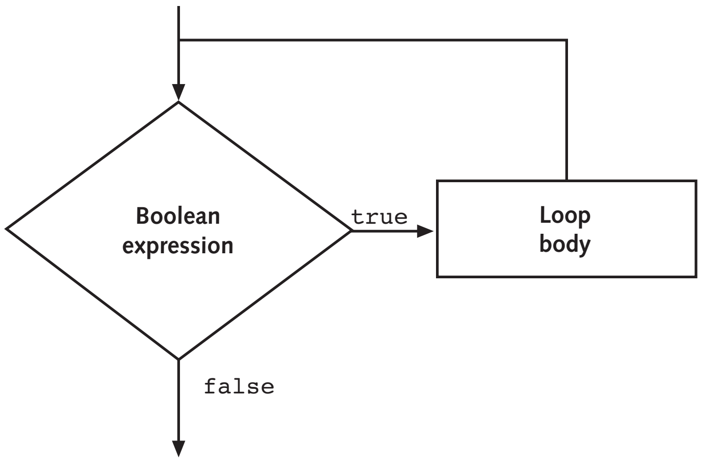


Figure 6-1 Logic of a loop

Tip

One execution of any loop is called an iteration.

USING A while LOOP

You can use a **while loop** to execute a body of statements continually as long as the Boolean expression continues to be true. A while loop consists of the keyword while followed by a Boolean expression within parentheses, followed by the body of the loop, which can be a single statement or a block of statements surrounded by curly braces.

You can use a while loop when you need to perform a task a predetermined number of times. A loop that executes a specific number of times is a **definite loop** or a counted loop. To write a definite loop, you initialize a loop control variable, and while the loop control variable does not pass a limit, you continue to execute the body of the while statement. You must include in the body of the while loop a statement that alters the loop control variable. For example, the program segment shown in Figure 6-2 prints the series of integers 1 through 10. The variable val starts the loop holding a value of 1, and while the value remains under 11, the val continues to print and be incremented.

```
int val = 1;
while (val < 11)
{
  System.out.println(val);
  ++val;
}
```

Figure 6-2 Printing the integers 1 through 10 with a while loop

The code shown in Figure 6-3 causes the message "Hello" to display (theoretically) forever because there is no code to end the loop. A loop that never ends is called an **infinite loop**.

Tip

An infinite loop might not actually execute infinitely. Eventually the computer memory will be exhausted (literally and figuratively) and execution will stop. Also, it's possible that the processor has a time-out feature. Either way, and depending on your system, quite a bit of time could pass before the loop stops running.

```
while (4 > 2)
  System.out.println("Hello");
```

Figure 6-3 An infinite loop

In Figure 6-3, the expression `4 > 2` evaluates to `true`. You obviously never need to make such an evaluation, but if you do so in this `while` loop, the body of the loop is entered and "Hello" displays. Next, the expression is evaluated again. The expression `4 > 2` is still `true`, so the body is entered again. "Hello" displays repeatedly; the loop never finishes because `4 > 2` is never `false`.

It is a bad idea to intentionally write an infinite loop. However, even experienced programmers write them by accident. So, before you start writing any loops, it is good to know how to break out of an infinite loop in case you find yourself in the midst of one. If you think your program is in a loop, you can press and hold [Ctrl], and then press C or [Break].

To prevent a `while` loop from executing infinitely, four separate actions must occur:

1. A named loop control variable is initialized to a starting value.
2. The loop control variable is tested in the `while` statement.
3. If the test expression is `true`, the body of the `while` statement must take some action that alters the value of the loop control variable.
4. The action of the body statement(s) must eventually change the value of the control variable so that the test of the `while` statement evaluates to `false`.

All of these conditions are met by the example in Figure 6-4. First, a loop control variable loopCount is named and set to a value of 1. Second, the statement `while(loopCount < 3)` is tested. Third, the loop body is executed because the loopControl variable loopCount is less than 3. Note that the loop body shown in Figure 6-4 consists of two statements made into a block by their surrounding curly braces. The first statement prints "Hello," and then the second statement adds one to loopCount. The next time loopCount is evaluated, it is 2. It is still less than 3, so the loop body executes again. "Hello" prints a second time, and loopCount becomes 3. Fourth, because the expression `loopCount < 3` now evaluates to `false`, the loop ends. Program execution then continues with any subsequent statements.

Tip

To an algebra student, a statement such as `loopCount = loopCount + 1` looks wrong—a value can never be one more than itself. In programming, however, `loopCount = loopCount + 1;` takes the value of loopCount, adds one to it, and then assigns the new value back into loopCount.

```
loopCount = 1;
while (loopCount < 3)
{
  System.out.println("Hello");
  loopCount = loopCount + 1;
}
```

Figure 6-4 A simple loop that executes twice

If the curly braces are omitted from the code shown in Figure 6-4, the `while` loop stops at the end of the "Hello" statement. Adding one to the loopCount is no longer part of a block that contains the loop, so an infinite situation is created.

Also, if a semicolon is mistakenly placed at the end of the partial statement `while (loopCount < 3);`, the loop is also infinite. This loop has an **empty body**, or a body with no statements in it, so the Boolean expression is evaluated, and because it is `true`, the loop body is entered. Because the loop body is empty, no action is taken, and the Boolean expression is evaluated again. Nothing has changed, so it is still `true`, the empty body is entered, and the infinite loop continues.

It is very common to alter the value of a loop control variable by adding one to it, or **incrementing** the variable. However, not all loops are controlled by adding one. The loop shown in Figure 6-5 prints "Hello" twice, just as the loop in Figure 6-4 does, but its loop is controlled by subtracting 1 from a loop control variable, or **decrementing** it.

```
loopCount = 10;
while (loopCount > 8)
{
  System.out.println("Hello");
  loopCount = loopCount - 1;
}
```

Figure 6-5 Another simple loop that executes twice

In the program segment shown in Figure 6-5, the variable loopCount begins with a value of 10. The loopCount is greater than 8, so the loop body prints "Hello" and decrements loopCount so it becomes 9. The Boolean expression in the `while` loop is tested again. Because 9 is more than 8, "Hello" prints again and loopCount becomes 8. Now loopCount is not greater than 8, so the loop ends.

The possibilities are endless. Figure 6-6 shows the loopCount being increased by 5, and the results are still the same—the loop prints "Hello" twice. In general, you should not use such unusual methods because they simply make a program confusing. The clearest and best method is to start loopCount at 0 or 1, and continue while it is less than 2 or 3, incrementing by one each time through the loop.

```
loopCount = 30;
while (loopCount < 40)
{
  System.out.println("Hello");
  loopCount = loopCount + 5;
}
```

Figure 6-6 A third simple loop that executes twice

Within a loop, you are not required to alter the loop control variable by adding to it or subtracting from it. When you write a loop that is controlled by an arithmetic result, you need to know how many times you want a loop to execute. Often, the value of a loop control variable is not altered by arithmetic, but instead is altered by user input. For example, perhaps you want to continue performing some task as long as the user indicates a desire to continue. In that case, when you write the program you do not know whether the loop will be executed two times, 200 times, or at all.

Tip

Unlike a loop that you program to execute a fixed number of times, a loop controlled by the user is a type of indefinite loop because you don't know how many times it will eventually loop.

Consider a simple program in which you display a bank balance and ask the user whether he or she wants to see what the balance will be after interest has accumulated for each year. Each time the user indicates a desire to continue, an increased balance appears, reflecting one more year of accumulated interest. When the user finally indicates a desire to exit, the program ends. The program appears in Figure 6-7.

```
public class LoopingBankBal
{
  public static void main(String[] args) throws Exception
  {
    double bankBal = 1000;
    double intRate = 0.04;
    char response;
    System.out.println
      ("Do you want to see your balance? Y or N");
    response = (char)System.in.read();
    System.in.read(); System.in.read();
        // Absorbs Enter key
    while (response == 'Y')
    {
      System.out.println("Bank balance is " + bankBal);
      bankBal = bankBal + bankBal * intRate;
      System.out.println
        ("Do you want to see next year's balance? Y or N");
      response = (char)System.in.read();
      System.in.read(); System.in.read();
        // Absorbs Enter key
    }
    System.out.println("Have a nice day!");
  }
}
```

Figure 6-7 LoopingBankBal program

The program shown in Figure 6-7 continues to display bank balances while the response is Y. It could also be written to display while the response is not N, as in `while (response != 'N')....` A value that a user must supply to stop a loop is called a **sentinel value**.

The program shown in Figure 6-7 contains three variables: a bank balance, an interest rate, and a response. The program asks the user, "Do you want to see your balance?" and reads the response. Recall that the second and third read() statements are required to accept [Enter] that is typed after the Y or N entry. The loop in the program begins with `while(response == 'Y')`. If the user types any response other than Y, then the loop body never executes; instead, the next statement to execute is the "Have a nice day!" statement at the bottom of the program. However, if the user enters Y, then all five statements within the loop body will execute. The current balance will display, and then the program increases the balance by the interest rate value. The program then prompts the user to type Y or N, and two characters are entered—the response and [Enter]. The loop ends with a closing curly brace, and program control returns to the top of the loop, where the Boolean expression in the `while` loop is tested again. If the user typed Y, then the loop is entered and the increased bankBal value that was calculated is finally displayed.

Next you will improve the ChooseManager4 program so the user cannot make an invalid choice for the type of event.

To improve the ChooseManager4 program:

1. If necessary, start your text editor and then open the **ChooseManager4.java** text file from the Chapter.06 folder on your Student Disk.
2. Position the insertion point to the right of the statement that reads in the character for the event type, press **[Enter]** to start a new line, and then type the beginning of the `while` loop that will continue to execute while the user's entry is not one of the three allowed event types:

   ```
   while (eventType != 'C' && eventType != 'P'
   && eventType != 'N')
   ```

3. On a new line, type the opening curly brace for the `while` loop, press **[Enter]**, and then type the two statements that will accept the [Enter] key carriage return and line feed remaining from the user's data entry:

   ```
   System.in.read(); System.in.read();
   ```

4. On a new line, display the following message so the user knows the data entry was invalid:

   ```
   System.out.println("Entry must be C or P or N!");
   ```

5. On another new line, read in the eventType value again by typing:

   ```
   eventType = (char)System.in.read();
   ```

6. On a new line, type the closing curly brace for the `while` loop. After making it through the `while` loop you just added, the program is guaranteed that the eventType is C, P, or N, so you can make the nonprofit event the default case.
7. Change the statement `case 'N':` to **`default:`**, and then delete the three lines that represent the current default case—the existing `default` keyword and the `System.out.printIn("Invalid Entry");` statement that follow it.
8. Save, compile, and test the program. No matter how many invalid entries you make, the program will continue to prompt you until you enter C, P, or N.

Using Shortcut Arithmetic Operators

It is common to increase the value of a variable in a program. As you saw in the last section, many loops are controlled by continually adding one to some variable, or incrementing, as in `count = count + 1;`. Similarly, in the looping bank balance program, the program increased a bank balance by an interest amount with the statement `bankBal = bankBal + bankBal * intRate;`. In other words, the bank balance became its old value *plus* a new interest amount; this process is known as **accumulating**.

Because increasing a variable is so common, Java provides you with several shortcuts for incrementing and accumulating. The statement `count += 1;` is identical in meaning to `count = count + 1`. The `+=` adds and assigns in one operation. Similarly, `bankBal += bankBal * intRate;` increases a bankBal by a calculated interest amount.

When you want to increase a variable's value by exactly one, you can use two other shortcut operators—the **prefix ++** and the **postfix ++**. To use a prefix ++, you type two plus signs before the variable name. The statement `someValue = 6;`followed by `++someValue;` results in someValue holding 7—one more than it held before you applied the ++. To use a postfix ++, you type two plus signs just after a variable name. The statements `anotherValue = 56; anotherValue++;` result in anotherValue containing 57.

Tip

You can use the prefix ++ and postfix ++ with variables, but not with constants. An expression such as `++84;` is illegal because an 84 must always remain an 84. However, you can create a variable as `int val = 84` and then write `++val;` or `val++;` to increase the variable's value.

The prefix and postfix increment operators are unary operators because you use them with one value. Most arithmetic operators, such as those used for addition and multiplication, are **binary** operators—they operate on two values.

When you simply want to increase a variable's value by one, there is no difference between using the prefix and postfix increment operators. Each operator results in increasing the variable by one. However, these operators do function differently. When you use the prefix ++, the result is calculated and stored, and then the variable is used. For example, if `b = 4;` and `c = ++b;`, this results in both b and c holding the value 5. When you use the postfix ++, the variable is used and then the result is calculated and stored. For example, if `b = 4;` and then `c = b++;`, 4 will be assigned to c, and then after the assignment, b is increased and takes the value 5. In other words, if `b = 4`, then the value of `b++` is also 4, but after the statement is completed, the value of b will be 5. If `d = 8` and `e = 8`, both `++d == 9` and `e++ == 8` are `true` expressions.

Similar logic can be applied when you use the prefix and the postfix decrement operators. For example, if `b = 4;` and `c = b—;`, 4 will be assigned to c, and then after the assignment, b is decreased and takes the value 3. If `b = 4;` and `c = —b;`, b will be decreased to 3 and 3 will be assigned to c.

Next you will add a program that demonstrates how prefix and postfix operators are used in incrementation, and how incrementing affects the expressions that contain these operators.

To demonstrate the effect of the prefix and postfix increment operators:

1. Open a new text file and begin a demonstration class named DemoIncrement by typing:

```
 public class DemoIncrement
{
 public static void main(String[] args) throws Exception
 {
```

2. On a new line, add a variable v and assign it a value of 4. Then declare a variable named plusPlusV and assign it a value of `++v` by typing:

```
int v = 4;
int plusPlusV = ++v;
```

   The last statement, `int plusPlusV = ++v;`, will increase v to 5, so before declaring a vPlusPlus variable to which you assign `v++`, reset v to 4 by typing:

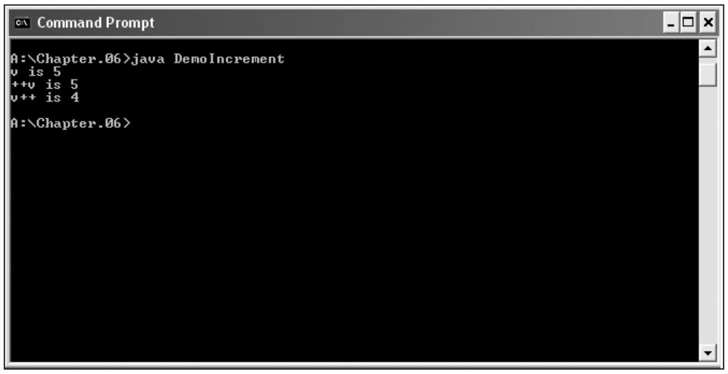
```
v = 4;
int vPlusPlus = v++;
```

4. Add the following statements to print the three values:

```
System.out.println("v is " + v);
System.out.println("++v is " + plusPlusV);
System.out.println("v++ is " + vPlusPlus);
```

5. Add the closing curly brace for the main() method and the closing curly brace for the DemoIncrement class.

6. Save the file as **DemoIncrement.java** in the Chapter.06 folder on your Student Disk. Compile and execute the program. Your output should look like Figure 6-8.



```
A:\Chapter.06>java DemoIncrement
v is 5
++v is 5
v++ is 4

A:\Chapter.06>
```

Figure 6-8 Output of the DemoIncrement program

To illustrate how comparisons are made, add a few more variables to the DemoIncrement program.

7. Position the insertion point to the right of the last println() statement, and then press **[Enter]** to start a new line.
8. Add three new integer variables and two new Boolean variables. The first Boolean variable compares `++w` to y; the second Boolean variable compares `x++` to y:

```
int w = 17, x = 17, y = 18;
boolean compare1 = (++w == y);
boolean compare2 = (x++ == y);
```

9. Add the following statements to display the values stored in the compare variables:

```
System.out.println("First compare is " + compare1);
System.out.println("Second compare is " + compare2);
```

10. Save, run, and compile the program.

Besides using the shortcut operator `+=`, you can use `-=`, `*=`, and `/=`. Each of these operators is used to perform the operation and assign the result in one step. For example, `balanceDue -= payment` subtracts payment from balanceDue and assigns the result to balanceDue.

USING A for LOOP

A **for loop** is a special loop that is used when a definite number of loop iterations is required. Recall that a while loop, also a definite loop type, has a variable number of iterations and may not execute the body of its loop at all if the initial test condition is false. The for loop is used when one or more loop iterations are known to be needed. You will need definite loops frequently when you write programs, and the for loop provides you with a shorthand notation that you can use to create those definite loops. When you use a for loop, you can indicate the starting value for the loop control variable, the test condition that controls loop entry, and the expression that alters the loop control variable all in one convenient place.

You begin a for loop with the keyword for followed by a set of parentheses. Within the parentheses there are three sections separated by exactly two semicolons. The three sections are usually used for the following:

- Initializing the loop control variable
- Testing the loop control variable
- Updating the loop control variable

The body of the for statement follows the parentheses. As with an if statement or a while loop, you can use a single statement as the body of a for loop, or you can use a block of statements enclosed in curly braces. The for statement shown in Figure 6-9 produces the same output as the while statement shown previously in Figure 6-2—it prints the integers 1 through 10.

```
for(int val = 1; val < 11; ++val)
{
  System.out.println(val);
}
```

Figure 6-9 Printing the integers 1 through 10 with a for loop

Tip

You did not have to declare the variable val within the for statement. If you declared val earlier in the program block as `int val;` before the for statement begins, then the for statement would be `for(val = 1; val < 11; ++val)`. In other words, the for statement does not need to declare a variable; it can simply give a starting value to a previously declared variable. Generally, it is not considered to be good practice to reuse variables in for statements.

Within the parentheses of the for statement shown in Figure 6-9, the first section prior to the first semicolon declares a variable named val and initializes it to 1. The program will execute this statement once, no matter how many times the body of the for loop executes.

After the initialization expression executes, program control passes to the middle, or test section, of the `for` statement. If the Boolean expression found there evaluates to `true`, then the body of the `for` loop is entered. In the program segment shown in Figure 6-9, val is set to 1, so when `val < 11` is tested, it evaluates to `true`. The loop body prints the val.

After the loop body executes, the final one-third of the `for` loop executes, and val is increased to 2. Following the third section, program control returns to the second section, where val is compared to 11 a second time. Because val is still less than 11, the body executes: val (now 2) prints, and then the third, altering portion of the `for` loop executes again. The variable val increases to 3, and the `for` loop continues.

Eventually, when val is not less than 11 (after 1 through 10 have printed), the `for` loop ends, and the program continues with any statements that follow the `for` loop.

Although the three sections of the `for` loop are most commonly used for initializing, testing, and incrementing, you can also perform the following tasks:

- Initialization of more than one variable by placing commas between the separate statements, as in `for(g = 0, h = 1; g < 6; ++g)`
- Performance of more than one test, as in `for(g = 0; g < 3 && h > 1; ++g)`
- Decrementation or performance of some other task, as in `for(g = 5; g >= 1; —g)`
- Leaving one or more portions of the `for` loop empty, although the two semicolons are still required as placeholders

Usually you should use the `for` loop for its intended purpose—a shorthand way of programming a definite loop. Occasionally, you will encounter a `for` loop that contains no body, such as `for(x = 0; x < 100000; ++x);`. This kind of loop exists simply to use time—for instance, when a brief pause is desired during program execution.

Tip: Java also contains a built-in method to pause program execution. The sleep() method is part of the Thread class in the java.lang package.

Learning How and When to Use a `do...while` Loop

With all the loops you have written so far, the loop body might execute many times, but it is also possible that the loop will not execute at all. For example, recall the bank balance program that displays compound interest, part of which is shown in Figure 6-10.

The program segment begins with the user prompt, "Do you want to see your balance? Y or N". If the user does not type Y, the loop body never executes. The **while** loop checks a value at the "top" of the loop before the body has a chance to execute. Sometimes you might need a loop body to execute at least one time. If so, you want to write a loop that checks at the "bottom" of the loop after the first iteration. The **do...while loop** checks the bottom of the loop after one repetition has occurred.

```
System.out.println("Do you want to see your balance? Y or N");
response = (char)System.in.read();
System.in.read(); System.in.read(); // Absorbs Enter key
while (response == 'Y')
{
  System.out.println("Bank balance is " + bankBal);
  bankBal = bankBal + bankBal * intRate;
  System.out.println
    ("Do you want to see next year's balance? Y or N");
  response = (char)System.in.read();
  System.in.read();  // Absorbs Enter key
}
```

Figure 6-10 Part of the bank balance program with a while loop

Figure 6-11 shows a **do...while** loop for the bank balance program. The loop starts with the keyword **do**. The body of the loop follows and is contained within curly braces. The bankBal variable is output before the user has any option of responding. At the end of the loop, the user is prompted, "Do you want to see next year's balance? Y or N". Now the user has the option of seeing more balances, but the first prompt was unavoidable. The user's response is checked at the bottom of the loop. If it is Y, then the loop repeats.

```
do
{
  System.out.println("Bank balance is " + bankBal);
  bankBal = bankBal + bankBal * intRate;
  System.out.println
    ("Do you want to see next year's balance? Y or N");
  response = (char)System.in.read();
  System.in.read(); System.in.read(); // Absorbs Enter key
} while (response == 'Y');
```

Figure 6-11 Part of the bank balance program with a do...while loop

In any situation where you want to loop, you are never required to use a **do...while** loop. Within the bank balance example, you could simply unconditionally display the bank balance once, prompt the user, and then start a **while** loop that might not be entered. However, when you know you want to perform some task at least one time, the **do...while** loop is convenient.

LEARNING ABOUT NESTED LOOPS

Just as `if` statements can be nested, so can loops. You can place a `while` loop within a `while` loop, a `for` loop within a `for` loop, a `while` loop within a `for` loop, or use any combination you can think of.

For example, suppose you want to find all the numbers that divide evenly into 100. You can write a `for` loop that sets a variable to 1 and increments it to 100. Each of the 99 times through the loop, if 100 is evenly divisible by the number (that is, if 100%num is equivalent to 0), then the program prints the number. Next you will write a program that determines all the integers that divide evenly into 100.

Tip

To find all the numbers that divide evenly into 100, you actually have to test divisors only through 50. You cannot evenly divide any number by a number that is more than half of the original number.

To write a program that finds the values that divide evenly into 100:

1. Open a new text file.
2. Begin the program named EvenInt by typing the following code to declare an integer variable named num:

```
public class EvenInt
{
 public static void main(String[] args)
 {
  int num;
```

3. Type a statement that explains the purpose of the program:

```
System.out.print("100 is evenly divisible by ");
```

4. Write the `for` loop that varies num from 1 to 99. With each iteration of the loop, test whether 100%num is 0. If you divide 100 by a number and there is no remainder, then the number goes into 100 evenly.

```
for(num = 1; num < 100; ++num)
 if(100%num == 0)
  System.out.print(num + "  ");
  // Print the number and two spaces
```

5. Add an empty println() statement to advance the insertion point to the next line by typing **`System.out.println();`**.
6. Type the closing curly braces for the main() method and the EvenInt class.
7. Save the program as **EvenInt.java** in the Chapter.06 folder on your Student Disk. Compile and run the program. The program prints 100 is evenly divisible by 1 2 4 5 10 20 25 50.

What if you want to know what number goes evenly into 100, but also what every number up to 100 can be evenly divided by? You can write 99 more loops, or you can place the current loop inside a different, outer loop, as you will do next.

When you use a loop within a loop, you should always think of the outer loop as the all-encompassing loop. When you describe the task at hand, you often use the word "each" when referring to the inner loop. For example, if you want to print three mailing labels each for 20 customers, the label variable would control the inner loop:

```
for(customer = 1; customer <= 20; ++customer)
for(label = 1; label <=3; ++label)
 printLabelMethod();
```

If you want to print divisors for each number from 1 to 100, then the loop that varies the number to be divided is the outside loop. You need to perform 100 mathematical calculations on each number, so that constitutes the "smaller" or inside loop.

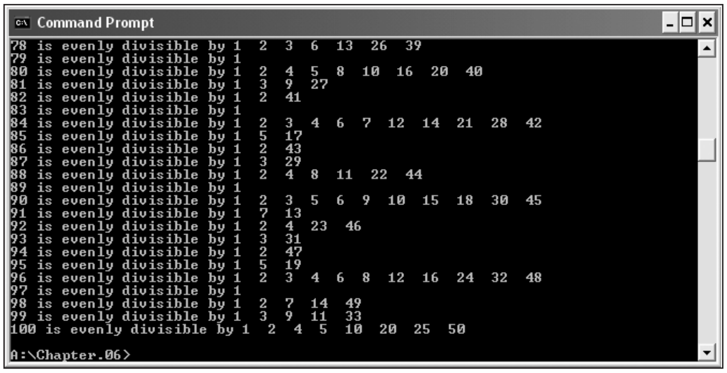
To create a nested loop to print even divisors for every number up to 100:

1. Open the file **EvenInt.java** in your text editor, and then save the class as **`EventInt2`**. Create an outer loop that uses the variable testNum to test every number from 1 to 100. Position the insertion point after the declaration of num but before the semicolon that ends the declaration, and then type a comma and **`testNum`**.
2. Position the insertion point to the right of the line with the variable declarations, press **[Enter]** to start a new line, and then type the other `for` loop:

   `for(testNum = 1; testNum <= 100; ++testNum)`
3. Press **[Enter]**, and then type the opening curly brace for this loop on the next line.
4. Change the statement that prints "100 is evenly divisible by " to the following:

   `System.out.print(testNum + " is evenly divisible by ");`
5. Change the `for` statement that varies num from 1 to 100 so it only varies num from 1 to testNum. For example, during each iteration, if testNum is 46, you want to divide it only by numbers that are 45 or less. Type the following code to make this change:

   `for(num = 1; num < testNum; ++num)`
6. Change the statement that tests 100%num to **`if(testNum%num == 0)`**.
7. Following the empty println() statement, add the closing curly brace for the outer `for` loop.
8. Save the file as **EventInt2**, compile, and run the program. The output will scroll on the screen. When it stops, it should look similar to Figure 6-12.



```
Command Prompt
78 is evenly divisible by 1  2  3  6  13  26  39
79 is evenly divisible by 1
80 is evenly divisible by 1  2  4  5  8  10  16  20  40
81 is evenly divisible by 1  3  9  27
82 is evenly divisible by 1  2  41
83 is evenly divisible by 1
84 is evenly divisible by 1  2  3  4  6  7  12  14  21  28  42
85 is evenly divisible by 1  5  17
86 is evenly divisible by 1  2  43
87 is evenly divisible by 1  3  29
88 is evenly divisible by 1  2  4  8  11  22  44
89 is evenly divisible by 1
90 is evenly divisible by 1  2  3  5  6  9  10  15  18  30  45
91 is evenly divisible by 1  7  13
92 is evenly divisible by 1  2  4  23  46
93 is evenly divisible by 1  3  31
94 is evenly divisible by 1  2  47
95 is evenly divisible by 1  5  19
96 is evenly divisible by 1  2  3  4  6  8  12  16  24  32  48
97 is evenly divisible by 1
98 is evenly divisible by 1  2  7  14  49
99 is evenly divisible by 1  3  9  11  33
100 is evenly divisible by 1  2  4  5  10  20  25  50

A:\Chapter.06>
```

Depending on your monitor resolution, you may see more lines than are displayed here

Figure 6-12 Output of the EvenInt2 program

When the program executes, 100 lines of output display on the screen. But, as Figure 6-12 shows, the first 76 (or so) lines scroll so rapidly that you can't read them. It would help if you could stop the output after every 20 lines or so; then you would have time to read the messages. Next you will use the modulus operator for this task. If you stop output when testNum is 20, 40, 60, and 80, then you can test testNum to see if it is evenly divisible by 20. When it is, you can pause program execution by asking the user to press [Enter] and accept keyboard input.

To pause your program after every 20 lines of output:

1. At the end of the EvenInt file and just prior to the closing brace for the `for` loop, type the following code to test testNum to determine if 20 divides into it evenly, then tell the user to press [Enter] and accept an [Enter] key from the keyboard (you don't have to store the entered key in a variable):

   ```
   if(testNum % 20 == 0)
   {
    System.out.println("Press Enter to continue");
    System.in.read(); System.in.read();
   }
   ```

2. Because the program now uses the System.in.read() method, you must position the insertion point at the end of the main() method header line and add **`throws Exception`**.

3. Save, compile, and test the program. It will pause after every 20 lines of output and wait until you press [Enter] before continuing until the program ends.

Chapter Summary

- A loop is a structure that allows repeated execution of a block of statements. A loop that never ends is called an infinite loop. A loop that executes a specific number of times is a definite loop or counted loop. You can nest loops.

- Within a looping structure, a Boolean expression is evaluated, and if it is `true`, a block of statements called the loop body executes; then the Boolean expression is evaluated again.
- You can use a `while` loop to execute a body of statements continually `while` some condition continues to be `true`.
- To execute a `while` loop, you initialize a loop control variable, test it in a `while` statement, and then alter the loop control variable in the body of the `while` structure.
- The `+=` operator adds and assigns in one operation.
- The prefix ++ and the postfix ++ increase a variable's value by one. The prefix -- and postfix -- decrement operators reduce a variable's value by one. When you use the prefix ++, the result is calculated and stored, and then the variable is used. When you use the postfix ++, the variable is used, and then the result is calculated and stored.
- Unary operators are used with one value. Most arithmetic operators are binary operators that operate on two values.
- The shortcut operators `+=`, `-=`, `*=`, and `/=` perform operations and assign the result in one step.
- A `for` loop initializes, tests, and increments in one statement. There are three sections within the parentheses of a `for` loop that are separated by exactly two semicolons.
- The `do...while` loop tests a Boolean expression after one repetition has taken place, at the bottom of the loop.

REVIEW QUESTIONS

1. A structure that allows repeated execution of a block of statements is a(n) ________________.
   a. cycle
   b. loop
   c. ring
   d. iteration
2. A loop that never ends is a(n) ________________ loop.
   a. iterative
   b. infinite
   c. structured
   d. illegal

3. To construct a loop that works correctly, you should initialize a loop control ________________.
   a. variable
   b. constant
   c. structure
   d. condition
4. What is the output of the following code?
   ```
   b = 1;
   while (b < 4)
   System.out.println(b + " ");
   ```
   a. 1
   b. 1 2 3
   c. 1 2 3 4
   d. 1 1 1 1 1 1...
5. What is the output of the following code?
   ```
   b = 1;
   while (b < 4)
   {
    System.out.println(b + " ");
    b = b + 1;
   }
   ```
   a. 1
   b. 1 2 3
   c. 1 2 3 4
   d. 1 1 1 1 1...
6. What is the output of the following code?
   ```
   e = 1;
   while (e < 4);
   System.out.println(e + " ");
   ```
   a. 1
   b. 1 1 1 1 1 1...
   c. 1 2 3 4
   d. 4 4 4 4 4 4...

7. If `total = 100` and `amt = 200`, then after the statement `total += amt`, ____________.
   a. total is equal to 200
   b. total is equal to 300
   c. amt is equal to 100
   d. amt is equal to 300
8. The modulus operator `%` is a ____________ operator.
   a. unary
   b. binary
   c. tertiary
   d. postfix
9. The prefix ++ is a ____________ operator.
   a. unary
   b. binary
   c. tertiary
   d. postfix
10. If `g = 5`, then the value of the expression `++g` is ____________.
    a. 4
    b. 5
    c. 6
    d. 7
11. If `h = 9`, then the value of the expression `h++` is ____________.
    a. 8
    b. 9
    c. 10
    d. 11
12. If `j = 5` and `k = 6`, then the value of `j++ == k` is ____________.
    a. 5
    b. 6
    c. `true`
    d. `false`

13. You must always include ________________ in a `for` loop's parentheses.
    a. two semicolons
    b. three semicolons
    c. two commas
    d. three commas
14. The statement `for(a = 0; a < 5; ++a) System.out.print(a + " ");` prints ________________.
    a. 0 0 0 0 0
    b. 0 1 2 3 4
    c. 0 1 2 3 4 5
    d. nothing
15. The statement `for(b = 1; b > 3; ++b) System.out.print(b + " ");` prints ________________.
    a. 1 1 1
    b. 1 2 3
    c. 1 2 3 4
    d. nothing
16. What does the following statement print?

```
for(f = 1, g = 4; f < g; ++f, —g)
 System.out.print(f + " " + g + " ");
```

    a. 1 4 2 5 3 6 4 7...
    b. 1 4 2 3 3 2
    c. 1 4 2 3
    d. nothing
17. The loop that performs its conditional check at the bottom of the loop is a ________________ loop.
    a. `while`
    b. `do...while`
    c. `for`
    d. `for...while`

18. What does this program segment print?

```
d = 0;
do
{
 System.out.print(d + " ");
 d++;
} while (d < 2);
```

   a. 0

   b. 0 1

   c. 0 1 2

   d. nothing

19. What does this program segment print?

```
for(f = 0; f < 3; ++f)
 for(g = 0; g < 2; ++g)
 System.out.print(f + " " + g + " ");
```

   a. 0 0 0 1 1 0 1 1 2 0 2 1

   b. 0 1 0 2 0 3 1 1 1 2 1 3

   c. 0 1 0 2 1 1 1 2

   d. 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2

20. What does this program segment print?

```
for(m = 0; m < 4; ++m);
 for(n = 0; n < 2; ++n);
 System.out.print(m + " " + n + " ");
```

   a. 0 0 0 1 1 0 1 1 2 0 2 1 3 0 3 1

   b. 0 1 0 2 1 1 1 2 2 1 2 2

   c. 4 2

   d. 3 1

EXERCISES

1. Write a program that prints all even numbers from 2 to 100 inclusive. Save the program as **EvenNums.java** in the Chapter.06 folder on your Student Disk.
2. Write a program that asks a user to type A, B, C, or Q. When the user types Q, the program ends. When the user types A, B, or C, the program displays the message "Good job!" and then asks for another input. When the user types anything else, issue an error message and then ask for another input. Save the program as **ABCInput.java** in the Chapter.06 folder on your Student Disk.
3. Write a program that prints every integer value from 1 to 20 along with its squared value. Save the program as **TableOfSquares.java** in the Chapter.06 folder on your Student Disk.

4. Write a program that sums the integers from 1 to 50 (that is, 1 + 2 + 3... + 50). Save the program as **Sum50.java** in the Chapter.06 folder on your Student Disk.
5. Write a program that shows the sum of 1 to n for every n from 1 to 50. That is, the program prints 1, 3 (the sum of 1 and 2), 6 (the sum of 1, 2, and 3), and so on. Save the program as **EverySum.java** in the Chapter.06 folder on your Student Disk.
6. Write a program that prints every perfect number from 1 through 1000. A perfect number is one that equals the sum of all the numbers that divide evenly into it. For example, 6 is perfect because 1, 2, and 3 divide evenly into it, and their sum is 6. Save the program as **Perfect.java** in the Chapter.06 folder on your Student Disk.
7. Write a program that calculates the amount of money earned on an investment; that amount should include 12 percent interest. Prompt the user to choose the investment amount from one menu and the number of years for the investment from a second menu. Display the total amount (balance) for each year of the investment. Use a loop instruction to calculate the balance for each year. Use the formula amount = investment * (1 + interest) raised to a power equal to the year to calculate the balance. Use the Math power function from Chapter 4, Math.pow(x,y) where x = (1 + interest) and y is the year. Save the program as **Investment.java** in the Chapter.06 folder on your Student Disk.
8. Write a program that creates a quiz that contains questions about a hobby, popular music, astronomy, or any other personal interest. After the user selects a topic, display a series of questions. The user should answer the questions with one character for multiple choice, true/false, or yes/no. If the user responds to a question correctly, display an appropriate message. If the user responds to a question incorrectly, display an appropriate response and the correct answer. At the end of the quiz, display the number of correct answers. Save the program as **Quiz.java** file in the Chapter.06 folder on your Student Disk.
9. Write a program that displays a series of survey questions, with one-character answers. At the end of the survey, ask the user if he or she wants to enter another set of responses. If the user responds no, then display the results of the survey for each question. Enter several sets of responses to test the program. Save the program as **Survey.java** in the Chapter.06 folder on your Student Disk.
10. Write a program that displays the results of a coin toss. The user is prompted to enter a for heads and b for tails. Use the Math.random() function from Chapter 4 to generate a number between 0 and 1. Use the probabilities of .5 for either a head or tail. Make sure that the user enters an a or b by continuing to loop until a or b is entered. Print either "You win" or "You lose" as the output. Save the program as **FlipCoin.java** in the Chapter.06 folder on your Student Disk.
11. Write a program that will count and display the number of heads and tails for a coin flipped from 1 to 9 times. The user is prompted for an integer from 1 to 9. Use the Math.random() function from Chapter 4 to generate a number between 0 and 1. Use the probabilities of .5 for either a head or a tail. Make use of the fact that the ASCII code for the numerals 0 to 9 is decimal 48 through 57. Save the program as **CountFlips.java** in the Chapter.06 folder on your Student Disk.

12. Each of the following files in the Chapter.06 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, save DebugSix1.java as FixDebugSix1.java.
    a. DebugSix1.java
    b. DebugSix2.java
    c. DebugSix3.java
    d. DebugSix4.java

CASE PROJECT

Local Caterer's Company operates a small Mom and Pop catering service. They want you to write an object-oriented program for them to schedule their catering events. They mostly cater special events, but they occasionally cater corporate, private and nonprofit events as well. Mom and Pop are both managers; Mom manages nonprofit and special events, and Pop manages the corporate and private events. All catered events have an event minimum rate shown in the table below:

Local Caterer's Company fee schedule

	Event Minimum Rate	Manager
Corporate	$500.00	Pop
Private	$300.00	Pop
Nonprofit	$150.00	Mom
Special	$200.00	Mom

Write a program for Local Caterer's Company that contains a class that has methods for the event minimum rate and event type. Write a test program that accepts keyboard input and checks the event type input for errors until a valid event type is entered. After a valid input type is entered, print the manager's name for that event, the type of event chosen, and the minimum rate to be charged.

CHAPTER

7 CHARACTERS, STRINGS, AND THE STRINGBUFFER

In this section, you will:

- Manipulate characters
- Declare a String object
- Compare String values
- Use other String methods
- Convert Strings to numbers
- Learn about the StringBuffer class

"I can write an interactive program that accepts a character, but I really want to be able to manipulate characters and let users enter words or numbers into programs," you announce to Lynn Greenbrier, your Java mentor.

"You need to learn about the String," Lynn says. "The wide variety of String methods provided with the Java programming language will help you use words and phrases efficiently. You'll even be able to let your users input numbers."

Previewing a Guessing Game Program

To demonstrate the use of the String methods, you will test a simple guessing game, similar to Hangman. The user first will guess letters, and then guess the motto of Event Handlers Incorporated.

To preview the guessing game:

1. Open the **Chap7SecretPhrase.java** file in the Chapter.07 folder on your Student disk.
2. At the command prompt, compile the class with the command **`javac Chap7SecretPhrase.java`**.
3. Run the program with the command **`java Chap7SecretPhrase`**. Enter keyboard characters one at a time and guess Event Handlers' motto.

Manipulating Characters

You learned in Chapter 2 that the char data type is used to hold any single character. In addition to the primitive data type char, Java offers a Character class. The **Character class** contains standard methods for testing the values of characters, such as letters or digits.

```
Java.lang.Object
  |
  +--java.lang.Character
```

Figure 7-1 Structure of the Character class

Table 7-1 Common methods of the Character class

Method	Description
isUpperCase()	Tests if character is uppercase
toUpperCase()	Changes a lowercase character to uppercase
isLowerCase()	Tests if character is lowercase
toLowerCase()	Changes an uppercase character to lowercase
isDigit()	Returns `true` if the argument is a digit (0-9) and `false` otherwise
isLetter()	Returns `true` if the argument is a letter and `false` otherwise
isLetterOrDigit()	Returns `true` if the argument is a letter or digit and `false` otherwise
isWhitespace()	Returns `true` if the argument is whitespace and `false` otherwise. (This includes the space, tab, newline, carriage return, and form feed.)

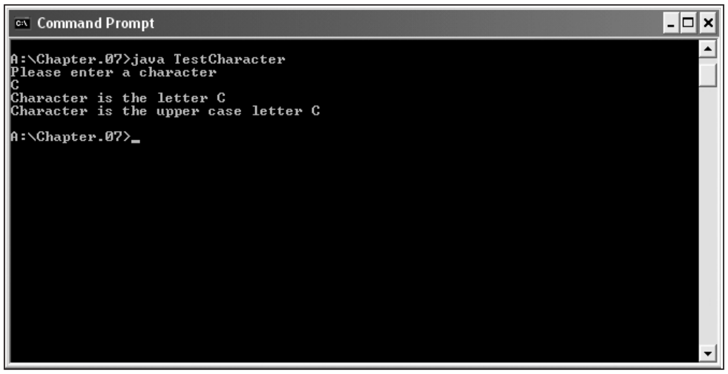
Figure 7-1 shows the Character class is defined in java.lang.Object and is automatically imported into every program you write. Commonly used methods available in the Character class are shown in Table 7-1.

Figure 7-2 contains a program that uses many of the methods shown in Table 7-1. The program declares a character variable named aChar and allows the user to supply a value for aChar by entering a single character from the keyboard. The program tests aChar using a series of `if` and `if…else` statements to determine whether the entered character is a digit, letter, or whitespace, and if it is a letter, whether it is lowercase or not.

```
public class TestCharacter
{
  public static void main(String[] args)throws Exception
  {
       char aChar;
       System.out.println("Please enter a character");
       aChar = (char)System.in.read();
       System.in.read();System.in.read();
       if(Character.isDigit(aChar))
              System.out.println(aChar+"is a number");
       else System.out.println(aChar+"is not a number");
       if(Character.isWhitespace(aChar))
              System.out.println(aChar+ "is a letter");
       else System.out.println(aChar+ "is not a letter");
       if(Character.isWhitespace(aChar))
              System.out.println
              ("Character is a whitespace character");
       else System.out.println
              ("Character is not a whitespace character");
       if(Character.isLetter(aChar))
              if(Character.isLowerCase(aChar))
                     System.out.println(aChar+
                     " is lowercase character");
              else
                     System.out.println(aChar+
                     " is not a lowercase character");
  }
}
```

Figure 7-2 The TestCharacter program

The output of the TestCharacter program for sample keyboard input of an uppercase character "C" is shown in Figure 7-3.



```
A:\Chapter.07>java TestCharacter
Please enter a character
C
Character is the letter C
Character is the upper case letter C

A:\Chapter.07>_
```

Figure 7-3 Sample output of the TestCharacter program

DECLARING A STRING OBJECT

You learned in Chapter 1 that a sequence of characters enclosed within double quotation marks is a literal string. You have used many literal strings, such as "First Java program," within println() statements.

Tip: You also use String in main() method headers.

A **String** variable is simply an object of the class String. The class String is defined in java.lang.String, which is automatically imported into every program you write. Figure 7-4 shows that the String class descends directly from the Object class. The String itself is distinct from the variable you use to refer to it. You create a String object by using the keyword `new` and the String constructor method, just as you would create an object of any other type. For example, `String aGreeting = new String("Hello");` is a statement that defines an object named aGreeting, declares it to be of type String, and assigns an initial value of "Hello" to the String. The variable aGreeting stores a reference to a String object—it keeps track of where the String object is stored in memory. When you declare and initialize aGreeting, it links to the initializing String value. Alternately, you can declare a String containing "Hello" with `String aGreeting = "Hello";`. Unlike other classes, the String class is special because you can create a String object without using the keyword `new` or calling the class constructor.

```
Java.lang.Object
  |
  +--java.lang.String
```

Figure 7-4 Structure of the String class

After declaring a String, you can display it in a print() or println() statement, just as you would for any other variable—for example, `System.out.println("The greeting is " + aGreeting);`.

Comparing String Values

In the Java programming language, String is a class, and each created String is a class object. A String variable name is a reference; that is, a String variable name refers to a location in memory, rather than to a particular value.

The distinction is subtle, but when you declare a variable of a basic, primitive type, such as `int x = 10;`, the memory address where x is located holds the value 10. If you later assign a new value to x, for example, `x = 45;`, then 45 replaces 10 at the assigned memory address. When you declare a String, such as `String aGreeting = "Hello";`, aGreeting holds a memory address where the characters "Hello" are stored. If you subsequently assign a new value to aGreeting, such as `aGreeting = "Bonjour";`, then the address held by aGreeting is altered; now aGreeting holds a new address where the characters "Bonjour" are stored. "Bonjour" is an entirely new object created with its own location. The "Hello" String is still in memory; it's just that aGreeting no longer holds its address. Eventually, a part of the Java system called the garbage collector will discard the "Hello" characters. Strings, therefore, are never actually changed; instead, new Strings are created and String variables hold the new addresses. Strings and other objects that can't be changed are known as **immutable**.

Because String variables hold memory addresses, you cannot make a simple comparison to determine whether two String objects are equivalent. For example, if you declare two Strings as `String aGreeting = "Hello";` and `String anotherGreeting = "Hello";`, Java will evaluate a comparison, such as `if(aGreeting == anotherGreeting)` as `false`. This is because when you compare aGreeting to anotherGreeting with the == operator, you are comparing their memory addresses, not their values.

Fortunately, the String class provides you with a number of useful methods. The **equals() method** evaluates the contents of two String objects to determine if they are equivalent. The method returns `true` if the objects have identical contents. For example, Figure 7-5 shows two String objects and several comparisons. Each comparison in Figure 7-5 is true; each results in printing the line "Name's the same."

```
String aName = "Roger";
String anotherName = "Roger";
if(aName.equals(anotherName))
  System.out.println("Name's the same");
if(anotherName.equals(aName))
  System.out.println("Name's the same");
if(aName.equals("Roger");
  System.out.println("Name's the same");
```

Figure 7-5 String comparisons using the equals() method

Tip

The String class equals() method returns `true` only if two Strings are identical in content. Thus, a String holding "Roger " (with a space after the r) is not equivalent to a String holding "Roger" (with no space after the r).

Each String shown in Figure 7-5 (aName and anotherName) is an object of type String, so each String has access to the String class equals() method. The aName object can call equals() with `aName.equals()`, or the anotherName object can call equals() with `anotherName.equals()`. The equals() method can take either a variable String object or a literal string as its argument.

The **equalsIgnoreCase() method** is similar to the equals() method. As its name implies, it ignores case when determining if two Strings are equivalent. Thus, `aName.equals("roGER")` is `false`, but `aName.equalsIgnoreCase("roGER")` is `true`. This method is useful when users type responses to prompts in your programs. You cannot predict when a user might use the Shift key or the Caps Lock key during data entry. The equalsIgnoreCase() method allows you to test entered data without regard to capitalization.

When the **compareTo() method** is used to compare two Strings, it provides additional information to the user in the form of an integer value. When you use compareTo() to compare two String objects, the method returns zero only if the two Strings hold the same value. If there is any difference between the Strings, a negative number is returned if the calling object is "less than" the argument, and a positive number is returned if the calling object is "more than" the argument. Strings are considered "less than" or "more than" each other based on their Unicode values; thus, "a" is less than "b," and "b" is less than "c."

For example, if aName holds "Roger," then `aName.compareTo("Robert");` returns a 5. The number is positive, indicating that "Roger" is more than "Robert." This does not mean that "Roger" has more characters than "Robert"; it means that "Roger" is alphabetically "more" than "Robert." The comparison proceeds as follows:

- The R in "Roger" and the R in "Robert" are compared, and found to be equal.
- The o in "Roger" and the o in "Robert" are compared, and found to be equal.

- The g in "Roger" and the b in "Robert" are compared; they are different. The numeric value of g minus the numeric value of b is 5 (because g is five letters after b in the alphabet), so the compareTo() method returns the value 5.

Often you won't care what the specific return value of compareTo() is; you simply will want to determine if it is positive or negative. For example, you can use a test, such as `if(aWord.compareTo(anotherWord)<0)...` to determine whether aWord is alphabetically less than anotherWord. If aWord is a String variable that holds the value "hamster," and anotherWord is a String variable that holds the value "iguana," then the comparison `if(aWord.compareTo(anotherWord)<0)` yields `true`.

Using Other String Methods

A wide variety of additional String methods are available with the String class. The methods toUpperCase() and toLowerCase() convert any String to its uppercase or lowercase equivalent. For example, if you declare a String as `String aWord = "something";`, then `aWord = aWord.toUpperCase` assigns "SOMETHING" to aWord. Because aWord now is assigned "SOMETHING," `aWord = aWord.toLowerCase()` assigns "something" to aWord. The **indexOf() method** determines whether a specific character occurs within a String. If it does, the method returns the position of the character. The first position of a String begins with zero rather than 1. The return value is -1 if the character does not exist in the String. For example, in `String myName = "Stacy";`, the value of `myName.indexOf('a')` is 2, and the value of `myName.indexOf('q')` is -1.

The **charAt() method** requires an integer argument which indicates the position of the character that the method returns. For example, if myName is a String holding "Stacy," then the value of `myName.charAt(0)` is 'S' and `myName.charAt(1)` is 't'.

The **endsWith() method** and the **startsWith() method** each take a String argument and return `true` or `false` if a String object does or does not end or start with the specified argument. For example, if `String myName = "Stacy";`, then `myName.startsWith("Sta")` is `true`, and `myName.endsWith("z")` is `false`.

The **replace() method** allows you to replace all occurrences of some character within a String. For example, if `String yourName = "Annette";`, then `String goofyName = yourName.replace('n', 'X');` assigns "AXXette" to goofyName.

The **toString() method** converts any primitive type to a String. So, if you declare a String as `theString` and an integer as `int someInt = 4;`, then `theString = Integer.toString(someInt);` results in the String "4" being assigned to theString. If you declare another String as `aString` and a double as `double someDouble = 8.25`, then `aString = Double.toString(someDouble);` assigns the String "8.25" to aString.

Another method is available to convert any primitive type to a String. If you declare a String as `anotherString` and a float as `float someFloat = 12.34f`, then `anotherString = "" + someFloat`, assigns the String "12.34" to anotherString. The Java interpreter first converts the float 12.34f to a String "12.34," and adds it to the null String "". Joining Strings is called **concatenation**. The resulting string "12.34" is then assigned to anotherString.

Tip

The toString() method is not part of the String class; it is a method included in Java that you can use with any type of object. You have been using toString() throughout this book without knowing it. When you use print() and println(), their arguments are automatically converted to Strings if necessary. You don't need import statements to use toString() because it is part of java.lang, which is imported automatically.

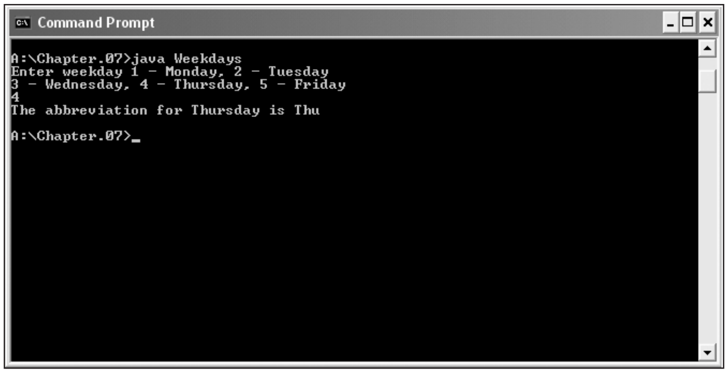
Tip

Because the toString() method takes arguments of any primitive type, including int, char, double, and so on, it is an overloaded method.

You already know that you can join Strings with other Strings or values by using a plus sign (+); you have used this approach in println() statements since Chapter 1. For example, you can print a firstName, a space, and a lastName with `System.out.println(firstName + " " + lastName);`. Additionally, you can extract part of a String with the substring() method, and use it alone or concatenate it with another String. The substring() method takes two arguments—a start position and an end position—that are both based on the fact that a String's first position is position zero. For example, the program segment in Figure 7-6 shows the names Monday through Friday as Strings representing the days of the week. An abbreviation of a weekday, Monday for example, can be printed after using the substring method `System.out.println("The abbreviation for Monday is " + monday.substring(0,3));`. Here `monday.substring(0,3)` starts at the first character (index of 0), and extracts the first three letters from the string stored as "Monday." The output of the weekday abbreviations program is shown in Figure 7-7.

```
public class Weekdays
{
   public static void main(String() args)throws Exception
   {
      char weekday;

      String monday = "Monday";
      String tuesday = "Tuesday";
      String wednesday = "Wednesday";
      String thursday = "Thursday";
      String friday = "Friday";

      System.out.println("Enter weekday 1 - Monday, 2 - Tuesday");
      System.out.println("3 - Wednesday, 4 - Thursday, 5 - Friday");
      weekday = (char)System.in.read();
      System.in.read();System.in.read();

      if(weekday == '1')
        System.out.println("The abbreviation for Monday is " +
         monday.substring(0,3));
      else if(weekday == '2')
        System.out.println("The abbreviation for Tuesday is " +
         tuesday.substring(0,3));
      else if(weekday == '3')
        System.out.println("The abbreviation for Wednesday is " +
         wednesday.substring(0,3));
      else if(weekday == '4')
        System.out.println("The abbreviation for Thursday is " +
         thursday.substring(0,3));
      else
        System.out.println("The abbreviation for Friday is " +
         friday.substring(0,3));
   }
}
```

Figure 7-6 Program segment demonstrating the substring method and String concatenation



```
A:\Chapter.07>java Weekdays
Enter weekday 1 - Monday, 2 - Tuesday
3 - Wednesday, 4 - Thursday, 5 - Friday
4
The abbreviation for Thursday is Thu

A:\Chapter.07>_
```

Figure 7-7 Output of the substring method and String concatenation code segment

To demonstrate the use of the String methods, you will create a simple guessing game, similar to Hangman. The user first will guess letters, and then attempt to guess the motto of Event Handlers Incorporated.

To create the guessing game:

1. Open a new text file in your text editor. Enter the following first few lines of a SecretPhrase program. The program will contain the target phrase that the user will try to guess ("Plan With Us"), as well as a display phrase that is mostly asterisks (with a few hints).

   ```
   public class SecretPhrase
   {
     public static void main(String[] args) throws Exception
     {
       String targetPhrase = "Plan With Us";
       String displayPhrase = "P*** W*** U*";
   ```

2. Add the following variables that will hold the user's guess and the position of a guess that is found within the phrase:

   ```
   char guess;
   int position;
   ```

3. Next, add the following brief instruction:

   ```
   System.out.println("Play our game - guess our motto");
   ```

4. Enter the statement to display the hint phrase:

   ```
   System.out.println(displayPhrase);
   ```

5. Add the following loop that continues while asterisks remain in the displayPhrase. The user will enter a letter. You will use the indexOf() method to determine whether the guessed letter appears in the targetPhrase. If it does not, then ask the user to guess again. If the guessed letter appears in the phrase, you reconstruct the display phrase with the following:

   - The substring of characters in the display phrase that comes before the correct guess
   - The correct guess
   - The substring of characters in the display phrase that appears after the correct guess; in other words, the correct letter replaces the appropriate asterisk

   Add the following code:

   ```
   while(displayPhrase.indexOf('*') != -1)
   {
     System.out.println("Enter a letter");
     guess = (char)System.in.read();
   ```

```
  System.in.read(); System.in.read();
    // Absorbs Enter key
  position = targetPhrase.indexOf(guess);
    // Determines position of guess
  if(position == -1) // If guess is not in target phrase
    System.out.println("Sorry, guess again");
  else // If guess is in target phrase
  {
    displayPhrase = displayPhrase.substring(0,position) +
      guess + displayPhrase.substring
        (position+1,displayPhrase.length());
    System.out.println(displayPhrase);
  }
}
```

6. The `while` loop will continue until all the asterisks in the targetPhrase are replaced by correct letters. Therefore, after the closing curly brace for the `while` loop, enter:

   `System.out.println("Congratulations!");`.

7. Type the closing curly braces for the main() method and for the SecretPhrase class.
8. Save the program as **SecretPhrase.java** in the Chapter.07 folder on your Student Disk, then compile and run the program. Make sure you understand how all the String methods contribute to the success of this program.

Converting Strings to Numbers

If a String contains all numbers, as in "649," you can convert it from a String to a number so you can use it for arithmetic, or use it like any other number. To convert a String to an integer, you use the **Integer class**, which is part of java.lang and automatically imported into programs you write. The **parseInt() method** is part of the Integer class, and takes a String argument and returns its integer value. For example, `int anInt = Integer.parseInt("649");` stores the numeric value 649 in the variable anInt. You can then use the integer value just as you would any other integer.

Tip

The word parse in English means "to resolve into component parts," as when you "parse a sentence." In Java, to parse a String means to break down its separate characters into a numeric format.

It is also easy to convert a String object to a double value. You must use the **Double** class, which, like the Integer class, is also imported into your programs automatically. A method of the Double class is parseDouble(), which takes a String argument and returns its double value. For example, `double doubleValue = Double.parseDouble("147.82");` stores the numeric value 147.82 in the variable doubleValue.

To convert a String containing "147.82" to a double, you can use the following code:

```
String stringValue = new String("147.82");
Double tempValue = Double.valueOf(stringValue);
double value = tempValue.doubleValue();
```

The stringValue is passed to the Double.valueOf() method, which returns a Double object. The doubleValue() method, is used with the tempValue object. This method returns a double that is stored in value.

Tip

The Double and Integer classes are examples of wrappers. A wrapper is a class or object that is "wrapped around" a simpler thing. You use the Double (uppercase D) class to make it convenient to work with primitive double (lowercase d) variables.

When planning an event, Event Handlers Incorporated must know how many guests to expect. Next you will prompt the user for the number of guests, read characters from the keyboard, store the characters in a String, and then convert the String to an integer.

To create a program that accepts integer input:

1. Open a new text file in your text editor. Type the statement **`import javax.swing.*;`**, press **[Enter]**, and then enter the following first few lines of a DialogInput class that will accept string input:

```
public class NumInput
{
  public static void main(String[] args) throws Exception
  {
```

2. Declare the following variables to hold the input String and the resulting integer:

```
String inputString;
int inputNumber;
```

3. Enter the following input dialog box statement that stores the user keyboard input in the String variable inputString:

```
inputString = JOptionPane.showInputDialog(null,
  "Enter the number of guests at your event");
```

4. Use the following Integer.parseInt() method to convert the input String to an integer. Then use the integer in a numeric decision that displays a message dialog box when the number of guests entered is greater than 100:

```
inputNumber = Integer.parseInt(inputString);
if(inputNumber > 100)
  JOptionPane.showMessageDialog(null,
    "A surcharge will  apply!);
```

5. Enter the closing statement **`System.exit(0);`** press **[Enter]**, and then enter the final two closing curly braces for the program.
6. Save the program as **NumInput.java** in the Chapter.07 folder on your Student Disk, then compile and test the program.

Learning About the StringBuffer Class

A String class limitation is that the value of a String is fixed after the String is created; Strings are immutable. When you write `someString = "Hello";` and follow it with `someString = "Goodbye";`, you have neither changed the contents of computer memory at someString, nor have you eliminated the characters "Hello." Instead, you have stored "Goodbye" at a new computer memory location and stored the new address in the someString variable. If you want to modify someString from "Goodbye" to "Goodbye Everybody," you cannot add a space and "Everybody" to the someString that contains "Goodbye." Instead, you must create an entirely new String, "Goodbye Everybody," and assign it to the someString address.

To circumvent these limitations, you can use the StringBuffer class. **StringBuffer** is an alternative to the String class, and can usually be used anywhere you would use a String. The structure of the StringBuffer class is shown in Figure 7-8. Like the String class, the StringBuffer class is part of the java.lang package and is automatically imported into every program.

```
Java.lang.Object
  |
  +--java.lang.StringBuffer
```

Figure 7-8 Structure of the StringBuffer class

You can create a StringBuffer object that contains a given String with the statement `StringBuffer eventString = new StringBuffer("Event Handlers Incorporated");`. When you initialize a `StringBuffer` object you must use the keyword `new` and provide an initializing value between parentheses. You can create the StringBuffer variable using syntax similar to the syntax for creating a String variable, such as `StringBuffer philosophyString = null;`. The variable does not refer to anything until you initialize it with a defined `StringBuffer` object. For example, you could write `philosophyString = new StringBuffer("Dedicated to making your event a most memorable one");`. You can also initialize the StringBuffer variable philosophyString with an existing StringBuffer object using `philosophyString =  eventString;`.

Generally when you create a String object, sufficient memory is allocated to accommodate the number of Unicode characters in the string. A StringBuffer object, however, contains a memory block called a **buffer** which might not contain a string. Even if it does contain a String, the String might not occupy all of the buffer. In other words, the length of a String can be different from the length of the buffer. The actual length of the buffer is referred to as the **capacity** of the StringBuffer object. It is generally more efficient to make the StringBuffer capacity sufficient for the needs of your program, rather than modify a String that has been originally stored with a small capacity.

You can change the length of a String in a StringBuffer object with the setLength() method. The length is a property of the String held by the StringBuffer. When you increase a StringBuffer object's length to be longer than the String it holds, the extra characters contain '\u0000.' If you use the setLength() method to specify a length shorter than its String, the string is truncated.

To find the capacity of a StringBuffer object, you use the capacity() method. For example, the EventStringBuffer program in Figure 7-9 shows the creation of the eventString object as `StringBuffer eventString = new StringBuffer("Event Handlers Incorporated");`. The capacity of the StringBuffer object is obtained as `int aCapacity = eventString.capacity();` and printed using `System.out.println("The capacity is " + aCapacity);`. Figure 7-10 shows the StringBuffer capacity is 43. Note that the capacity of 43 is 16 characters larger than the length of the string "Event Handlers Incorporated," which contains 27.

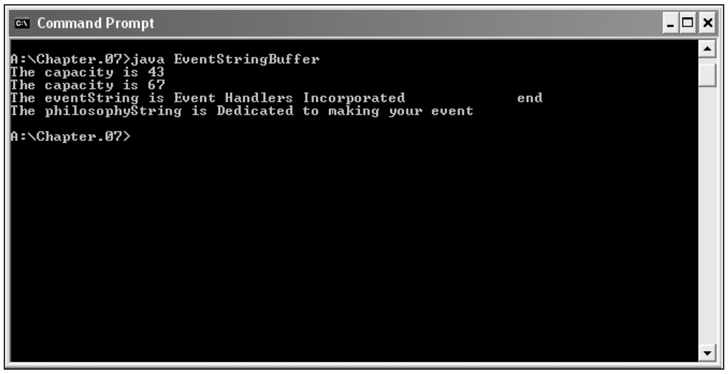
Tip: In general, when a StringBuffer object is created from a String, the capacity will be the length of the string plus 16.

In Figure 7-9 the philosophyString variable is created as `StringBuffer philosophyString = null;`. The variable does not refer to anything until it is initialized with the defined StringBuffer object `philosophyString = new StringBuffer("Dedicated to making your event a most memorable one");`. The capacity of philosophyString is shown in Figure 7-10 as the length of the string plus 16 or 67.

In the program shown in Figure 7-9, the length of the eventString is changed with the statement `eventString.setLength(40);`. When the value of the length prints, as shown in Figure 7-10, the extra characters show as blanks before the String "end" is printed. Also in Figure 7-10, the philosophyString length is shortened to a length of 30 in the statement `philosophyString.setLength(30);`. The shortened 30-character output string is shown as "Dedicated to making your event" in Figure 7-10.

```
public class EventStringBuffer
{
  public static void main(String[] args)
  {
  StringBuffer eventString =
        new StringBuffer("Event Handlers Incorporated");
     int aCapacity = eventString.capacity();
     System.out.println("The capacity is " + aCapacity);

     StringBuffer philosophyString = null;
     philosophyString =
        new StringBuffer("Dedicated to making your event a most memorable one");
     int bCapacity = philosophyString.capacity();
     System.out.println("The capacity is " + bCapacity);

     eventString.setLength(50);
     System.out.println("The eventString is " + eventString + "end");

     philosophyString.setLength(30)
        System.out.println("The philosophyString is " + philosophyString);
  }
}
```

Figure 7-9 EventStringBuffer program



```
A:\Chapter.07>java EventStringBuffer
The capacity is 43
The capacity is 67
The eventString is Event Handlers Incorporated             end
The philosophyString is Dedicated to making your event

A:\Chapter.07>
```

Figure 7-10 Output of the EventStringBuffer program

Using StringBuffer objects provides more flexibility than String objects because you can insert or append new contents into a StringBuffer. The StringBuffer class provides you with three constructors:

- `public StringBuffer()` constructs a StringBuffer with no characters and a default size of 16 characters
- `public StringBuffer(int length)` constructs a StringBuffer with no characters, and a capacity specified by length

- `public StringBuffer(String s)` contains the same characters as those stored in the String object s (The capacity of the StringBuffer is the length of the String argument you provide, plus 16 additional characters.)

The **append() method** lets you add characters to the end of a StringBuffer object. For example, if a StringBuffer object is declared as `StringBuffer someBuffer = new StringBuffer("Happy");`, then the statement `someBuffer.append (" birthday")` alters someBuffer to hold "Happy birthday."

The **insert() method** lets you add characters at a specific location within a StringBuffer object. For example, if someBuffer holds "Happy birthday," then `someBuffer.insert(6, "30th ");` alters the StringBuffer to contain "Happy 30th birthday." The first character in the StringBuffer object occupies position zero. To alter just one character in a StringBuffer, you can use the **setCharAt() method**. This method requires two arguments, an integer position, and a character. If someBuffer holds "Happy 30th birthday," then `someBuffer.setCharAt(6,'4');` changes the someBuffer value into a 40th birthday greeting.

Next you will use StringBuffer methods.

To use StringBuffer methods:

1. Open a new text editor file, and type the following first lines of a DemoStringBuffer class:

```
public class DemoStringBuffer
{
  public static void main(String[] args)
  {
```

2. Use the following code to create a StringBuffer variable, and then call a print() method (that you will create in Step 7) to print the StringBuffer:

```
StringBuffer str = new StringBuffer("singing");
print(str);
```

3. Enter the following append() method to add characters to the existing StringBuffer and print it again:

```
str.append(" in the dead of ");
print(str);
```

4. Enter the following insert() method to insert characters, print, insert additional characters, and print the StringBuffer again:

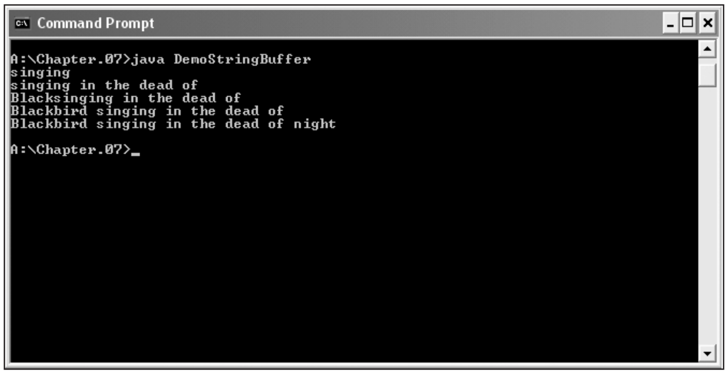
```
str.insert(0, "Black");
print(str);
str.insert(5, "bird ");
print(str);
```

5. Add one more append() and print() combination:

```
str.append("night");
print(str);
```

6. Add a closing curly brace for the main() method.
7. Enter the following print() method that prints StringBuffer objects:
```
public static void print(StringBuffer s)
{
  System.out.println(s);
}
```
8. Type the closing curly brace for the class, and then save the file as **DemoStringBuffer.java** in the Chapter.07 folder on your Student Disk. Compile and execute, and then compare your output to Figure 7-11.

You can extract characters from a StringBuffer object using the charAt()and getChars() methods. The **charAt() method** accepts an argument that is the offset of the character position from the beginning of a String. If you declare `StringBuffer text = new StringBuffer("Java  Programming");`, then `text.charAt(5)` refers to the character 'P.'



```
A:\Chapter.07>java DemoStringBuffer
singing
singing in the dead of
Blacksinging in the dead of
Blackbird singing in the dead of
Blackbird singing in the dead of night

A:\Chapter.07>_
```

Figure 7-11 Output of the DemoStringBuffer program

Tip: If you try to use an index less than 0 or greater than the index of the last position in the StringBuffer object, you will cause an exception to be thrown and your program will terminate.

Finally, you can change a single character in a StringBuffer object using the setCharAt() method with two arguments. The first argument indicates the index position of the character to be changed; the second argument specifies the replacement character. If you declare `StringBuffer text = "java";`, the statement `text.setCharAt(0,'J')` sets the first character to 'J'.

Chapter Summary

- A sequence of characters enclosed within double quotation marks is a literal string.
- You create a String object by using the keyword `new` and the String constructor method.
- Each String is a class object, and a String variable name is actually a reference. Strings, therefore, are never changed; they are immutable.
- The equals() method evaluates the contents of two String objects to determine whether they are equivalent, and then returns a Boolean value.
- The equalsIgnoreCase() method determines if two Strings are equivalent without considering case.
- The compareTo() method returns zero if two String objects hold the same value. A negative number is returned if the calling object is "less than" the argument, and a positive number is returned if the calling object is "greater than" the argument.
- The methods toUpperCase() and toLowerCase() convert any String to its uppercase or lowercase equivalent.
- The indexOf() method determines whether a specific character occurs within a String. If it does, the method returns the position of the character. The return value is -1 if the character does not exist in the String.
- The endsWith() and startsWith() methods each take a String argument and return `true` or `false`, depending on whether or not a String object ends with or starts with the specified argument.
- The replace() method allows you to replace all occurrences of some character within a String.
- The toString() method converts any primitive type to a String.
- You can join Strings with other Strings or values by using a plus sign (+); this process is called concatenation.
- You can extract part of a String with the substring() method, which takes two arguments, and a start and end position, both of which are based on the fact that a String's first position is position zero.
- If a String contains all numbers, you can convert it to a number.
- The parseInt() method takes a String argument and returns its integer value.
- The Double.valueOf() method converts a String to a Double object; the doubleValue() method converts a Double object to a double variable.
- To circumvent some limitations of the String class, you can use the StringBuffer class. You can insert or append new contents into a StringBuffer.

Review Questions

1. A sequence of characters enclosed within double quotation marks is a ________________.
   a. symbolic string
   b. literal string
   c. prompt
   d. command
2. To create a String object, you can use the keyword ________________.
   a. `object`
   b. `create`
   c. `char`
   d. `new`
3. A String variable name is a ________________.
   a. reference
   b. value
   c. constant
   d. literal
4. Objects that cannot be changed are ________________.
   a. irrevocable
   b. nonvolatile
   c. immutable
   d. stable
5. If you declare two String objects as `String word1 = new String("happy");` and `String word2 = new String("happy");`, then the value of `word1 == word2` is ________________.
   a. `true`
   b. `false`
   c. illegal
   d. unknown
6. If you declare two String objects as `String word1 = new String("happy");` and `String word2 = new String("happy");`, then the value of `word1.equals(word2)` is ________________.
   a. `true`
   b. `false`
   c. illegal
   d. unknown

7. The method that determines whether two String objects are equivalent, regardless of case, is ________________.
   a. equalsNoCase()
   b. toUpperCase()
   c. equalsIgnoreCase()
   d. equals()
8. If a String is declared as `String aStr = new String("lima bean");`, then `aStr.equals("Lima Bean");` is ________________.
   a. `true`
   b. `false`
   c. illegal
   d. unknown
9. If you create two String objects using `String name1 = new String("Jordan");` and `String name2 = new String("Jore");`, then `name1.compareTo(name2)` has a value of ________________.
   a. `true`
   b. `false`
   c. -1
   d. 1
10. If String `myFriend = new String("Ginny");`, then which of the following has the value 1?
    a. `myFriend.compareTo("Gabby");`
    b. `myFriend.compareTo("Gabriella");`
    c. `myFriend.compareTo("Ghazala");`
    d. `myFriend.compareTo("Hammie");`
11. If String `movie = new String("West Side Story");`, then the value of `movie.indexOf('s')` is ________________.
    a. `true`
    b. `false`
    c. 2
    d. 3
12. The String class replace() method replaces ________________.
    a. a String with a character
    b. one String with another String
    c. one character in a String with another character
    d. every occurrence of a character in a String with another character

13. The toString() method converts any ______________ to a String.
    a. character
    b. integer
    c. float
    d. all of the above
14. Joining Strings is called ______________.
    a. chaining
    b. joining
    c. linking
    d. concatenation
15. The first position in a String ______________.
    a. must be alphabetic
    b. must be uppercase
    c. is position zero
    d. is ignored by the compareTo() method
16. The substring() method requires ______________ arguments.
    a. no
    b. one
    c. two
    d. three
17. The method parseInt() converts a(n) ______________.
    a. integer to a String
    b. integer to a Double
    c. Double to a String
    d. String to an integer
18. The difference between int and Integer is ______________.
    a. int is a primitive type; Integer is a class
    b. int is a class; Integer is a primitive type
    c. nonexistent; they both are primitive types
    d. nonexistent; both are classes
19. For an alternative to the String class, you can use ______________.
    a. char
    b. StringHolder
    c. StringBuffer
    d. StringMerger

20. The default capacity for a StringBuffer object is ________________ characters.
    a. zero
    b. two
    c. 16
    d. 32

Exercises

1. Write a program that concatenates the three Strings: "Event Handlers is dedicated", "to making your event", and "a most memorable one." Print each String and the concatenated String. Save the program as **JoinStrings.java** in your Chapter.07 folder on your Student Disk.
2. Write a program that calculates the total number of vowels contained in the String "Event Handlers is dedicated to making your event a most memorable one." Save the program name **StringVowels.java** in your Chapter.07 folder on your Student Disk.
3. Write a program that calculates the total number of letters contained in the String "Event Handlers Incorporated, 8900 U.S. Highway 14, Crystal Lake, IL 60014". Save the program name as **StringLetters.java** in your Chapter.07 folder on your Student Disk.
4. Write a program that calculates the total number of whitespaces contained in the String "[TAB][TAB]   ", which represents two tabs and three spaces. Save the program as **StringWhite.java** in your Chapter.07 folder on your Student Disk.
5. Write a program that converts the variables someInt and someDouble in the statements `int someInt = 21;` and `someDouble = 128.04;` to strings using the classes Integer and Double. Save the program as **ToString.java** in your Chapter.07 folder on your Student Disk.
6. Write a program that converts the variables someInt, someDouble, and someFloat in the statements `int someInt = 567;`, `double someDouble = 48.25;`, and `float someFloat = 443.21f;` to strings. Do not use the Integer, Double, and Float classes to make the conversions. Save the program as **ToString2.java** in your Chapter.07 folder on your Student Disk.
7. a. Write a program that demonstrates that when two identical names are compared and the case differs, the equals method will return `false` when making a comparison. Save the program as **Comparison.java** in your Chapter.07 folder on your Student Disk.
   b. Demonstrate that the equalIgnoreCase() method will change the comparison in question 7a. from `false` to `true`.

8. Write a program to demonstrate that the compareTo() method returns either a positive number, a negative number, or a zero when used to compare two Strings. Save the program as **Compare.java** in your Chapter.07 folder on your Student Disk.
9. Write a program to demonstrate the following, based on the statement `String dedicate = "Dedicated to making your event a most memorable one"`:
   a. index of 'D'
   b. char at (15)
   c. endsWith(one)
   d. eplace('a', 'A').

   Save the program as **Demonstrate.java** in your Chapter.07 folder on your Student Disk.

10. Create a class that holds three initialized StringBuffer objects: your first name, middle name, and last name. Create three new StringBuffer objects as follows:
    - An object named EntireName that holds your three names, separated by spaces
    - An object named LastFirst that holds your last name, a comma, a space, and your first name, in that order
    - An object named Signature that holds your first name, a space, your middle initial (not the entire name), a period, a space, and your last name

    Display all three objects. Save the program as **Buffer.java** in the Chapter.07 folder on your Student Disk.
11. Each of the following files in the Chapter.07 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugSeven1.java will become FixDebugSeven1.java.
    a. DebugSeven1.java
    b. DebugSeven2.java
    c. DebugSeven3.java
    d. DebugSeven4.java

Case Project

The Tax Advantage Company provides free tax services to residents who cannot calculate their personal taxes or pay for calculation by a public tax service. You have been asked to write a Java program that will calculate an estimated tax for either a single or married taxpayer, given a keyboard income entry. After the income is entered, a code status “S” for single or “M” for married is entered.

The necessary classes are a Tax class with a main() method for keyboard entry of income and code status values, and a TaxReturn class to calculate the tax based on the input. The input values should be passed from the Tax class to the TaxReturn class through the instantiation of a TaxReturn object. The instantiation could take a form such as `TaxReturn aTaxReturn = new TaxReturn(income, status);`.

For taxpayers, two tax rates are needed, 15 percent and 30 percent. For single taxpayers, the cutoff rate of 15 percent is $10,000, and 30 percent above $10,000. For married taxpayers, the cutoff rate is 15 percent for amounts below $20,000, and 30 percent for amounts of $20,000 and above.

Program output should show the code status, income, and the amount of tax. Save the program files as **Tax.jav**a and **TaxReturn.java** in the Chapter.07 folder on your Student Disk.

CHAPTER

8 Arrays

In this section, you will:

- Declare and initialize an array
- Use subscripts with an array
- Declare an array of objects
- Search an array for an exact match or a range match
- Pass arrays to methods
- Use the length field
- Create arrays of Strings
- Sort primitive, object, and String array elements
- Use two-dimensional and multidimensional arrays

"I've learned how to create objects and how to use decisions and loops to perform a variety of tasks with those objects," you say as you meet with Lynn Greenbrier at Event Handlers Incorporated. "Still, it seems as though I'm doing a lot of work. If I need to check a variable's value against 20 possibilities, it takes me 20 `if` statements or a long `switch` statement to get the job done. I thought computers were supposed to make things easier!"

"I think what you're looking for," Lynn says, "is how to use the power of arrays."

PREVIEWING A PROGRAM THAT USES ARRAYS AND STRINGS

The Chap8Events program demonstrates a variety of procedures that rely on arrays or Strings for efficient execution. You will answer questions when prompted.

To preview the Chap8Events program:

1. Start your text editor, open the **Chap8Event.java** file in the Chapter.08 folder on your Student Disk, and then examine the code. This program is a simple Event class similar to one you created in Chapter 5.
2. At the command prompt, compile the class by typing the command `javac Chap8Event.java`.
3. In the text editor, open the **Chap8Events.java** file and examine the code. This program is divided into two parts to demonstrate two of the major concepts you will learn about in this chapter. In the first part of the program, you enter codes for five upcoming events to be handled by Event Handlers Incorporated. The program stores all five events and displays them for you. The program prompts you to specify an Event type by entering a C, P, or N. You can enter any other character, but you must enter a valid character (C, P, or N) for each of the five events before the program will proceed. After you enter five valid characters, you will see a summary of the five events. The second part of the program will prompt you for the number of guests at your event. If you enter a value over 100, you will see a message regarding a surcharge.
4. At the command prompt, compile this file by typing the command:

   `javac Chap8Events.java`. Then run the program by typing the command `java Chap8Events`. Test the program by following the on-screen directions; you can press [Ctrl] and C to stop the program at any time.

In this chapter, you will write programs that are similar to the two parts of this program.

DECLARING AND INITIALIZING AN ARRAY

While completing the first five chapters in this book, you stored values in variables. In the early sections, you simply stored a value and used it. In Chapter 6, you created loops that allow you to "recycle" variables; that is, after creating a variable, you can assign a value, use the value, and then, in successive cycles through the loop, reuse the variable as it holds different values.

There are times, however, when storing just one value at a time in memory does not meet your needs. For example, a sales manager who supervises 20 employees might want to determine whether each employee has produced sales above or below the average amount. When you enter the first employee's sales figure into a program, you can't determine whether it is above or below average, because you don't know what the average is until you have all 20 figures. Unfortunately, if you assign 20 sales figures to the same variable, when you assign the figure for the second employee, it replaces the figure for the first employee.

A possible solution is to create 20 separate employee sales variables, each with a unique name, so you can store all the sales until you can determine an average. A drawback to this method is that if you have 20 different variable names to be assigned values, then you need 20 separate assignment statements. For 20 different variable names, the statement that calculates total sales will be unwieldy, such as `total = firstAmt + secondAmt + thirdAmt + ...`. This method might work for 20 salespeople, but what if you have 10,000 salespeople?

The best solution is to create an array. An **array** is a named list of data items that all have the same type. You declare an array variable in the same way as you declare any scalar variable, but you insert a pair of square brackets after the type. For example, to declare an array of double values to hold sales figures for salespeople, you write `double[] salesFigure;`.

Tip

You can also declare an array variable by placing the square brackets after the array name, as in `double salesFigure[];`. This format is familiar to C and C++ programmers, but the preferred format among Java programmers is to place the brackets following the variable type and before the variable name, as in `double[] salesFigure;`.

After you create an array variable, you still need to reserve memory space. You use the same procedure to create an array that you use to create an object. Recall that when you create a class named Employee, you can declare an Employee object with a declaration such as `Employee oneWorker;`, but that declaration does not actually create the oneWorker object. You create the oneWorker object when you use the keyword `new` and the constructor method, as in `oneWorker = new Employee();`. Similarly, declaring an array and reserving memory space for it are two distinct processes. To reserve memory locations for 20 salesFigure objects, you declare the array variable with `double[] salesFigure;`, and then you create the array with `salesFigure = new double[20];`. Just as with objects, you can declare and create an array in one statement with `double[] salesFigure = new double[20];`.

Tip

Other languages, such as COBOL, BASIC, and Visual Basic, use parentheses rather than brackets to refer to individual array elements. By using brackets, the creators of Java made it easier for you to distinguish arrays from methods.

The statement `double[] salesFigure = new double[20];` reserves 20 memory locations for 20 salesFigures. You can distinguish each salesFigure from the others with a subscript. A **subscript** is an integer contained within square brackets that indicates one of an array's variables, or **elements**. In the Java programming language, any array's elements are numbered beginning with zero, so you can legally use any subscript from zero through 19 when working with an array that has 20 elements. In other words, the first salesFigure array element is `salesFigure[0]` and the last salesFigure element is `salesFigure[19]`.

It is common to forget that the first element in an array is element zero, especially if you know another programming language in which the first array element is element one. Making this mistake means you will be "off by one" in your use of any array.

Tip: To remember that array elements begin with element zero, it might help if you think of the first array element as being "zero elements away from" the beginning of the array, the second element as being "one element away from" the beginning of the array, and so on.

When you work with any individual array element, you treat it no differently than you would treat a single variable of the same type. For example, to assign a value to the first salesFigure in an array, you use a simple assignment statement, such as `salesFigure[0] = 2100.00;`. To print the last salesFigure in an array of 20, you write `System.out.println(salesFigure[19]);`.

Next you will create a small array to see how arrays are used. The array will hold salaries for four categories of employees.

To create a program that uses an array:

1. Open a new text file in your text editor.
2. Begin the class that will demonstrate array use by typing the following class and main() headers and their corresponding opening curly braces:

   ```
   public class DemoArray
   {
     public static void main(String[] args)
     {
   ```

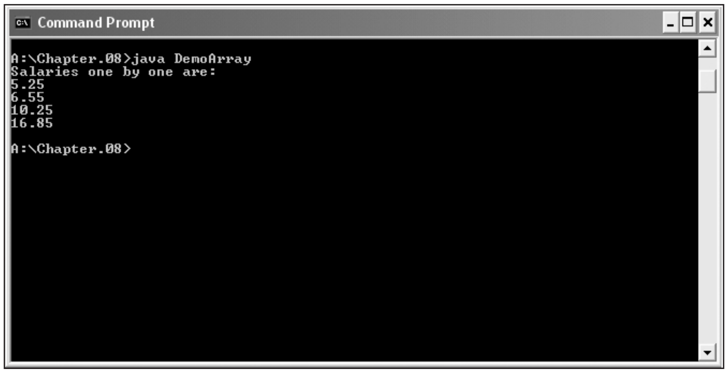
3. On a new line, declare and create an array that can hold four double values by typing **`double[] salary = new double[4];`**.
4. One by one, assign four values to the four salary array elements by typing the following:

   ```
   salary[0] = 5.25;
   salary[1] = 6.55;
   salary[2] = 10.25;
   salary[3] = 16.85;
   ```

5. To confirm that the four values have been assigned, print the salaries, one by one, using the following code:

   ```
   System.out.println("Salaries one by one are:");
   System.out.println(salary[0]);
   System.out.println(salary[1]);
   System.out.println(salary[2]);
   System.out.println(salary[3]);
   ```

6. Add the two closing curly braces that end the main() method and the DemoArray class.
7. Save the program as **DemoArray.java** in the Chapter.08 folder on your Student Disk.
8. Compile and run the program. The program's output appears in Figure 8-1.



```
A:\Chapter.08>java DemoArray
Salaries one by one are:
5.25
6.55
10.25
16.85

A:\Chapter.08>
```

Figure 8-1 Output of the DemoArray program

Initializing an Array

A variable that has a primitive type, such as int, holds a value. A variable with a reference type, such as an array, holds a memory address where a value is stored.

Array names represent computer memory addresses; that is, array names are references, as are all Java objects. When you declare an array name, no computer memory address is assigned to it. Instead, the array variable name has the special value `null`, or Unicode value '\u0000'. When you declare `int[] someNums;`, the variable someNums has a value of `null`.

When you define someNums as `int[] someNums = new int[10];`, then someNums has an actual memory address value. Each element of someNums has a value of zero because someNums is a numeric array. By default, character array elements are assigned '\u0000'. Boolean array elements are automatically assigned `false`.

You already know how to assign a different value to a single element of an array, as in `someNums[0] = 46;`. You also can assign nondefault values to array elements upon creation. To initialize an array, you use a list of values separated by commas and enclosed within curly braces. For example, if you want to create an array named tenMult and store the first six multiples of 10 within the array, you can declare `int[] tenMult = {10, 20, 30, 40, 50, 60};`. When you initialize an array by giving it values upon creation, you do not give the array a size—the size will be assigned based on the number of values you place in the initializing list. Also, when you initialize an array, you do not need to use the keyword `new`; instead, new memory is assigned based on the length of the list of provided values.

Tip

In Java, you do not usually use a semicolon after a closing curly brace, for example, at the end of a method body. However, every statement in Java requires a semicolon, and an array initialization is a statement. Remember to type the semicolon after the closing brace at the end of an array's initialization list.

Next you will alter your DemoArray program to initialize the array of doubles, rather than declaring the array and assigning values later.

To initialize an array of doubles:

1. Open the **DemoArray.java** file in your text editor. Change the class name to **`DemoArray2`**. Delete the statement that declares the array of four doubles named salary, **`(double[] salary = new double[4];)`**, and then replace it with the following initialization statement:

   ```
   double[] salary = {5.25, 6.55, 10.25, 16.85};
   ```

2. Delete the following four statements that individually assign the values to the array:

   ```
   (salary[0] = 5.25; salary[1] = 6.55; salary[2] = 10.25;
      salary[3] = 16.85;)
   ```

3. Save the file as **DemoArray2.java**, compile, and test the program. The output is the same as that shown in Figure 8-1, except the program name displayed would be DemoArray2.

USING SUBSCRIPTS WITH AN ARRAY

If you treat each array element as an individual entity, then there isn't much of an advantage to declaring an array over declaring individual scalar (primitive) variables, such as int, double, or char. The power of arrays becomes apparent when you begin to use subscripts that are variables, rather than subscripts that are constant values.

For example, when you declare a price array of five integers, such as `int[] priceArray = {2, 14, 35, 67, 85};`, you often want to perform the same operation on each array element, such as increasing the price a constant amount. To increase each price array element by three dollars, for example, you can write the following:

```
priceArray[0] += 3;
priceArray[1] += 3;
```

```
priceArray[2] += 3;
priceArray[3] += 3;
priceArray[4] += 3;
```

With five price array elements, this task is manageable. However, you can shorten the task by using a variable as the subscript. Then you can use a loop to perform arithmetic on each array element in the array, as in the following example:

```
for(sub = 0; sub < 5; ++sub)
  priceArray[sub] += 3;
```

The variable sub is set to zero, and then it is compared to five. Because the value of sub is less than five, the loop executes and three is added to priceArray[0]. Then the variable sub is incremented and it becomes one, which is still less than five, so when the loop executes again, priceArray[1] is increased by three, and so on. A process that took five statements now takes only one. Additionally, if the array had 100 elements, the first method of increasing the array values by three would result in 95 additional statements. The only change required in the `for` loop would be to compare sub to 100 instead of to five.

Next you will modify the DemoArray program to use a `for` loop with the array.

To use a for loop with the array:

1. Open the **DemoArray2.java** file in your text editor. Change the class name to **`DemoArray3`**. Delete the four println() statements that print the four array values, and then replace them with the following `for` loop:

   ```
   for(int x = 0; x < 4; ++x)
     System.out.println(salary[x]);
   ```

2. Save the program as **DemoArray3.java**, compile, and run the program. Again, the output is the same as that shown in Figure 8-1, except the program name displayed would be DemoArray3.

Declaring an Array of Objects

Just as you can declare arrays of integers or doubles, you can declare arrays that hold elements of any type, including objects. For example, assume you created the Employee class shown in Figure 8-2. This class has two data fields (empNum and empSalary), a constructor, and a get method for each field.

```
public class Employee
{
   private int empNum;
   private double empSal;
   Employee(int e, double s)
   {
      empNum = e;
      empSal=s;
   }
   public int getEmpNum()
   {
      return empNum;
   }

   public double getEmpSal()
   {
      return empSal;
   }
}
```

Figure 8-2 A simple Employee class

You can create separate Employee objects with unique names, such as Employee painter, electrician, or plumber, but for many programs it is far more convenient to create an array of Employees. An array named emp that holds seven Employees is defined as `Employee[] emp = new Employee[7];`. This statement reserves enough computer memory for seven Employee objects named emp[0] through emp[6]. However, the statement does not actually construct those Employee objects; instead, you must call the seven individual constructors. According to the class definition shown in Figure 8-2, the Employee constructor requires two arguments: an employee number and a salary. If you want to number your Employees 101, 102, 103, and so on, and start each Employee at a salary of $5.35, then the loop that constructs seven Employee objects is as follows:

```
for(int x = 0; x < 7; ++x)
 emp[x] = new Employee(101 + x, 5.35);
```

As x varies from 0 through 6, each of the seven emp objects is constructed with an employee number that is 101 more than x, and each of the seven emp objects holds the same salary of 5.35.

To use a method that belongs to an object that is part of an array, you insert the appropriate subscript notation after the array name and before the dot that precedes the method name. For example, to print data for seven Employees stored in the emp array, you can write the following:

```
for(int x = 0; x < 7; ++x)
 System.out.println
  (emp[x].getEmpNum()+ " " + emp[x].getEmpSal());
```

Pay attention to the syntax of the Employee objects' method calls, such as `emp[x].getEmpNum()`. Although you might be tempted to place the subscript at the end of the expression after the method name, as in `emp.getEmpNum[x]`, you cannot—the values in x (0 through 6) refer to a particular emp, each of which has access to a single getEmpNum() method. Placement of the bracketed subscript so it follows emp means the method "belongs" to a particular emp.

Next you will create an array of Event2 objects for Event Handlers Incorporated.

To create an array of Event2 objects:

1. Open the **Event.java** file from the Chapter.08 folder on your Student Disk and change the class name to **`Event2`**. This program is the same one that you created in Chapter 5. As you examine the code, recall that the class contains two data fields: a character representing the type of event, and a double representing the minimum that is charged for the event. The constructor requires values for the two data fields. The class also contains methods to get the field values. Save the file as **Event2.java** in the Chapter.08 folder on your Student Disk.

2. Open a new text file in your text editor to create an EventArray program. Type the following class header, the main() method header, and their opening curly braces:

```
public class EventArray
{
 public static void main(String[] args)
 {
```

3. Declare an array of five Event2 objects using the following code. You will also declare an integer that can be used as a subscript.

```
Event2[] someEvents = new Event2[5];
int x;
```

4. Enter the following `for` loop that calls the Event2 constructor five times, making each Event type 'X' with a minimum charge of 0.0:

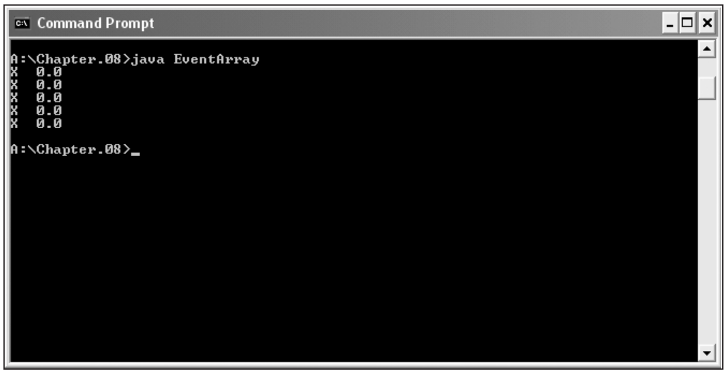
```
for(x = 0; x < 5; ++x)
 someEvents[x] = new Event2('X',0.0);
```

5. To confirm that the Event2 objects have been created, print their values by typing the following:

```
for(x = 0; x < 5; ++x)
System.out.println(someEvents[x].getEventType() +
 "  " + someEvents[x].getEventMinRate());
```

6. Add the two curly braces that end the main() method and the class definition.

7. Save the program as **EventArray.java** in the Chapter.08 folder. Compile and run the program. Figure 8-3 shows the program's output.



```
A:\Chapter.08>java EventArray
X  0.0
X  0.0
X  0.0
X  0.0
X  0.0

A:\Chapter.08>_
```

Figure 8-3 Output of the EventArray program

An array of five Event objects—each of which has the same event type and fee—is not very interesting or useful. Next you will modify the EventArray program so that it creates the events interactively and so each event possesses unique properties.

To create an interactive EventArray2 program:

1. Open the **EventArray.java** file in your text editor. Change the class name to **`EventArray2`**. Position the insertion point to the right of the opening curly brace of the EventArray2 class, press **[Enter]**, and then type the following statements to declare three constants for the corporate, private, and nonprofit event rates.

```
static final double CORP_RATE = 75.99;
static final double PRI_RATE = 47.99;
static final double NON_PROF_RATE = 40.99;
```

2. The program will accept keyboard input, so position the insertion point to the right of the main() method header, and then type **`throws Exception`**.

3. Just before the `for` statement that constructs five events, add the following two new variables that will hold an event type and rate, and initialize them with dummy values:

```
char event = 'Z';
double rate = 0;
```

4. Within the `for` loop, remove the line that constructs events with type 'X' and fee 0.0, `(someEvents[x] = new Event('X',0.0);)`, and then replace the line with the following block that prompts the user for one of three event types and constructs an appropriate event based on the value entered:

```
{
 System.out.println("Enter event type");
 System.out.println("C for corporate");
 System.out.println("P for private");
 System.out.println("N for non-profit");
 event = (char)System.in.read();
 System.in.read(); System.in.read();
  // Absorbs Enter key
 if(event == 'C')
  rate = CORP_RATE;
 else if(event == 'P')
  rate = PRI_RATE;
 else rate = NON_PROF_RATE;
 someEvents[x] = new Event2(event, rate);
}
```

5. Change the body of the last `for` loop as follows so it prints an event number along with the event information:

```
System.out.println("Event " + (x + 1) + "   " +
 someEvents[x].getEventType()+ "   " +
 someEvents[x].getEventMinRate());
```

Tip: At this point, when you run the program, if you enter an event type that is not C or P, the program assumes that the rate is nonprofit by default.

6. Save the program as **EventArray2.java**, compile, and run the program several times. Confirm that no matter what combination of C, P, and N you use for data entry, the list of events is stored and displayed correctly.

Searching an Array for an Exact Match or a Range Match

When you want to determine whether some variable holds one of many valid values, one option is to use a series of `if` statements to compare the variable to a series of valid values. Suppose that a company manufactures 10 items. When a customer places an order for an item, you need to determine whether the item number on the order form is valid. If valid item numbers are sequential, such as 101 through 110, then the following simple `if` statement that uses a logical AND can verify the order number and set a Boolean field to true: `if(itemOrdered >= 101 && itemOrdered <= 110) validItem = true;`. If the valid item numbers are nonsequential, for example, 101, 108, 201, 213, 266, 304, and so on, you must code the following deeply nested `if` statement or a lengthy OR comparison to determine the validity of an item number:

```
if(itemOrdered == 101)
  validItem = true;
else if(itemOrdered == 108)
  validItem = true;
else if(itemOrdered == 201)
  validItem = true;
// and so on
```

Instead of a long series of `if` statements, a more elegant solution is to compare the itemOrdered variable to a list of values in an array. You can initialize the array with the valid values with the following statement:

```
int[] validValues = {101, 108, 201, 213, 266,
             304, 311, 409, 411, 412};
```

Then you can use a `for` statement to loop through the array, and set a Boolean variable to `true` when a match is found:

```
for(int x = 0; x < 10; ++x)
{
 if(itemOrdered == validValues[x])
  validItem = true;
}
```

This simple `for` loop replaces the long series of `if` statements. Also, if a company carries 1,000 items instead of 10, then the only part of the `for` statement that changes is the comparison in the middle. As an added bonus, if you set up another parallel array with the same number of elements and corresponding data, you can use the same subscript to access additional information. For example, if the 10 items your company carries have 10 different prices, then you can set up an array to hold those prices: `double[] prices = {0.89, 1.23, 3.50, 0.69...};`. The prices must appear in the same order as their corresponding item numbers in the validValues array. Now the same `for` loop that finds the valid item number also finds the price, as shown in Figure 8-4. In other words, if the item number is found in the second position in the validValues array, then you can find the correct price in the second position in the prices array.

Tip

If you initialize parallel arrays, it is convenient to use spacing so that the values that correspond to each other visually align on the screen or printed page.

```
int[] validValues = {101, 108, 201, 213, 266,
                     304, 311, 409, 411, 412};
double[] prices =   {0.89, 1.23, 3.50, 0.69, 5.79,
                     3.19, 0.99, 0.89, 1.26, 8.00};
for(int x = 0; x <10; ++x)
{
   if(itemOrdered == validValues[x])
   {
      validItem = true;
      itemPrice = prices[x];
   }
}
```

Figure 8-4 Accessing information in parallel arrays

Tip

In an array with many possible matches, it is most efficient to place the more common items first, so they are matched right away. For example, if item 311 is ordered most often, place 311 first in the validValues array, and its price ($0.99) first in the prices array.

Within the code shown in Figure 8-4, you compare every itemOrdered with each of the 10 validValues. Even when an itemOrdered is equivalent to the first value in the validValues array (101), you always make nine additional cycles through the array. On each of these nine additional cycles, the comparison between itemOrdered and validValues[x] is always `false`. As soon as a match for an itemOrdered is found, it is most efficient to break out of the `for` loop early. An easy way to accomplish this is to set x to a high value within the block of statements executed when there is a match. Then, after a match, the `for` loop will not execute again because the limiting comparison (x< 10) will have been surpassed. Figure 8-5 shows this program.

```
for(x = 0; x < 10; ++x)
{
   if(itemOrdered == validValues[x])
   {
      validItem = true;
      itemPrice = prices[x];
      x = 10; // Break out of loop when you find a match
   }
}
```

Figure 8-5 Breaking out of a `for` loop early

Tip

Instead of the statement that sets x to 10 when a match is found, in its place you could place a `break` statement within the loop.

Tip

Breaking out of a `for` loop early, whether you do it by setting a variable's value or by using a `break` statement, disrupts the loop flow and makes the code harder to understand. If you (or your instructor) agree with this philosophy, then consider using a method that employs a `while` statement, as described next.

You can choose to forgo the `for` loop entirely and, as an alternative use, a `while` loop to search for a match. Using this approach, you set a subscript to zero and while the itemOrdered is not equal to a value in the array, you increase the subscript and keep looking. You search only while the subscript remains lower than the number of elements in the array. If the subscript increases to 10, then you never found a match in the 10-element array. If the loop ends before the subscript reaches 10, then you found a match and the correct price can be assigned to the itemPrice variable. Figure 8-6 shows this programming approach.

```
int x = 0;
while(x < 10 && itemOrdered != validValues[x])
   ++x;
if(x != 10)
{
   validItem = true;
   itemPrice = prices[x];
}
```

Figure 8-6 Searching with a `while` loop

Next you will delete the `if` statements that determine a price for each event at Event Handlers Incorporated and replace them with an array search.

To determine event pricing using parallel arrays:

1. Open the **EventArray2.java** file in your text editor. Change the class name to **`EventArray3`**. Position the insertion point to the right of the statement that declares the rate variable `(double rate=0;)`, press **[Enter]** to start a new line of text, and then type the following character array to hold the codes for the three allowed event types:

   ```
   char[] eventCode = {'C', 'P', 'N'};
   ```

2. Press **[Enter]**, and then add the following code to create a double array to hold the rates charged for the three event types:

   ```
   double[] eventRate = {CORP_RATE, PRI_RATE, NON_PROF_RATE};
   ```

Tip

Notice that you can use symbolic constants as well as literal constants as array elements. You can even combine the two types of constants within the same array and use variable names as array elements. Don't forget, however, that all elements within a single array must have the same type.

3. Remove the five lines of code (beginning with `if(event == 'C')...)`) that constitute the `if...else` structure that determines the event rate. This `if` structure is no longer needed. Replace it with the following `for` loop that searches through the eventCode array, and, upon finding a match, selects a price from the eventRate3 array:

   ```
   for(int i = 0; i < 3; ++i)
   {
    if(event == eventCode[i])
      rate = eventRate[i];
   }
   ```

4. Save the program as **EventArray3.java**, compile, and run the program. Confirm that, just as before, no matter what combination of C, P, and N you use for data entry, the list of events is stored and displayed correctly.

When you run the program, if you enter an invalid event code, an Event object is created and an incorrect rate is assigned to the Event object. Now that you have an array of valid event codes, it is simple to disallow any invalid event codes (those other than C, P, or N).

To force all five Event objects to contain valid codes and rates:

1. Open the **EventArray3** file, if necessary, and change the class name to **`EventArray4`**. Within the EventArray4.java program, position the insertion point to the right of the `double rate = 0;` variable declaration, press **[Enter]**, and then enter the following code to create a Boolean variable named codeIsValid: **`boolean codeIsValid;`**.
2. Position the insertion point at the beginning of the `for` statement that begins the search through the eventCode array `(for(int i = 0; ...))`, and then press **[Enter]** to start a new line. Just before the `for` loop, you must ensure that the codeIsValid variable is set to `false` by typing **`codeIsValid = false;`**.
3. Within the `for` loop, change the `if` statement that checks the eventCode array as follows, so that if the event variable is equivalent to one of the eventCodes, then a block of two statements will execute—besides setting the rate, the block sets the codeIsValid variable to `true`:

```
if(event == eventCode[i])
{
 rate = eventRate[i];
 codeIsValid = true;
}
```

You can make the program more efficient by breaking out of the `for` loop early when an event matches an eventCode array element. Set the loop control variable i to a high value when a match is found.

4. Place the insertion point to the right of `codeIsValid = true;`, press **[Enter]** to start a new line, and then type **`i = 3;`**.
5. Position the insertion point at the beginning of the statement that constructs one of the five someEvents objects `(someEvents[x] = new Event2 (event, rate);)`. Press **[Enter]** to start a new line, and then insert the following condition so the object is created only if the code is valid: **`if(codeIsValid)`**. To show clearly that the assignment statement depends on the `if` statement, insert two spaces at the beginning of the line containing `someEvents[x] = new Event2(event, rate);`.
6. Position the insertion point just after `someEvents[x] = new Event2 (event, rate);`, and press **[Enter]** to start a new line. Enter the following `else` clause that reduces x:

```
else
  --x;
```

Now, for example, if a code is not valid on the third pass through the loop when x is 2, x will be decremented to 1. At the top of the `for` loop (in the third section within the parentheses), x is increased to 2 again for the next pass through the `for` loop. So if the user enters a valid code during this fourth execution of the loop, x still will be 2, and an object will correctly be created at someEvent[2].

7. Save the program as **EventArray4.java**, compile, and run the program. Enter as many valid and invalid codes as you like. After five of the codes you enter are identified as valid, the five constructed objects will display.

Searching an Array for a Range Match

Searching an array for an exact match is not always practical. Suppose your company gives customer discounts based on the quantity of items ordered. Perhaps no discount is given for any order of fewer than a dozen items, but there are increasing discounts available for orders of increasing quantities, as shown in Table 8-1.

Table 8-1 Discount table

Total Quantity Ordered	Discount
1 to 12	none
13 to 49	10%
50 to 99	14%
100 to 199	18%
200 or more	20%

One awkward option is to create a single array to store the discount rates. You could use a variable named numOfItems as a subscript to the array, but the array would need hundreds of entries, for example, `double[] discount = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, .10, .10, .10 ...};`. When numOfItems is 3, for example, then discount[numOfItems] or discount[3] is 0. When numOfItems is 14, then discount[numOfItems] or discount[14] is .10. Because a customer might order thousands of items, the array would need to be ridiculously large.

Tip

Notice that there are 13 zeroes listed in the discount array in the preceding example. The first array element has a zero subscript (and a zero discount for zero items). The next 12 discounts (1 through 12 items) are also discounts of zero.

A better option is to create parallel arrays. One array will hold the five discount rates, and the other array will hold five discount range limits. The Total Quantity Ordered column in Table 8-1 shows five ranges. If you use only the first figure in each range, you can create an array that holds five low limits: `int[] discountRangeLimit= {1, 13, 50, 100, 200};`. A parallel array will hold the five discount rates: `double[] discount = {0, .10, .14, .18, .20};`. Then, starting at the last discountRangeLimit array element, for any numOfItems greater than or equal to discountRangeLimit[4], the appropriate discount is discount[4]. In other words, for any numOf Items less than discountRangeLimit[4], you should decrement the subscript and look in a lower range. Figure 8-7 shows the code.

```
int[] discountRangeLimit = {1, 13, 50, 100, 200};
double[] discount =        {0, .10, .14, .18, .20};
double customerDiscount;
int sub = 4;
while(sub >= 0 && numOfItems < discountRangeLimit[sub])
 --sub;
customerDiscount = discount[sub];
```

Figure 8-7 Searching an array of ranges

Tip

It is a good programming practice to make sure that the subscript does not fall below zero in the statement `while(sub >= 0 && numOfItems < discountRangeLimit[sub])`. Although this would happen only if a numOfItems held a negative value, such a check will prevent a program error.

Passing Arrays to Methods

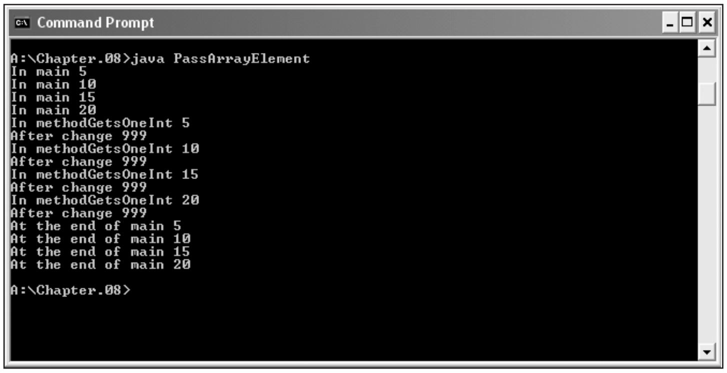
You have already seen that you can use any individual array element in the same manner as you would use any single variable of the same type. That is, if you declare an integer array as `int[] someNums = new int[12];`, then you can subsequently print someNums[0] or add one to someNums[1], just as you would for any integer. Similarly, you can pass a single array element to a method in exactly the same manner as you would pass a variable.

Examine the program shown in Figure 8-8. The program creates an array of four integers and prints them. Then the program calls the methodGetsOneInt() method four times, passing each element in turn. The method prints the number, changes the number to 999, and then prints the number again. Finally, back in main() method, the four numbers are printed again.

As you can see in Figure 8-9, the four numbers that were changed in the methodGetsOneInt() method remain unchanged back in main(). The variable named one is local to the methodGetsOneInt() method, and any changes to variables passed into the method are not permanent and are not reflected in the array in the main() program. Each variable named one in the methodGetsOneInt() method holds only a copy of the array element passed into the method.

```
public class PassArrayElement
{
   public static void main(String[] args)
   {

      int[] someNums = {5, 10, 15, 20};
      int x;
      for(x = 0; x < 4; ++x)
         System.out.println("In main " + someNums[x]);
      for(x = 0; x < 4; ++x)
         methodGetsOneInt(someNums[x]);
      for(x = 0; x < 4; ++x)
         System.out.println("At end of main " + someNums[x]);
      }
   public static void methodGetsOneInt(int one)
   {
      System.out.println("In methodGetsOneInt " + one);
      one = 999;
      System.out.println("After change " + one);
   }
}
```

Figure 8-8 PassArrayElement program



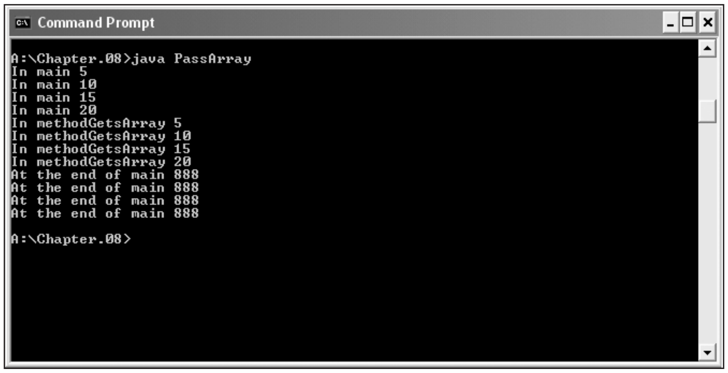
```
A:\Chapter.08>java PassArrayElement
In main 5
In main 10
In main 15
In main 20
In methodGetsOneInt 5
After change 999
In methodGetsOneInt 10
After change 999
In methodGetsOneInt 15
After change 999
In methodGetsOneInt 20
After change 999
At the end of main 5
At the end of main 10
At the end of main 15
At the end of main 20

A:\Chapter.08>
```

Figure 8-9 Output of the PassArrayElement program

The outcome is quite different when you pass an entire array to a method. Arrays, like all objects, are passed by reference, which, as you will recall from Chapter 4, means that the method receives the actual memory address of the array and has access to the actual values in the array elements. The program shown in Figure 8-10 creates an array of four integers. After the integers print, the entire array is passed to a method named methodGetsArray(). Within the method, the numbers print, which shows that they retain their values from main(), but then the value 888 is assigned to each number. Even though the methodGetsArray() method is a void method (meaning nothing is returned to the main() method), when the program prints the array for the second time within main(), all of the values have been changed to 888, as you can see in Figure 8-10. Because arrays are passed by reference, the methodGetsArray() method "knows" the address of the array declared in main() and makes its changes directly to the original array that was declared in the main() method. Figure 8-11 shows the output of the PassArray program.

```
public class PassArray
{
   public static void main(String[] args) throws Exception
   {
      int[] someNums = {5, 10, 15, 20};
      int x;
      for(x = 0; x < 4; ++x)
         System.out.println("In main " + someNums[x]);
      methodGetsArray(someNums);
      for(x = 0; x < 4; ++x)
         System.out.println("At end of main " + someNums[x]);
      }
   public static void methodGetsArray(int[] arr)
   {
      for(int y=0; y<4; ++y)
      {
         System.out.println("In methodGetsArray " + arr[y]);
         arr[y] = 888;
      }
   }
}
```

Figure 8-10 PassArray program



```
A:\Chapter.08>java PassArray
In main 5
In main 10
In main 15
In main 20
In methodGetsArray 5
In methodGetsArray 10
In methodGetsArray 15
In methodGetsArray 20
At the end of main 888
At the end of main 888
At the end of main 888
At the end of main 888

A:\Chapter.08>
```

Figure 8-11 Output of the PassArray program

Next you will add a new method to the Event object class type. Then you will add steps to the EventArray program so you can pass an array of Event objects to a method. This program will demonstrate that changes made within the method permanently affect values in the array.

To add a new method to the Event class:

1. In your text editor, open the **Event2.java** file from the Chapter.08 folder on your Student Disk. This text file contains the class definition for Event2 objects. Rename the class **`Event3`** and change the Constructor name to **`Event3`**.
2. Add a new setEventMinRate() method that you can use to alter an Event object's eventMinRate. Position the insertion point to the left of the final closing curly brace for the Event2 class, press **[Enter]** to start a new line above the closing brace, and then enter the following setEventMinRate() method:

```
public void setEventMinRate(double rate)
{
 eventMinRate = rate;
}
```

3. Save the file as **Event3.java** and compile the program.

Next you will add a method call and a method to the EventArray4 program. The method will receive an array of Event3 objects and increase the rate for each event by $5.00.

To add a method call and a method to the EventArray4 program:

1. Open the **EventArray4.java** file in your text editor. Change the class name to **`EventArray5`**. Find the statement `Event2 [] someEvents = new Event2 [5];`, and change the statement to **`Event3[] someEvents = new Event3[5];`**. Position the insertion point to the left of the closing curly brace for the main() method in this EventArray5 class, and then press **[Enter]** to insert a new blank line above the closing curly brace. Locate the statement that creates the someEvents array `someEvents[x] = new Event2(event,rate)` and change Event2 to **Event3**.
2. Enter the following method call to a raiseRates() method, which will receive the someEvents array and raise each Event's rate by $5.00:

```
raiseRates(someEvents);
```

3. Press **[Enter]**. To demonstrate that the rates changed as a result of the raiseRates() method, add the following print loop on the new line:

```
for(x = 0; x < 5; ++x)
 System.out.println("Event " + (x + 1) + "  " +
  someEvents[x].getEventType()+ "  " +
  someEvents[x].getEventMinRate());
```

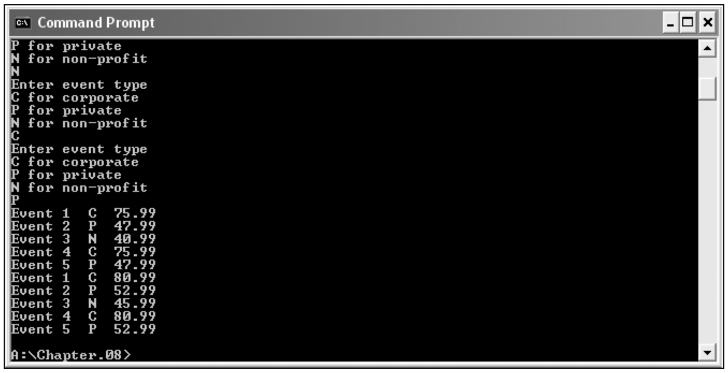
Tip: If you do not want to type this statement, you can simply use your text editor's copy function to copy the identical statement that already exists within the program.

4. Place the insertion point to the right of the main() method's closing curly brace, and then press **[Enter]** to start a new blank line before the closing brace for the EventArray5 program. Then enter the following raiseRates() method. The method loops through the array five times. With each iteration, the method gets the array element's current rate, stores it in a temporary double variable, and adds $5.00 to the temporary variable. Then the temporary variable value is assigned back into the array object's rate with the setEventMinRate() method.

```
private static void raiseRates(Event3[] event)
{
 double temp;
 for(int q = 0; q < 5; ++q)
 {
  temp = event[q].getEventMinRate();
  temp += 5;
  event[q].setEventMinRate(temp);
 }
}
```

Tip: You can replace the three statements `temp = event[q].getEventMinRate();`, `temp += 5;`, and `event[q].setEventMinRate(temp);` with one statement: `event[q].setEventMinRate(event[q].getEventMinRate() + 5);`. If this method call within a method call is clear to you, feel free to use it.

5. Save the program as **EventArray5.java**, then compile and run it. After you answer the prompts to create five events, the old rates and the new increased rates should display on the screen. Figure 8-12 shows a sample run.



```
P for private
N for non-profit
N
Enter event type
C for corporate
P for private
N for non-profit
C
Enter event type
C for corporate
P for private
N for non-profit
P
Event 1  C  75.99
Event 2  P  47.99
Event 3  N  40.99
Event 4  C  75.99
Event 5  P  47.99
Event 1  C  80.99
Event 2  P  52.99
Event 3  N  45.99
Event 4  C  80.99
Event 5  P  52.99

A:\Chapter.08>
```

Figure 8-12 Sample output of the EventArray5 program using the raiseRates() method

USING THE `length` FIELD

Every array object that you create is automatically assigned a data field named length. The **`length` field** contains the number of elements in the array. For example, when you declare `double[] salaries = new double[8];`, the field `salaries.length` is assigned the value 8.

When you work with array elements, you must ensure that the subscript you use remains in the range zero through one less than the length. To access all eight elements of a salaries array, for example, you can code the number 8 explicitly, as in `for(x = 0; x < 8; ++x)....` If you modify your program to hold more or fewer array elements, you must remember to change every appropriate reference to the array size within the program. Many text editors have a "find and replace" feature that lets you change every 8, but you must be careful not to change an 8 that has nothing to do with the array. A better technique is to use `salaries.length`, as in `for(x = 0; x < salaries.length; ++x)....`That way, if you change the size of the salaries array, the array will always use the correct maximum length.

Next you will remove the explicit 5 you have used in each `for` loop within the EventArray5 program, and replace each 5 with a reference to the `length` field.

To use the array `length` field:

1. Open the **EventArray5.java** program, if necessary change the class name to **`EventArray6`**, and locate the first `for` statement in the main() method of your EventArray6 program. This loop constructs five Event6 objects. Delete the 5 in the `for` expression and then replace that number with **`someEvents.length`**.
2. Locate the next `for` loop in this EventArray6 program, which checks the event codes entered by a user. Delete the **3** from the `for` loop and replace that number with **`eventCode.length`**.
3. Locate the next `for` loop, which prints array values. Delete the **5** in the `for` loop and replace that number with **`someEvents.length`**.
4. The last `for` loop in the main() method of this EventArray6 program prints the objects. Delete the **5** in this loop and replace it with **`someEvents.length`**.
5. Within the raiseRates() method, locate the `for` loop that raises the rate of each array element. Within the raiseRates() method, the array name is `event`. Delete the **5** in this `for` loop and replace it with **`event.length`**.
6. Save the program as **EventArray6.java**, compile, and run the program. The execution of the program should be the same as before—five objects are created and three valid event codes are checked.

Creating Arrays of Strings

As with any other object, you can create an array of Strings. For example, you can store three company department names as `String[] deptName = {"Accounting", "Human Resources", "Sales"};`. You can access these department names like any other array object. For example, to print them, you can use the following code:

```
for(int a = 0; a < deptName.length; ++a)
 System.out.println(deptName[a]);
```

Tip

Notice that `deptName.length;` refers to the length of the array deptName (three elements), and not to the length of any of the String objects stored in the deptName array. Each String object has access to a length() method that returns the length of a String. For example, if deptName[0] is "Accounting", then `deptName[0].length()` is 10 because "Accounting" contains ten characters.

Next you will create two arrays to hold event types and manager names for Event Handlers Incorporated. Then when a user enters an event type, the appropriate event type and manager name will display on the screen.

To add event types and manager names to the EventArray6 program:

1. Open the **EventArray6.java** file in your text editor and change the class name to **EventArray7**. Position the insertion point after the opening curly brace of the main() method, and then press **[Enter]** to start a new line.
2. Add the array of event types by entering the following:

```
String[] eventType =
 { "Corporate", "Private", "Non-Profit"};
```

3. Press **[Enter]** to start a new line, and then add the following array of manager names:

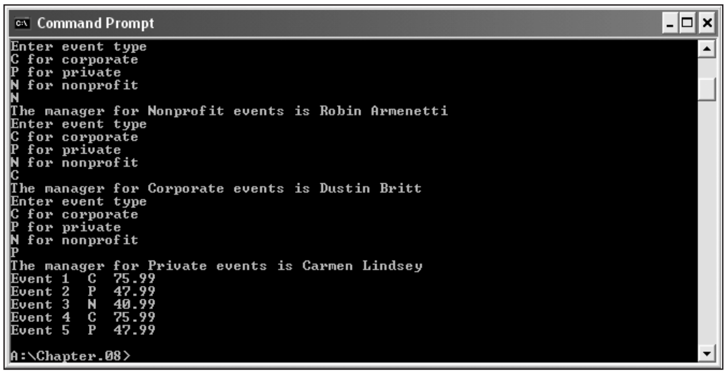
```
String[] managerName =
 {"Dustin Britt", "Carmen Lindsey", "Robin Armenetti"};
```

4. Locate the `for` loop that determines if the user entered a valid code. Place the insertion point after the statement `codeIsValid = true;` that appears within the `for` statement, and then press **[Enter]** to start a new line. At this point in the program, the variable i indicates the position of a correct event type in the eventCode array. The correct event type is in the same relative position in the eventType array as the correct manager's name is within the managerName array. In other words, managerName[i] "goes with" eventCode[i].

5. Type the following statement that prints the event type and manager's name on the new line:

```
System.out.println("The manager for " + eventType[i]
 + " events is " + managerName[i]);
```

6. To simplify screen output, comment out the call to raiseRates() as well as the lines that print the raised rates. Type // at the beginning of each of these lines.
7. Save the program as **EventArray7.java**, compile, and test the program. Your output should look like Figure 8-13.



```
Enter event type
C for corporate
P for private
N for nonprofit
N
The manager for Nonprofit events is Robin Armenetti
Enter event type
C for corporate
P for private
N for nonprofit
C
The manager for Corporate events is Dustin Britt
Enter event type
C for corporate
P for private
N for nonprofit
P
The manager for Private events is Carmen Lindsey
Event 1  C  75.99
Event 2  P  47.99
Event 3  N  40.99
Event 4  C  75.99
Event 5  P  47.99

A:\Chapter.08>
```

Figure 8-13 Output of the EventArray7 program showing event types and manager names

In Chapter 7 you learned about methods for comparing characters and comparing strings. You determined whether they were the same, or if different, which one was considered larger. With arrays, you often want to know whether a certain character or string can be found within the elements of the array. For example, does the letter z appear in an array of characters, or does the name John appear in the array of first names? The idea is to search the array to see if you can find an exact match. Next you will search an array of states, compare each state name to every element in the array, and determine whether either or both state names can be matched.

To compare a String to each element in an array:

1. Open a new text file in your text editor, and then enter the following opening lines for a FindState class:

```
public class FindState
{
 public static void main(String[] args)
 {
```

2. Enter the following array of String objects that holds the state names where Event Handlers Incorporated has offices:

```
String[] states = {"Alaska", "California", "Illinois",
 "Oregon", "Texas", "Wisconsin", "Wyoming"};
```

3. For testing purposes, assign the following two state names to two String objects named firstState and secondState:

```
String firstState = "Illinois";
 // This state will be found in the list
String secondState = "Ohio";
 // This state will not be found in the list
```

4. Next declare the following integer variable that you will use as a subscript, and a Boolean variable that you will set to `true` when a state name is found in the array:

```
int x;
boolean found = false;
```

5. Enter the following `for` loop that compares the firstState to each state in the array. When a match is found, set the Boolean found variable to `true`.

```
for(x = 0; x < states.length; ++x)
 if(firstState.equals(states[x]))
  found = true;
```

6. At the end of the loop, enter the following statements to print a statement indicating whether the firstState was found:

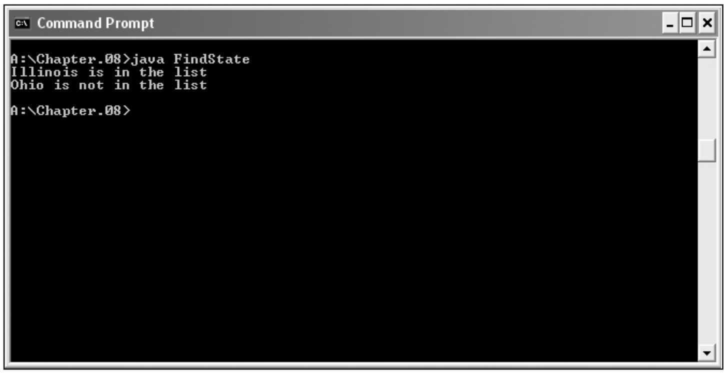
```
if(found)
 System.out.println(firstState + " is in the list");
else
 System.out.println(firstState + " is not in the list");
```

7. Now enter the following statements to reset the variable found to `false`, and then repeat the search process for the secondState variable:

```
found = false;
for(x = 0; x < states.length; ++x)
 if(secondState.equals(states[x]))
   found = true;
 if(found)
   System.out.println(secondState + " is in the list");
 else
   System.out.println(secondState + " is not in the list");
```

8. Enter the two closing curly braces that end the main() method and the FindState class.

9. Save the program as **FindState.java** in the Chapter.08 folder on your Student Disk. Compile and test the program. The program's output appears in Figure 8-14.



```
A:\Chapter.08>java FindState
Illinois is in the list
Ohio is not in the list

A:\Chapter.08>
```

Figure 8-14 Output of the FindState program

Sorting Primitive, Object, and String Array Elements

Sorting is the process of arranging a series of objects in some logical order. When you place objects in order, beginning with the object with the lowest value, you are sorting in **ascending** order; conversely, when you start with the object that has the largest value, you are sorting in **descending** order.

The simplest possible sort involves two values that are out of order. To place the values in order, you must swap the two values. Suppose that you have two variables—valA and valB—and further suppose that `valA = 16` and `valB = 2`. To exchange the values of the two variables, you cannot simply use the following code:

```
valA = valB; // 2 goes to valA
valB = valA; // 2 goes to valB
```

If valB is 2, then after you execute `valA = valB;`, both variables hold the value 2. The value 16 that was held in valA is lost. When you execute the second assignment statement, `valB = valA;`, each variable still holds the value 2.

The solution that allows you to retain both values is to employ a variable to hold valA's value temporarily during the swap:

```
temp = valA; // 16 goes to temp
valA = valB; // 2 goes to valA
valB = temp; // 16 goes to valB
```

Using this technique, valA's value (16) is assigned to the temp variable. The value of valB (2) is then assigned to valA, so valA and valB are equivalent. Then the temp value (16) is assigned to valB, so the values of the two variables finally are swapped.

If you want to sort any two values, valA and valB, in ascending order so that valA is always the lower value, then you use the following `if` statement to make the decision whether to swap. If valA is more than valB, you want to switch the values. If valA is not more than valB, you do not want the values to switch.

```
if(valA > valB)
{
 temp = valA;
 valA = valB;
 valB = temp;
}
```

Sorting two values is a fairly simple task; sorting more values (valC, valD, valE, and so on) is more complicated. Without the use of an array, sorting a series of numbers is a daunting task; the task becomes manageable when you know how to use an array.

As an example, you might have a list of five numbers that you want to place in ascending numeric order. One approach is to use a method popularly known as a bubble sort. To use a **bubble sort**, you place the original, unsorted values in an array, such as `int[] someNums = {88, 33, 99, 22, 54};`. After a series of comparisons and swaps, the numbers eventually will be placed in order within the array. You compare the first two numbers; if they are not in ascending order, you swap them. You compare the second and third numbers; if they are not in ascending order, you swap them. You continue down the list. Generically, for any someNums[x], if the value of someNums[x] is larger than someNums[x + 1], then you want to swap the two values.

With the numbers 88, 33, 99, 22, and 54, the process proceeds as follows:

1. Compare 88 and 33. They are out of order. Swap them. The list becomes 33, 88, 99, 22, 54.
2. Compare the second and third numbers in the list—88 and 99. They are in order. Do nothing.
3. Compare the third and fourth numbers in the list—99 and 22. They are out of order. Swap them. The list becomes 33, 88, 22, 99, 54.
4. Compare the fourth and fifth numbers—99 and 54. They are out of order. Swap them. The list becomes 33, 88, 22, 54, 99.

When you reach the bottom of the list, the numbers are not in ascending order, but the largest number, 99, has moved to the bottom of the list. This feature gives the bubble sort its name—the "heaviest" value has sunk to the bottom of the list as the "lighter" values have bubbled to the top.

Assuming b and temp both have been declared as integer variables, the code so far is as follows:

```
for(b = 0; b < 4; ++b)
 if(someNums[b] > someNums[b + 1])
 {
   temp = someNums[b];
   someNums[b] = someNums[b + 1];
   someNums[b + 1] = temp;
 }
```

Notice that the `for` statement tests every value of b from zero through three. The array someNums contains five integers. The subscripts in the array range in value from zero through four. Within the `for` loop, each someNums[b] is compared to someNums[b + 1], so the highest legal value for b is three when array element b (3) is compared to array element b + 1 (4). For a sort on any size array, the value of b must remain less than the array's length minus one.

The list of numbers that began as 88, 33, 99, 22, 54 is currently 33, 88, 22, 54, 99. You must perform the entire comparison-swap procedure again.

1. Compare the first two values—33 and 88. They are in order; do nothing.
2. Compare the second and third values—88 and 22. They are out of order. Swap them so the list becomes 33, 22, 88, 54, 99.
3. Compare the third and fourth values—88 and 54. They are out of order. Swap them so the list becomes 33, 22, 54, 88, 99.
4. Compare the fourth and fifth values—88 and 99. They are in order; do nothing.

After this second pass through the list, the numbers are 33, 22, 54, 88, and 99—close to ascending order, but not quite. You can see that with one more pass through the list, the values 22 and 33 would swap, and the list would finally be placed in order. With the worst-case list, one in which the original numbers were descending (as out-of-ascending order as they could possibly be), you would need to go through the list four times making comparisons and swaps. You always, at most, need to pass through the list as many times as its length minus one. Figure 8-15 shows the entire procedure.

```
for(a = 0; a < (someNums.length - 1); ++a)
   for(b = 0; b < (someNums.length - 1); ++b)
      if(someNums[b] > someNums[b + 1])
      {
         temp = someNums[b];
         someNums[b] = someNums[b + 1];
         someNums[b + 1] = temp;
      }
```

Figure 8-15 Ascending sort of the someNums array

Tip

To place the list in descending order, you need to make only one change in the method in Figure 8-15: You change the greater than sign (>) in `if(someNums[b] > someNums[b + 1])` to a less than sign (<).

Next you will begin to write a program that includes a method that sorts characters that you enter from the keyboard.

To write a program that sorts characters:

1. Open a new file in your text editor, and then type the following class header and main() method header for a SortCharArray program:

```
public class SortCharArray
{
 public static void main(String[] args) throws Exception
 {
```

2. Enter the following code to declare a character array that can hold 10 characters, and an integer x to use as a subscript with the array:

```
char[] someChars = new char[10];
int x;
```

3. Enter the following `for` loop that allows the user to enter 10 characters from the keyboard:

```
for(x = 0; x < someChars.length; ++x)
{
 System.out.print("Enter a character ");
 someChars[x] = (char)System.in.read();
 System.in.read(); System.in.read();
}
```

4. Enter the following `for` loop that displays the characters as originally entered:

```
System.out.println("Before sort");
for(x = 0; x < someChars.length; ++x)
 System.out.print(someChars[x] + " ");
```

5. Call a method named bubbleSort(). You will pass two arguments to bubbleSort()—the array and the length of the array.

```
bubbleSort(someChars, someChars.length);
```

6. Add a loop that prints the characters after the sort has executed:

```
System.out.println("\nAfter sort");
for(x = 0; x < someChars.length; ++x)
 System.out.print(someChars[x] + " ");
```

Tip

When you sort a series of numbers, you place them in arithmetic order. Characters are sorted by their numeric Unicode value.

7. Add a final println() statement to the program, as well as a closing curly brace for the main() method:

```
 System.out.println();
}
```

8. Save the file as **SortCharArray.java** in the Chapter.08 folder on your Student Disk. You cannot compile the file yet because you have not written the bubbleSort() method.

Next you will add the bubbleSort() method to the SortCharArray program. An advantage of creating the sort as a method which is separate from the main() method is that you can then use this method with other programs. The bubbleSort() method will sort any size array of characters; you might be able to use it in an application in which you have a number of characters to sort.

To write the bubbleSort() method:

1. Below the main() method in the existing SortCharArray program, write the header for a bubbleSort() method that takes a character array and an integer length as arguments, press **[Enter]**, and then type the method's opening curly brace:

```
public static void bubbleSort(char[] array, int len)
{
```

Tip: In Step 1 in the header method of bubbleSort, the parameters char[] array and int len are used to match the type of array and the length, or number of array elements contained in the array, that are to be passed.

2. Type the following code to declare two integers, a and b, to use in the method's `for` loops. Additionally, declare a temporary character variable.

```
int a, b;
char temp;
```

3. The two `for` loops you need to sort the array each must execute `len - 1` times. If you place the subtraction calculation within each `for` statement, as in `for(a = 0; a < (len - 1); ++a)`, the subtraction is performed on each cycle through the loop. It is more efficient to calculate `len - 1` once, store the value in a variable, and use the new variable in the `for` loops. Figure 8-16 shows this process. Add the code displayed in Figure 8-16 to your program.

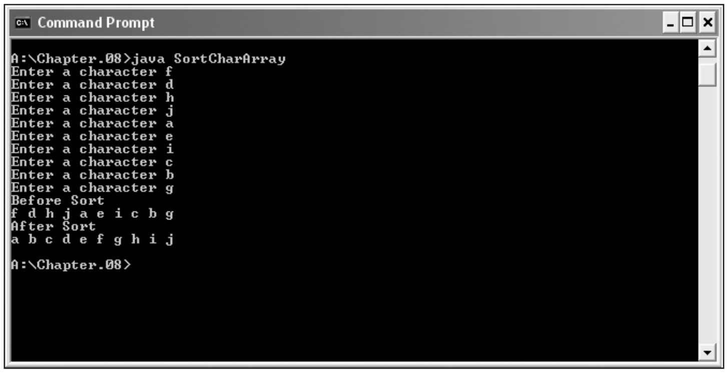
4. Add the closing curly brace for the bubbleSort() method and the closing curly brace for the SortCharArray class.

5. Save, compile, and execute the program. Figure 8-17 shows a typical program run.

When you use a bubble sort to sort any array into ascending order, the largest value "falls" to the bottom of the array after you have compared each pair of values in the array one time. The second time you go through the array making comparisons, there is no need to check the last pair of values. The largest value is guaranteed to already be at the bottom of the array. You can make the sort process even more efficient by using a new variable for the inner `for` loop and reducing the value by one on each cycle through the array.

```
int highSubscript = len - 1;
for(a = 0; a < highSubscript; ++a)
{
   for(b = 0; b < highSubscript; ++b)
      if(array[b] > array[b + 1])
      {
         temp = array[b];
         array[b] = array[b + 1];
         array[b + 1] = temp;
      }
}
```

Figure 8-16 Portion of the sort process



```
A:\Chapter.08>java SortCharArray
Enter a character f
Enter a character d
Enter a character h
Enter a character j
Enter a character a
Enter a character e
Enter a character i
Enter a character c
Enter a character b
Enter a character g
Before Sort
f d h j a e i c b g
After Sort
a b c d e f g h i j

A:\Chapter.08>
```

Figure 8-17 Output of the SortCharArray program

To make the array more efficient:

1. Open the **SortCharArray** program, if necessary, and change the class name to **`SortCharArray2`**. Within the bubbleSort() method in the SortCharArray.java file, position your insertion point after the statement `int highSubscript = len -1;`, and then press **[Enter]** to start a new line.

2. Declare a variable that holds the number of comparisons to make by typing the following:

   `int compsToMake = len - 1;`

3. Replace the inner, b-loop statement, `for(b = 0; b < highSubscript; ++b)`, with **`for(b = 0; b < compsToMake; ++b)`**.

4. Position the insertion point after the closing curly brace for the `if`, and then press **[Enter]** to start a new line.

5. Type the statement that reduces compsToMake by one on each cycle through the array by typing **`--compsToMake;`**. Make sure that this statement is between the closing curly brace of the inner `for` loop and the closing curly brace of the outer `for` loop.
6. Save the program as **SortCharArray2.java**, compile, and run the program. The program executes exactly as before; however, it is more efficient. When you sort an array with 10 or 20 elements, you will not notice any improved efficiency. However, if you need to sort an array with thousands of elements, the program will run much faster if you employ this technique to reduce unnecessary comparisons.

Sorting Arrays of Objects

You can sort arrays of objects in much the same way that you sort arrays of primitive types. The major difference occurs when you make the comparison that determines whether you want to swap two array elements. When you construct an array of the primitive element type, you compare the two array elements to determine whether they are out of order. When array elements are objects, you usually want to sort based on a particular object field.

Assume you have created a simple Employee class similar to the Employee class of Chapter 3, and as shown in Figure 8-18. The class holds two data fields, a constructor, and get and set methods for the fields.

You can write a program that contains an array of Employee objects using the statement `Employee[] someEmps = new Employee[5];`. After you assign employee numbers and salaries to the Employee objects, you want to sort the Employees in empSal order. You can pass the array and its length to a bubbleSort() method that is prepared to receive Employee objects. Figure 8-19 shows the method.

Examine Figure 8-19 carefully and notice that the bubbleSort() method is very similar to the bubbleSort() method you use for an array of any primitive type, but there are three major differences:

- The bubbleSort() method header shows it receives an array of type Employee.
- The temp variable created for swapping is type Employee.
- The comparison for swapping uses the method call getEmpSal() to compare the salary for each Employee object in the array with the salary of the adjacent Employee object.

```
public class Employee
{
   private int empNum;
   private double empSal;
   public Employee(int e, double s)
   {
      empNum = e;
      empSal = s;
   }
   public int getEmpNum()
   {
      return empNum;
   }
   public void setEmpNum(int id)
   {
      empNum = id;
   }
   public double getEmpSal()
   {
      return empSal;
   }
   public void setEmpSal(double r)
   {
      empSal = r;
   }
}
```

Figure 8-18 Employee class

```
public static void bubbleSort(Employee[] array, int len)
{
   int a,b;
   Employee temp;
   int highSubscript = len - 1;
   for(a = 0; a < highSubscript; ++a)
      for(b = 0; b < highSubscript; ++b)
         if(array[b].getEmpSal() > array[b + 1].getEmpSal())
         {
            temp = array[b];
            array[b] = array[b + 1];
            array[b + 1] = temp;
         }
}
```

Figure 8-19 The SortObjects program with a bubbleSort() method for Employee objects

Tip

It is important to note that even though only Employee salaries are compared, you do not swap Employee salaries. You do not want to substitute one Employee's salary for another's. Instead, you swap the entire Employee object so that each Employee object's empNum and empSal are swapped as a unit.

Sorting String Array Elements

When you sort an array of Strings, you must remember that String names are addresses. Therefore, you cannot determine whether two String objects require swapping by comparing their names. Instead, you must use the compareTo() method. Next you will sort a list of Strings.

To sort an array of String objects:

1. Open a new file in your text editor, and then enter the first few lines of a program that will sort String objects:

   ```
   public class SortStrings
   {
    public static void main(String[] args)
    {
   ```

2. Use the following code to declare an array of student names and an integer variable to use as a subscript:

   ```
   String[] students =
    {"Kim", "Ken", "Tom", "Kathy", "Brad"};
   int x;
   ```

3. Write the code that prints the list of Strings, passes the list to a sortStrings() method, and prints the list again:

   ```
   System.out.println("Before sort");
   for(x = 0; x < 5; ++x)
    System.out.println(students[x]);
   sortStrings(students, students.length);
   System.out.println("\nAfter sort");
   for(x = 0; x < 5; ++x)
    System.out.println(students[x]);
   ```

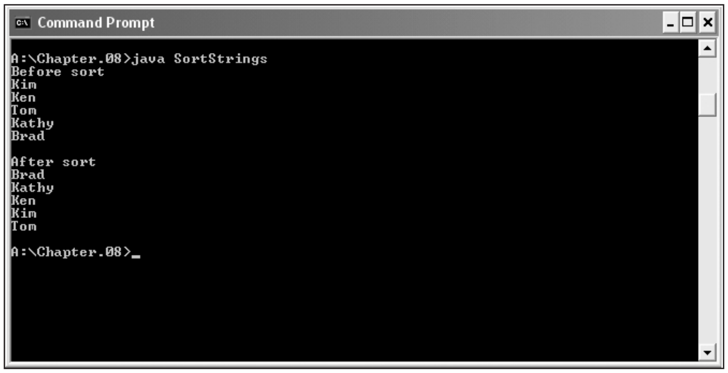
4. Add the closing curly brace for the main() method.

5. Enter the sortStrings() method shown in Figure 8-20. The method uses the compareTo() method to determine whether two Strings should be swapped. Recall that when the compareTo() method returns a value greater than zero, then the first String is larger than (that is, out of order with) the second String.

6. Save the program as **SortStrings.java** in the Chapter.08 folder on your Student Disk and compile. When you run the program, the output is similar to Figure 8-21.

```
public static void sortStrings(String[] array, int len)
{
   int a,b;
   String temp;
   int highSubscript = len - 1;
   for(a = 0; a < highSubscript; ++a)
     for(b = 0; b < highSubscript; ++b)
       if(array[b].compareTo(array[b + 1]) > 0)
       {
          temp = array[b];
          array[b] = array[b + 1];
          array[b + 1] = temp;
       }
}
```

Figure 8-20 The sortStrings() method



```
A:\Chapter.08>java SortStrings
Before sort
Kim
Ken
Tom
Kathy
Brad

After sort
Brad
Kathy
Ken
Kim
Tom

A:\Chapter.08>_
```

Figure 8-21 Output of the SortStrings program

Using Two-Dimensional and Multidimensional Arrays

When you declare an array such as `int[] someNumbers = new int[3];`, you can envision the three declared integers as a column of numbers in memory, as shown in Figure 8-22. In other words, you can picture the three declared numbers stacked one on top of the next. An array that you can picture as a column of values is a one-dimensional or **single-dimensional array**.

Tip

You can think of the single dimension of a single-dimensional array as the height of the array.

SomeNumbers[0]
SomeNumbers[1]
SomeNumbers[2]

Figure 8-22 Single-dimensional array

The Java programming language also supports two-dimensional arrays. **Two-dimensional arrays** have more than one column of values, as shown in Figure 8-23. It is easiest to picture two-dimensional arrays as having both rows and columns. When mathematicians use a two-dimensional array, they often call it a **matrix**; you might have used a two-dimensional array called a spreadsheet.

SomeNumbers[0][0]	SomeNumbers[0][1]	SomeNumbers[0][2]	SomeNumbers[0][3]
SomeNumbers[1][0]	SomeNumbers[1][1]	SomeNumbers[1][2]	SomeNumbers[1][3]
SomeNumbers[2][0]	SomeNumbers[2][1]	SomeNumbers[2][2]	SomeNumbers[2][3]

Figure 8-23 Two-dimensional array

Tip

You can think of the two dimensions of a two-dimensional array as height and width.

When you declare a one-dimensional array, you type a set of square brackets after the array type. To declare a two-dimensional array, you type two sets of brackets after the array type. For example, `int[][] someNumbers = new int[3][4];` declares an array named someNumbers that holds three rows and four columns.

Just as with a one-dimensional array, if you do not provide values for the elements in a two-dimensional numeric array, the values default to zero. You can assign values to the array elements later. For example, `someNumbers[0][0] = 14;` assigns the value 14 to the element of the someNumbers array that is in the first column of the first row. Alternately, you can initialize a two-dimensional array with values when it is created. For example, the following code assigns values to someNumbers when it is created:

```
int[][] someNumbers =
{       {8, 9, 10, 11},
        {1, 3, 12, 15},
        {5, 9, 44, 99}  };
```

The someNumbers array contains three rows and four columns. You contain the entire set of values within a pair of curly braces. The first row of the array holds the four integers 8, 9, 10, and 11. Notice that these four integers are placed within their own set of curly braces to indicate that they constitute one row, or the first row, which is row zero. Similarly, 1, 3, 12, and 15 make up the second row, which you reference with the subscript 1; 5, 9, 44, and 99 are the values in the third row, which you reference with the subscript 2. The value of someNumbers[0][0] is 8. The value of someNumbers[0][1] is 9. The value of someNumbers[2][3] is 99. The value within the first bracket following the array name always refers to the row; the value within the second bracket refers to the column.

Assume you own an apartment building with four floors—a basement, which you refer to as floor zero, and three other floors numbered one, two, and three. Additionally, each of the floors has studio (with no bedroom) and one- and two-bedroom apartments. The monthly rent for each type of apartment is different—the higher the floor, the higher the rent (the view is better), and the rent is higher for apartments with more bedrooms. Table 8-2 shows the rental amounts.

Table 8-2 Rents charged

Floor	Zero Bedrooms	One Bedroom	Two Bedrooms
0	400	450	510
1	500	560	630
2	625	676	740
3	1000	1250	1600

To determine a tenant's rent, you need to know two pieces of information: the floor on which the tenant rents an apartment, and the number of bedrooms in the apartment. Within a Java program, you can declare an array of rents using the following code:

```
int[][] rents =
{       {400, 450, 510},
        {500, 560, 630},
        {625, 676, 740},
        {1000, 1250, 1600}  };
```

Assuming you declare two integers to hold the floor number and bedroom count as `int floor, bedrooms;`, then any tenant's rent is `rents[floor] [bedrooms]`.

To demonstrate the use of a two-dimensional array, you can create a short demonstration program. You will create a teacher's classroom seating chart that holds four rows and three columns. Then you will search for a particular student's location.

To write a program that uses a two-dimensional array to create a student seating chart:

1. Open a new text file in your text editor.
2. Enter the following class header and main() method header for a FindStudent class:

```
public class FindStudent
{
 public static void main(String[] args) throws Exception
 {
```

3. Create a two-dimensional String array that holds the names of 12 students who sit in four rows. For convenience, assign each student a name with a unique initial. That way, you can search for a student's position using an initial.

```
String[][] students =
{    {"Dave", "Bonnie", "Hannah"},
     {"Iris",  "Keith", "Carl"},
     {"Amy", "Jessica", "Francis"},
     {"Ellen", "George", "Lydia"}  };
```

4. You will use a character variable to hold an initial that is input from the keyboard, and two integer variables to hold the row and column position of the student whose initial matches the input initial:

```
char stu;
int r, c;
```

5. Add the following statements to prompt the user for an initial and read the character from the keyboard:

```
System.out.print("Enter student initial ");
stu = Character.toUpperCase((char)System.in.read());
```

6. You will use two nested `for` loops to test each combination of row and column positions. When the character input at the keyboard matches the character in the first position of any of the Strings in the two-dimensional array, print the row and column position. Enter the following `for` loops:

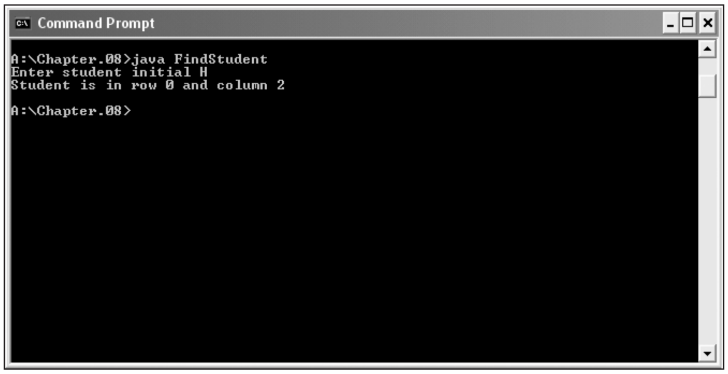
```
for(r = 0; r < 4; ++r)
 for(c = 0; c < 3; ++c)
  if(stu == students[r][c].charAt(0))
   System.out.println("Student is in row " + r +
    " and column " + c);
```

7. Add the closing curly brace for the main() method and the closing curly brace for the class.

8. Save the program as **FindStudent.java** in the Chapter.08 folder on your Student Disk. Compile the program, and then execute it several times. Confirm that with each initial you type, the correct row and column positions are located. Figure 8-24 shows a sample program run.

Understanding Multidimensional Arrays

The Java programming language supports **multidimensional arrays**, or arrays of more than two dimensions. For example, if you own an apartment building with a number of floors and different numbers of bedrooms available in apartments on each floor, you can use a two-dimensional array to store the rental fees. If you own several apartment buildings, you might want to employ a third dimension to store the building number. An expression such as `rents[building][floor][bedrooms]` refers to a specific rent figure for a building whose building number is stored in the building variable, and whose floor and bedroom numbers are stored in the floor and bedrooms variables. Specifically, `rents[5][1][2]` refers to a two-bedroom apartment on the first floor of building 5.



```
A:\Chapter.08>java FindStudent
Enter student initial H
Student is in row 0 and column 2

A:\Chapter.08>
```

Figure 8-24 Output of the FindStudent program

When you are programming in Java, you can use four, five, or more dimensions in an array. As long as you can keep track of the order of the variables needed as subscripts, and as long as you don't exhaust your computer's memory, Java will let you create arrays of any size.

Chapter Summary

- An array is a named list of data items that all have the same type. You declare an array variable by inserting a pair of square brackets after the type. Declaring an array and actually reserving memory space for it are two distinct processes. You use the keyword `new` to reserve memory locations for the array elements.
- A subscript is an integer contained within square brackets that indicates one of an array's variables, or elements. An array's elements are numbered beginning with zero. When you work with any individual array element, you treat it no differently than you would a single variable. Array names actually represent computer memory addresses—they are references, as are all Java objects.
- Each element of a numeric array is automatically set to zero, character array elements are assigned '\u0000', and Boolean array elements are automatically assigned the value `false`.
- To initialize an array, you can use a list of values that are separated by commas and enclosed within curly braces. When you initialize an array by giving it values upon creation, you do not give the array a size; the size will be assigned based on the number of values you place in the initializing list.
- Just as you can declare arrays of integers or doubles, you can declare arrays of any type, including objects. You must explicitly call individual constructors for array objects.

- You can pass a single array element to a method in exactly the same manner you would pass a variable. The array element is passed by value; that is, a copy is made. You can pass an array to a method. Arrays, like all objects, are passed by reference; the method has access to the actual values in the array elements.
- You can use an array's length field to determine the number of elements.
- Sorting is the process of arranging a series of objects in some logical order. When you place objects in order beginning with the object with the lowest value, you are sorting in ascending order; when you start with the object that has the largest value, you are sorting in descending order.
- To sort any two values in ascending order, you use an `if` statement to make the decision whether to swap the positions of the two values. You must use the compareTo() method when sorting Strings.
- You usually want to sort based on a particular field contained in each object. To use a bubble sort, you place the original, unsorted values in an array, compare pairs of values, and then, if they are not in ascending order, swap them. To use a bubble sort, you always, at most, need to pass through a list as many times as its length minus one.
- An array that you can picture as a column of values is a one-dimensional or single-dimensional array. The Java programming language supports multidimensional arrays, or arrays of more than two dimensions.

Review Questions

1. An array is a list of data items that ________________.
   a. all have the same type
   b. all have different names
   c. all are integers
   d. all are `null`
2. Declaring an array and reserving memory for an array ________________.
   a. are always done in the same statement
   b. are two distinct processes
   c. depends on the type of array
   d. must not use the keyword `new`
3. You reserve memory locations for an array when you ________________.
   a. declare the array name
   b. use the keyword `new`
   c. use the keyword `mem`
   d. explicitly store values within the array elements

4. The statement `int[] value = new int[34];` reserves memory for ________ integers.
   a. 0
   b. 33
   c. 34
   d. 35
5. A(n) ________ contained within square brackets is used to indicate one of an array's elements.
   a. character
   b. double
   c. integer
   d. string
6. If you declare an array as `int[] num = new int[6];`, the last element of the array is ________.
   a. num[0]
   b. num[1]
   c. num[5]
   d. impossible to tell
7. If you declare an integer array as `int[] num = {101,202,303,404,505, 606};`, then the array element num[2] contains the number ________.
   a. 101
   b. 202
   c. 303
   d. impossible to tell
8. Array names are ________.
   a. values
   b. functions
   c. references
   d. allusions
9. Unicode value '\u0000' also is known as ________.
   a. nill
   b. void
   c. nada
   d. null

10. When you initialize an array by giving it values upon creation, you ________________.
    a. do not give the array a size
    b. also must give the array a size
    c. must make all the values zero, blank, or `false`
    d. must make sure each value is different from the others
11. Assume an array is declared as `int[] num = new int[4];`. Which of the following statements correctly assigns the value 100 to each of the four array elements?
    a. `for(x = 0; x < 3; ++x) num[x] = 100;`
    b. `for(x = 0; x < 4; ++x) num[x] = 100;`
    c. `for(x = 1; x < 4; ++x) num[x] = 100;`
    d. `for(x = 1; x < 5; ++x) num[x] = 100;`
12. If a class named Student contains a method setID() that takes an integer argument, and you create an array of 20 Student objects named scholar, which of the following statements correctly assigns an ID number to the first Student scholar?
    a. `Student[0].setID(1234);`
    b. `scholar[0].setID(1234);`
    c. `Student.setID[0](1234);`
    d. `scholar.setID[0](1234);`
13. Searching through an array for an exact match to some variable is a good idea when ________________.
    a. no variable values are invalid
    b. the variable values to be matched are sequential
    c. there are relatively few possibilities for the value in the variable
    d. you need to match a variable based on a range of values
14. When you pass an array element to a method, the method receives ________________.
    a. a copy of the array
    b. the address of the array
    c. a copy of the value in the element
    d. the address of the element
15. When you pass an array to a method, the method receives ________________.
    a. a copy of the array
    b. a copy of the first element in the array
    c. the address of the array
    d. nothing

16. When you place objects in order beginning with the object with the highest value, you are sorting in ________ order.
    a. acquiescing
    b. ascending
    c. demeaning
    d. descending
17. Using a bubble sort involves ________.
    a. comparing parallel arrays
    b. comparing each array element to the average
    c. comparing each array element to the adjacent array element
    d. swapping every array element with its adjacent element
18. Which array types cannot be sorted?
    a. arrays of characters
    b. arrays of Strings
    c. arrays of objects
    d. You can sort all of the above array types.
19. When array elements are objects, you usually want to sort based on a particular ________ of the object.
    a. field
    b. method
    c. name
    d. type
20. The array `int[][]nums={{1,2},{3,4},{5,6}};` is a ________ array.
    a. one-dimensional
    b. two-dimensional
    c. multidimensional
    d. nondimensional

EXERCISES

1. Write a program that can hold five integers in an array. Display the integers from first to last, and then display the integers from last to first. Save the program as **IntArray.java** in the Chapter.08 folder on your Student Disk.
2. Write a program using dialog boxes that prompts the user to make a choice for a pizza size—S, M, L, or X—and then displays the price as $6.99, $8.99, $12.50, or $15.00 accordingly. Save the program as **PizzaChoice.java** in the Chapter.08 folder on your Student Disk.

3. a. Create a class named Taxpayer. Data fields for Taxpayer include Social Security number (use an int for the type, and do not use dashes within the Social Security number) and yearly gross income. Methods include a constructor that requires values for both data fields, and two get methods that return each of the data field values. Write a program named UseTaxpayer that declares an array of 10 Taxpayer objects. Set each Social Security number to 999999999 and each gross income to zero. Display the 10 Taxpayer objects. Save the programs as **Taxpayer.java** and **UseTaxpayer.java** in the Chapter.08 folder on your Student Disk.
   b. Modify your program so each Taxpayer has a successive Social Security number from 1 through 10, and gross incomes that range from $10,000 to $100,000, increasing by $10,000 for each successive Taxpayer. Save the program as **UseTaxpayer2.java** in the Chapter.08 folder on your Student Disk.
4. Create an array that stores 20 prices, such as $2.34, $7.89, $1.34, and so on. Display the sum of all the prices. Display all values less than $5.00. Calculate the average of the prices, and display all values that are higher than the calculated average value. Save the program as **Prices.java** in the Chapter.08 folder on your Student Disk.
5. a. Write a program that prompts a professor to input grades for five different courses for 10 students. Prompt the professor to enter one grade at a time using the prompt "Enter name for student #1" and "Enter grade #1." Verify that the professor enters only A, B, C, D, or F. Use variables for the student numbers (1 through 10) and grade numbers (1 through 5). Save the programs as **Student.java** and **GradePoint.java** in the Chapter.08 folder on your Student Disk.
   b. Modify the GradePoint program so that it calculates the grade point average (GPA) for each student. A student receives four grade points for an A, three grade points for a B, two grade points for a C, one grade point for a D, and zero grade points for an F. Store the grades and points in parallel arrays. Search the arrays to determine the points for the grade. Store the GPA for each student in another array. (*Hint*: Copy the GPA for each student to a different array by initializing the new array with GPAs from the other array.)
   c. Display the GPA scores from each of the two GPA arrays to verify that the GPAs were copied correctly. Identify which array the scores are from. Save the final program as **GradePoint.java** in the Chapter.08 folder on your Student Disk.
6. a. Write a program that displays a multiple choice quiz of 10 questions with topics related to your favorite hobby. Each question has one correct answer and three possible answers. Verify that the user enters only A, B, or C as the answer. Store the correct answers in an array. Store the user's answers in a second array. If the user responds to a question correctly, display "Correct!"; otherwise display "The correct answer is" and the letter of the correct answer. Determine what to display by comparing the response to the array of correct answers. Save the program as **Quiz.java** in the Chapter.08 folder on your Student Disk.
   b. Modify your Quiz class so that it displays the number of correct answers after the user answers 10 questions. Determine the score by comparing the two arrays.

7. a. Write a program that lets the user enter numbers (1 through 9) one at a time, and then prints the numbers that the user entered. Allow the user to enter up to 10 numbers. If the user tries to enter an 11th number, display a message that no more numbers can be entered. Store the numbers in an array. Verify that invalid characters are not entered. Save the program as **EnterNumbers.java** in the Chapter.08 folder on your Student Disk.

   b. Change the EnterNumbers program to EnterNumbers2, so that it lets the user delete or modify a number. Include a menu that shows the options to enter, remove, modify, or display a number, or quit the program. Verify that a correct option is entered. When the user chooses the remove option, prompt the user to specify which number to remove. Verify that the user enters a valid number (1 through 9), and then change that number in the array to 0.

   c. Change the program so that when the user chooses the modify option, the program prompts the user to specify which number to modify. Verify that the user enters a valid number (1 through 9), ask for the new number, verify that the user enters a valid number, and then change the number in the array. Save the final program as **EnterNumbers2.java** in the Chapter.08 folder on your Student Disk.

8. Write a program that stores vowels (a, e, i, o, and u) in an array. Ask the user to enter a character. Then the program should indicate whether the entered character is a vowel. Save the program as **VowelArray.java** in the Chapter.08 folder on your Student Disk.

9. Store 40 characters in an array, such as `1234%$#@UHGF....` Write a program that produces a count of how many of the characters are letters in the English alphabet, and how many of the characters are not letters. Save the program as **EnglishArray.java** in the Chapter.08 folder on your Student Disk.

10. Write a program that prompts the user for a first name. Print a greeting to the person using the name, such as "Hello Kimberly!" Save the program as **HelloArray.java** in the Chapter.08 folder on your Student Disk.

11. Store 20 integer employee ID numbers in an integer array, and 20 corresponding employee last names in a String array. Use dialog boxes to accept an ID number, and display the appropriate last name. Save the program as **EmployeeIdArray.java** in the Chapter.08 folder on your Student Disk.

12. Create an array of Strings containing the days of the week ("Sunday" through "Saturday"). Review the use of the GregorianCalendar class in Chapter 4. The GregorianCalendar class contains a method get(Object.DAY_OF_WEEK) that returns an integer value one through seven that represents Sunday through Saturday. Write a program in which you create a GregorianCalendar object, assign it a value, and then print a day that corresponds to the GregorianCalendar. Save the program as **DayArray.java** in the Chapter.08 folder on your Student Disk.

13. Create an array of Strings, each containing one of the top 10 reasons that you like Java. Prompt a user to enter a number, convert the number to an integer, and then use the integer to print one of the reasons for the user. Save the program as **JavaArray.java** in the Chapter.08 folder on your Student Disk.

14. Create an array of five Strings containing the first names of people in your family. Write a program that counts and displays the number of vowels in the Strings that you entered, without regard to case (uppercase versus lowercase letters). Save the program as **Vowels.java** in the Chapter.08 folder on your Student Disk.
15. Write a program that contains three parallel arrays. The first array holds student ID numbers, the second holds first names, and the third holds the students' grade point averages. Use dialog boxes to accept a student ID number with nine digits, and then display the student's first name and grade point average. If a match is not found display "No Match... ". Save the program as **StudentIDArray.java** in the Chapter.08 folder on your Student Disk.
16. A phone directory contains ID numbers, names, and phone numbers for 10 people. Write a program that uses dialog boxes to search for a phone number for a person based on the ID number. If the ID number is found, the persons name, ID number, and phone number are displayed and the program ends. If the ID is not found the user is prompted to enter a name, ID, and phone number, and the new information is displayed using a single dialog message box. Save the program as **PhoneNumberArray.java** in the Chapter.08 folder on your Student Disk.
17. a. Write a program containing an array of 15 double values. Include a method to sort the values in ascending order. Compile, run, and check the results.

    b. Change the sort to sort in descending order. Save the program as **SortDouble.java** in the Chapter.08 folder on your Student Disk.
18. Write an Employee program containing methods for setting and getting Employee numbers and Employee salaries. Write a program that instantiates five Employee objects, sorts the Employee objects, and prints the Employee objects in descending order by Employee number. Save the program as **EmployeeSort.java** in the Chapter.08 folder on your Student Disk.
19. Write a program that allows the user to enter a course ID number and then displays the course name (such as "CIS 110") and the day of the week and time that the course is held (such as "Th 3:30"). Store the course name and day/time in a two-dimensional array. Save the program as **Schedule.java** in the Chapter.08 folder on your Student Disk.
20. Write a program that stores an array of video titles (such as "True Grit") and their corresponding ID numbers in inventory (such as "145"). Display the list before it is sorted, and then display a list sorted by inventory ID number. Use two single-dimensional arrays—one for the titles and one for the inventory ID numbers. Save the program as **Video.java** in the Chapter.08 folder on your Student Disk.
21. Write a program that stores the name, title, and hourly wage of people employed by a grocery store. The data are: Ollie Regan, manager, $18/hour; William Sherman, assistant manager, $16/hour; Maureen Mooney, produce manager, $15/hour; Marty Sharik, bakery manager, $15.25/hour; and Marcella Riley, cashier manager, $13/hour. List the employee name and job title for employees who earn more than $15 per hour. Store the names and titles for each employee in a two-dimensional array, and store the rate in a single-dimensional array. Save the program as **Rate.java** in the Chapter.08 folder on your Student Disk.

22. Each of the following files in the Chapter.08 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugEight1.java will become FixDebugEight1.java.
    a. DebugEight1.java
    b. DebugEight2.java
    c. DebugEight3.java
    d. DebugEight4.java

CASE PROJECT

The We Cut Salon offers a variety of salon services for its customers. Jane Fields, the owner, has contracted to have you write a program that will allow reports to be output sorted by each type of service offered. Create a class for services offered by a hair styling salon. Data fields include a String to hold the service description (for example, "Cut", "Shampoo", or "Manicure"), a double to hold the price, and an integer to hold the average minutes it takes to perform the service. The class name is **HairSalon**. Include a constructor that requires arguments for all three data fields and three get methods that each return one of the data field's values.

Write a program named **SortSalon** that contains an array to hold six HairSalon objects and fill it with data. Include a method to sort the array in ascending order by price of service. Call the method and display the results. Add a second method to the SortSalon program that sorts the HairSalon objects in descending order by the time it takes to perform the service. Call the method and display the results. Add a third method to the SortSalon program that sorts the HairSalon objects in alphabetical order by service description. Call the method and display the results. Finally, add a prompt to the SortSalon program giving the user three choices: sort by description, price, or time. Depending on the user's input, call one of the three sort methods and display the results.

Table 8-3 shows the various services, service prices, and service times:

Table 8-3 Salon services, prices, and times

Service	Price ($)	Time (minutes)
Cut	8.00	15
Shampoo	4.00	10
Manicure	18.00	30
Style	48.00	55
Permanent	18.00	35
Trim	6.00	5

CHAPTER

9 APPLETS

In this chapter, you will:

- Write an HTML document to host an applet
- Understand simple applets
- Use Labels with simple AWT applets
- Write a simple Swing applet and use a JLabel
- Add JTextField and JButton Components to Swing applets
- Learn about event-driven programming
- Add output to a Swing applet
- Understand the Swing applet life cycle
- Create a more-sophisticated interactive Swing applet
- Use the setLocation() and setEnabled() methods

It seems like I've learned a lot," you tell Lynn Greenbrier during a coffee break at Event Handlers Incorporated. "I can use variables, make decisions, write loops, and use arrays."

"You've come a long way," Lynn agrees.

"But at the same time," you continue, "I feel like I know nothing! When I visit the simplest Web site, it looks far more sophisticated than my most advanced application. There is color and movement. There are buttons to click and boxes into which I can type responses to questions. Nothing I've done even approaches that."

"But you have a good foundation in Java programming," Lynn says. "Now you can put all that knowledge to work. By adding a few new objects to your repertoire, and by learning a little about applets, you can comfortably enter the world of interactive Web programming."

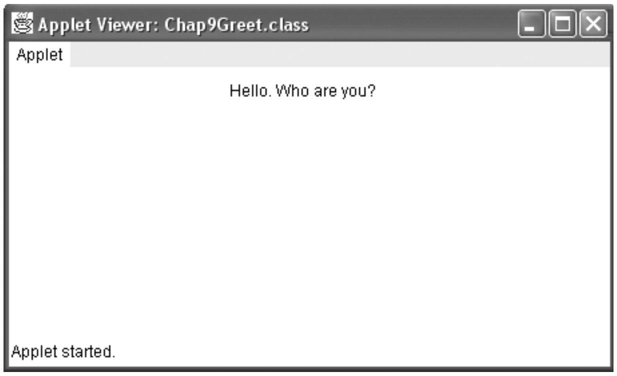
PREVIEWING THE AWT AND SWING GREET APPLETS

The Chap9Greet and Chap9JGreet classes preview two basic applets. Chap9Greet is an applet created using the Abstract Windows Toolkit (AWT) and the Applet class. Chap9JGreet is also an applet created using the AWT, but is different because Chap9JGreet uses the JApplet class.

You can now use a completed version of each applet, which you will find saved in the Chapter.09 folder on your Student Disk. Also saved in the Chapter.09 folder are copies of Chap9Greet and Chap9JGreet.

To run the Chap9Greet and Chap9JGreet applets:

1. At the command prompt for the Chapter.09 folder on your Student Disk, type **`appletviewer TestChap9Greet.html`**, and then press **[Enter]**. It might take a few minutes for the Applet Viewer window to open. See Figure 9-1.

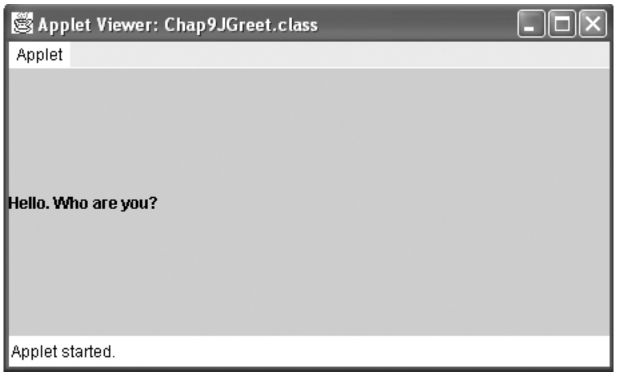


```
Hello. Who are you?
```

```
Applet started.
```

Figure 9-1 Chap9Greet applet

2. Click the **Close** button in the upper-right corner of the Applet Viewer window to close the Applet Viewer.
3. At the command prompt for the Chapter.09 folder on your Student Disk, type **`appletviewer TestChap9JGreet.html`**, and then press **[Enter]**. View the applet shown in Figure 9-2.
4. Close the Applet Viewer.



Hello. Who are you?

Applet started.

Figure 9-2 Chap9JGreet applet

Writing an HTML Document to Host an Applet

You have written many Java applications. When you write a Java application, you do the following:

- Write the application in the Java programming language, and then save it with a .java file extension.
- Compile the application into bytecode using the `javac` command. The bytecode is stored in a file with a .class file extension.
- Use the `java` command to interpret and execute the .class file.

As you know, applications are stand-alone programs. In contrast, **applets** are programs that are called from within another application. You run applets within a page on the Internet, an intranet, or a local computer from within another program called **Applet Viewer**, which comes with the Java Developer's Kit. To view an applet, it must be called from within another document written in HTML. **HTML**, **Hypertext Markup Language**, is a simple language used to create Web pages for the Internet. HTML contains many commands that allow you to format text on a Web page, import graphic images, and link your page to other Web pages. When you create an applet, you do the following:

- Write the applet in the Java programming language, and save it with a .java file extension, just as when you write a Java application.
- Compile the applet into bytecode using the `javac` command, just as when you write a Java application.
- Write an HTML document that includes a statement to call your compiled Java class.

- Load the HTML document into a Web browser (such as Netscape Navigator or Microsoft Internet Explorer), or run the Applet Viewer program, which, in turn, uses the HTML document.

Java, in general, and applets, in particular, are popular topics among programmers, mostly because users can execute applets using a Web browser on the Internet. A **Web browser** is a program that allows you to display HTML documents on your computer screen. Web documents often contain Java applets.

Fortunately, to run a Java applet, you don't need to learn the entire HTML language; you need to learn only two pairs of HTML commands, called **tags**. The tag that begins every HTML document is **`<HTML>`**. Like all tags, this tag is surrounded by angle brackets. `HTML` is an HTML keyword which specifies that an HTML document follows the keyword. The tag that ends every HTML document is **`</HTML>`**. Placing a backslash before any tag indicates the tag is the ending half of a pair of tags. The following is the simplest HTML document you can write:

```
<HTML>
</HTML>
```

Tip

Unlike the Java programming language, HTML is not case sensitive so you can use `<html>` in place of `<HTML>`. However, this book uses the all uppercase convention when typing HTML code. With the growing importance of XML and XHTML, many programmers recommend putting all tags in lowercase since XML and XHTML are case sensitive.

The simple HTML document begins and ends and does nothing in the process; you can create an analogous situation in a Java method by typing an opening curly brace and following it immediately with the closing curly brace. HTML documents generally contain more statements. For example, to run an applet from within an HTML document, you add an `<APPLET>` and `</APPLET>` tag pair. Usually, you place three attributes within the `<APPLET>` tag: `CODE`, `WIDTH`, and `HEIGHT`. **Attributes**, sometimes referred to as arguments, promote activity—with them the HTML tag can do something in a certain way. Note the following example:

```
<APPLET CODE = "AClass.class" WIDTH = 300 HEIGHT = 200
</APPLET>
```

The following are three `APPLET` tag attributes and a description of their corresponding arguments:

- `CODE =` is followed by the name of the compiled applet you are calling
- `WIDTH =` is followed by the width of the applet on the screen
- `HEIGHT =` is followed by the height of the applet on the screen

The name of the applet you call must be a compiled Java applet (with a .class file extension). The width and height of an applet are measured in pixels. **Pixels** are the picture elements, or tiny dots of light that make up the image on your video monitor. For monitors that display 800 pixels horizontally and 600 pixels vertically, a statement such as `WIDTH = 400 HEIGHT = 300` will create an applet that occupies approximately one-fourth of most screens (half the height and half the width).

Tip

SVGA monitors commonly display 800 x 600 pixels, 1024 x 768 pixels, or higher. If you want most users to see larger applets without scrolling, the maximum size of your applets should be 760 x 520 pixels. The standard 600 x 400 pixels was used to support the older 640 x 480 VGA monitors and is currently considered out-of-date. Keep in mind that the browser's menu bar and screen elements (such as the toolbar and the scrollbars) will take up some of the screen viewing area for an applet.

Next you will create a simple HTML document that you will use to display the applet that you create in the next section. You will name the applet Greet, and it will occupy a screen area of 450 × 200 pixels.

To create a simple HTML document:

1. Open a new file in your text editor.
2. Type the opening HTML tag, **`<HTML>`**.
3. On the next line, type the opening `APPLET` tag that contains the applet's name and dimensions: **`<APPLET CODE = "Greet.class" WIDTH = 450 HEIGHT = 200>`**.
4. On the next line, type the applet's closing tag: **`</APPLET>`**.
5. On the next line, type the closing HTML tag: **`</HTML>`**.
6. Save the file as **TestGreet.html** in the Chapter.09 folder on your Student Disk. Just as when you create a Java application, make sure that you save the file as text only using an .html extension. The .html file extension is required and makes the file easy to identify as an HTML file. If you are using Notepad or another text editor, you can enclose the filename in quotation marks to save the .html file extension, as in "A:\Chapter.09\TestGreet.html".

UNDERSTANDING SIMPLE APPLETS

To write an applet you must learn only a few additions and changes to writing a Java application. In addition to what you learned about creating applets in the beginning of this chapter, you must also do the following to write an applet:

- Include import statements to ensure that necessary classes are available.
- Learn to use some Windows components and applet methods.
- Learn to use the keyword `extends`.

In Chapter 3, you used an import statement to access classes such as java.util.Date and java.util.GregorianCalendar within your application. You imported these classes to avoid having to write common date-handling routines that already exist in Java. Similarly, Java's creators fashioned a variety of classes to handle common applet needs. Programmers use core classes of the Java platform that include the java.applet.Applet and javax.swing.JApplet. The structure for these core classes is shown in Figure 9-3.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
                          |
                          +--javax.swing.JApplet
```

Figure 9-3 Structure of Applet and JApplet classes

The class structure in Figure 9-3 shows that the `Applet` class is used to create an AWT applet and the `JApplet` class is used to create a Swing applet. You can use methods from the Component and Container classes for both types. A **Component** is a class that defines any object that you want to display. In the pages that follow, you will display frames, panels, buttons, labels, and text fields in the applets you create. A **Container** is a class that is used to define a component that can contain other components.

Most AWT applets contain at least two import statements: `import java.applet.*;` and `import java.awt.*;`. The java.applet package contains a class named Applet—every applet you create is based on this class. The java.awt package is the Abstract Windows Toolkit, or AWT. (In Java 1.1, the java.awt package contained the primary classes you would use to create a Graphical User Interface, or GUI.)

Most Swing applets contain at least two import statements: `import javax.swing.*;` and `import java.awt.*;`. Many of the AWT classes have been superceded in Java 2 by Swing classes. The classes in the javax.swing package define GUI elements, referred to as **Swing components**, and provide much-improved alternatives to components defined by classes in java.awt. The Swing classes are part of a more general set of GUI programming capabilities that are collectively referred to as the **Java Foundation Classes**, or **JFC**. JFC includes Swing component classes and selected classes from the java.awt package.

When you create an application, you follow any needed import statements with a class header such as `public class AClass`. AWT applets and Swing applets begin the same way as Java applications, but they must also include the words `extends Applet` and `extends JApplet`. The keyword `extends` indicates that your applet will build on, or inherit, the traits of the Applet or JApplet classes.

Both the Applet and JApplet classes provide a general outline used by any Web browser when it runs an applet. In an application, the main() method calls other methods that you write. With an applet, the browser calls many methods automatically. The following four methods are included in every applet:

- `public void init()`
- `public void start()`
- `public void stop()`
- `public void destroy()`

If you fail to write one or more of these methods, Java creates them for you. The methods Java creates have opening and closing curly braces only—in other words, they are empty. To create a Java program that does anything useful, you must code at least one of these methods.

The init() method is the first method called in any applet. You use it to perform initialization tasks, such as setting variables to initial values or placing applet components on the screen. You must code the init() method's header as `public void init()`.

Tip

In the text which follows, you will need to distinguish between applets created by the Applet and JApplet classes. The term Applet is used for the AWT applet; JApplet is the term used for the Swing applet. When the term applet is used alone, it refers to material that can apply to both applet types.

Using Labels with Simple AWT Applets

The java.awt package contains commonly used Windows components such as Labels, Menus, and Buttons. You import java.awt so you don't have to "reinvent the wheel" by creating these components yourself. AWT Applets are not required to contain Windows components, but they almost always do.

One of the simplest Window components is a Label. **Label** is a built-in class that holds text that you can display within an applet. The Label class also contains fields that indicate appearance information, such as font and alignment. As with other objects, you can declare a Label without allocating memory, as in `Label greeting;`, or you can call the Label constructor without any arguments, as in `Label greeting = new Label();`. You can assign some text to the Label with the **setText() method**, as in `greeting.setText("Hi there");`. Alternately, you can call the Label constructor and pass it a String argument so the Label is initialized upon construction, as in `Label greeting = new Label("Hello. Who are you?");`.

You use the **add() method** to add a component to an applet window. For example, if a Label is defined as `Label greeting = new Label("Hello. Who are you?");`, then you can place a greeting within an applet using the command `add(greeting);`.

Tip: The object of the add() method is the applet itself, so when you add a component to a window, you could write `this.add();` in place of `add();`. You learned about the `this` reference in Chapter 4.

Figure 9-4 shows the program to create an applet that displays "Hello. Who are you?" on the screen.

```
import java.applet.*;
import java.awt.*;

public class Greet extends Applet
{
   Label greeting = new Label("Hello. Who are you?");
     public void init()
     {
        add(greeting);
     }
}
```

Figure 9-4 AWT Greet applet

Next you will create and compile the Greet applet.

To create and run the Greet applet:

1. Open a new text file in your text editor.
2. Enter the code shown in Figure 9-4.
3. Save the file as **Greet.java** in the Chapter.09 folder on your Student Disk.
4. Compile the program with the command **`javac Greet.java`**.
5. If necessary, correct any errors, and then compile again.

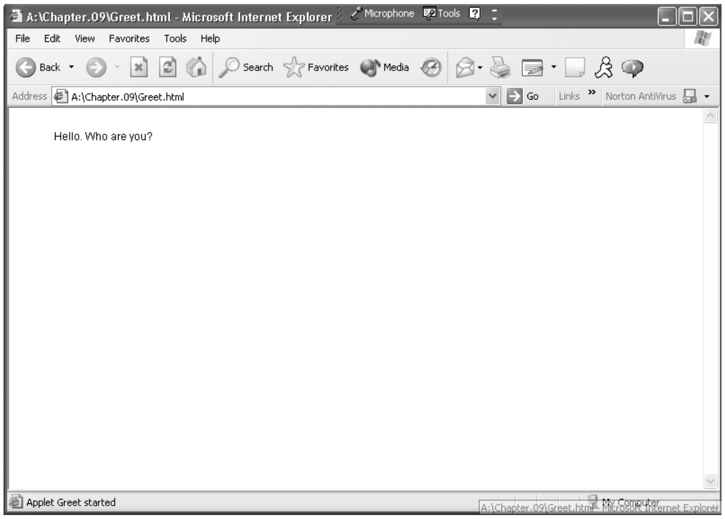
To run the Greet applet, you can use your Web browser or the `appletviewer` command. In the following steps, you will do both.

To run the applet using your Web browser:

1. Open any Web browser, such as Microsoft Internet Explorer or Netscape. You do not have to connect to the Internet; you will use the browser locally.

Tip: If you do not have a Web browser installed on your computer, skip to the end of Step 3.

2. Click **File** on the menu bar, click **Open** or **Open Page**, type **A:\Chapter.09\TestGreet.html**, which is the complete path for the HTML document that you created to access Greet.class, and then press **[Enter]**. The applet should appear on your screen, as shown in Figure 9-5. If you receive an error message, verify that the path and spelling of the HTML file are correct.


A:\Chapter.09\Greet.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address A:\Chapter.09\Greet.html
Go Links Norton AntiVirus
My Computer

Hello. Who are you?

Applet Greet started

Figure 9-5 TestGreet.html displayed in Internet Explorer

3. Click the **Close button** in the upper right-hand corner of the browser's program window to close your Web browser.

You can also view your applet using the `appletviewer` command. In this book, you will test your applets using this command.

Some applets may not work correctly using your browser. Java was designed with a number of security features so that when an applet displays on the Internet, the applet cannot perform malicious tasks, such as deleting a file from your hard drive. If an applet does nothing to compromise security, then testing it using the Web browser or the `appletviewer` command achieves the same results. For now, you can get your applets to perform better by using the Applet Viewer window because the output will not depend on browser type or version.

To run the applet using the `appletviewer` command:

1. At the command line, type **`appletviewer TestGreet.html`**, and then press **[Enter]**. After a few moments, the Applet Viewer window opens and displays the applet, as shown in Figure 9-6.

2. Use the mouse pointer to drag any corner of the Applet Viewer window to resize it. Notice that if you widen the window by dragging its right border to the right, the window is redrawn on the screen and the Label is automatically repositioned to remain centered within the window. If you narrow the window by dragging its left border to the left, the Label eventually is partially obscured when the window becomes too narrow for the display.

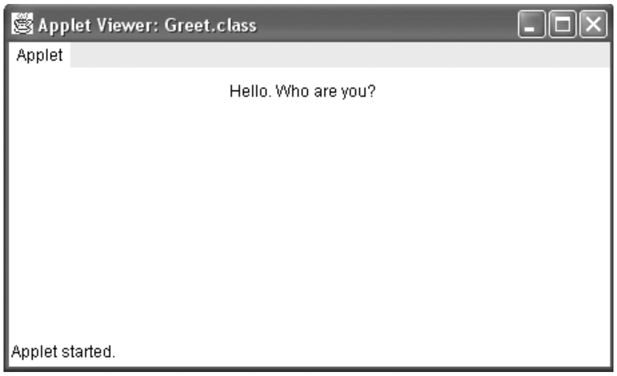


Figure 9-6 Output of TestGreet.html page in an Applet Viewer window

3. Close the Applet Viewer.

Writing a Simple Swing Applet and Using a JLabel

The Swing component classes offer more flexibility than classes defined in the java.awt package because they are implemented entirely in Java. Older java.awt components greatly relied on the native code of the operating system on which Java was then implemented. This chapter makes primary use of the GUI built on the Java Foundation classes. You will import java.awt and javax.swing classes so you don't have to create these components yourself.

The counterpart to the AWT Label is a JLabel. **JLabel** is a built-in class that holds text that you can display within an applet. The structure of the JLabel class is shown in Figure 9-7.

Available constructors for the JLabel class include:

- `JLabel()` creates a JLabel instance with no image and with an empty string for the title.
- `JLabel(Icon image)` creates a JLabel instance with the specified image.
- `JLabel(Icon image, int horizontalAlignment)` creates a JLabel instance with the specified image and horizontal alignment.
- `JLabel(String text)` creates a JLabel instance with the specified text.

- `JLabel(String text, Icon icon, inthorizontalAlignment)` creates a JLabel instance with the specified text, image, and horizontal alignment.
- `JLabel(String text, int horizontalAlignment)` creates a JLabel instance with the specified text and horizontal alignment.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JLabel
```

Figure 9-7 Structure of the JLabel class

Tip

Using the JLabel class, you allocate memory, create objects, and assign text using class constructors and methods that are virtually identical to those found in the Label class. The primary difference is that more constructors and methods are available in the JLabel class.

A major difference between AWT and Swing classes is the method used to add a component to a Swing window. AWT components are added directly to the Applet; Swing components must use a content pane. GUI components, such as JLabels, are attached to a separate content pane so they can be displayed at execution time. This is true for a Swing application as well as a Swing applet.

Referring back to Figure 9-3, the **content pane** is an object of the Container class from the java.awt package. A Container object can be created using the **getContentPane() method**. To create a Container object named `con`, the syntax is `Container con = getContentPane();`. Then the statement `JLabel greeting = new JLabel();` adds the greeting object to the content pane with the statement `con.add(greeting);`.

Figure 9-8 shows the program to create a Swing applet that displays "Hello. Who are you?" on the screen. Next you will create a Swing applet.

To create and run the JGreet applet:

1. Open a new text file in your text editor.
2. Enter the code shown in Figure 9-8.
3. Save the file as **JGreet.java** in the Chapter.09 folder on your Student Disk.
4. Compile the program and, if necessary, correct any errors, and then compile again.

5. Open a new file in your text editor, and then type the HTML document that will run the JApplet as follows:

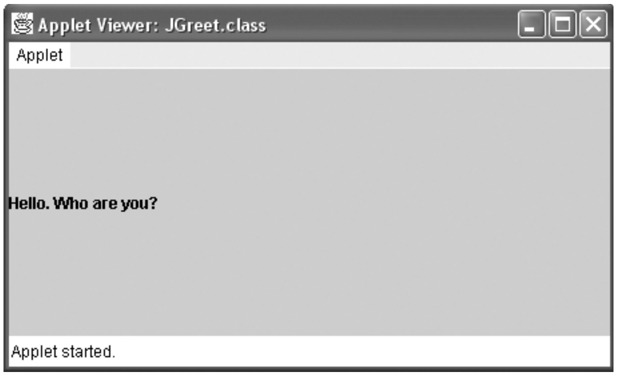
   ```
   <HTML>
   <APPLET CODE = "Greet.class" WIDTH = 450 HEIGHT = 200>
   </APPLET>
   </HTML>
   ```

   Save the file as **TestJGreet.html** in the Chapter.09 folder of your Student Disk.

6. At the command line, type **`appletviewer TestJGreet.html`**, and then press **[Enter]**. The applet appears on your screen, as shown in Figure 9-9.

```
import javax.swing.*;
import java.awt.*;
public class JGreet extends JApplet
{
   JLabel greeting = new JLabel("Hello. Who are you?");

   public void init()
   {
     Container con = getContentPane();
     con.add(greeting);
   }
}
```

Figure 9-8 Swing Greet applet



```
Hello. Who are you?
```

```
Applet started.
```

Figure 9-9 Output of TestJGreet.html page in Applet Viewer window

7. Close the Applet Viewer window.

Changing a JLabel's Font

If you use the Internet and a Web browser to visit Web sites, you probably are not very impressed with either of your simple java applets. You might think that the string, "Hello. Who are you?" is pretty plain and lackluster. Fortunately, Java provides you with a **Font object** that holds typeface and size information. The **setFont() method** requires a Font object argument. To construct a Font object, you need three arguments: typeface, style, and point size.

The **typeface** is a String representing a font. Common fonts are Arial, Helvetica, Courier, and Times New Roman. The typeface is only a request; the system on which your applet runs might not have access to the requested font and substitute a default font. The **style** applies an attribute to displayed text and is one of three arguments, Font.PLAIN, Font.BOLD, or Font.ITALIC. The **point size** is an integer that represents 1⁄72 of an inch. Printed text is usually about 12 points; a headline might be 30 points.

To give a Label object a new font, you create the Font object, as in `Font headlineFont = new Font("Helvetica", Font.BOLD, 36);`, and then you use the setFont() method to assign the font to a Label with the statement `greeting.setFont(headlineFont);`.

The typeface name is a String, so you must enclose it in double quotation marks when you use it to declare the Font object.

Next you will change the font of the text in your JGreet applet.

To change the appearance of the greeting in the JGreet applet:

1. Open the **JGreet.java** file in your text editor and change the class name to **JGreet2**.
2. Position the insertion point at the end of the line that declares the greeting Label, and then press **[Enter]** to start a new line of text.
3. Declare a Font object named bigFont by typing the following:

   ```
   Font bigFont = new Font("TimesRoman", Font.ITALIC, 24);
   ```
4. Place the insertion point to the right of the opening curly brace of the init() method, and then press **[Enter]** to start a new line.
5. Set the greeting font to bigFont by typing **`greeting.setFont(bigFont);`**.
6. Save the file using the filename **JGreet2.java**.
7. At the command line, compile the program, and if necessary, correct any errors.
8. Run the applet, changing the TestJGreet.html document created earlier to **TestJGreet2.html**. Within the TestJGreet2.html document, change the class internally to **`JGreet2.class`** and then execute the **`appletviewer TestJGreet2.html`** command. Figure 9-10 shows the output.
9. Close the Applet Viewer window.

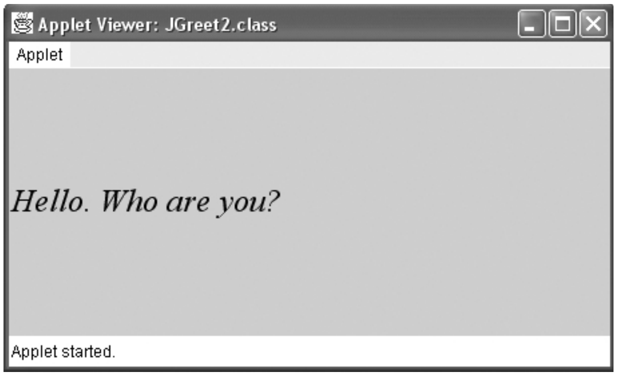


Figure 9-10 Output of the JGreet2.java program using bigFont

ADDING JTEXTFIELD AND JBUTTON COMPONENTS TO SWING APPLETS

In addition to including JLabels, Swing applets often contain other window features such as JTextFields and JButtons. A **JTextField** is a component into which a user can type a single line of text data. (Text data comprises any characters you can enter from the keyboard, including numbers and punctuation.) The structure of the JTextField class is shown in Figure 9-11.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.text.JTextComponent
                          |
                          +--javax.swing.JTextField
```

Figure 9-11 Structure of the JTextField class

Typically, a user types a line into a JTextField and then presses [Enter] on the keyboard or clicks a button with the mouse to enter the data. You can construct a JTextField object using one of several constructors:

- `public JTextField()` constructs a new JTextField.
- `public JTextField(int numColumns)` constructs a new empty JTextField with a specified number of columns.

- `public JTextField(String text)` constructs a new JTextField initialized with the specified text.
- `public JTextField(String text, int columns)` constructs a new JTextField initialized with the specified text and columns.

For example, to provide a JTextField for a user to answer the "Who are you?" question, you can code `JTextField answer = new TextField(10);` to provide a JTextField that is empty and displays approximately 10 characters. To add the JTextField named answer to an applet, you write `con.add(answer);`, where `con` is a Container object declared as `Container con = getContentPane();`.

Tip: The number of characters a JTextField can display depends on the font being used and the actual characters typed. For example, in most fonts, w is wider than i, so a JTextField of size 10 using Arial font can display 24 i characters, but only eight w characters.

Tip: Try to anticipate how many characters your users will enter when you create a JTextField. The user can enter more characters than those that display, but the extra characters scroll out of view. It can be disconcerting to try to enter data into a field that is not large enough. It is usually better to err on the high side when estimating the size of a user text field.

Several other methods are available for use with JTextFields. The **setText() method** allows you to change the text in a JTextField that has already been created, as in `answer.setText("Thank you");`. The **getText() method** allows you to retrieve the String of text in a JTextField, as in `String whatDidTheySay = answer.getText();`.

When a user encounters a JTextField you have placed within an applet, the user must position the mouse pointer in the JTextField and click to get an insertion point. When the user clicks within the JTextField, the JTextField has **keyboard focus**, which means that the next entries from the keyboard will be entered at that location. When you want the insertion point to appear automatically within the TextField without requiring the user to click in it first, you can use the **requestFocus() method**. For example, if you have added a JTextField named answer to an applet, then `answer.requestFocus()` causes the insertion point to appear within the JTextField, and the user can begin typing immediately without moving the mouse. In addition to saving the user some time and effort, requestFocus() is useful when you have several JTextFields and you want to direct the user's attention to a specific one. However, at any time, only one component within a window can have the keyboard focus.

When a JTextField has the capability of accepting keystrokes, the JTextField is **editable**. If you do not want the user to be able to enter data in a JTextField, you can use the **setEditable() method** to change the editable status of a JTextField. For example, if you want to give a user only one chance to answer a question correctly, then you can prevent the user from replacing or editing the characters in the JTextField by using the code `answer.setEditable(false);`. If conditions change, and you want the user to be able to edit the JTextField, use the code `answer.setEditable(true);`.

A JButton is even easier to create than a JTextField. There are five JButton constructors:

- `public JButton()` creates a button with no set text.
- `public JButton(Icon icon)` creates a button with an icon of type Icon or ImageIcon.
- `public JButton(String text)` creates a button with text.
- `public JButton(String text, Icon icon)` creates a button with initial text and an icon of type Icon or ImageIcon.

The structure of the JButton class is shown in Figure 9-12.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.AbstractButton
                          |
                          +--javax.swing.JButton
```

Figure 9-12 JButton class structure

To create a JButton with the label “Press when ready”, you write `JButton readyButton = new JButton("Press when ready");`. To add the JButton to an applet, you write `con.add(readyButton);`, where `Container con` has been created using `Container con = getContentPane();`. You can change a JButton’s Label with the **setLabel() method**, as in `readyJButton.setLabel("Don't press me again!");`, or get the JLabel and assign it to a String object with the **getLabel() method**, as in `String whatsOnJButton = readyButton.getLabel();`.

Tip: Make sure that the label on your JButton describes its function for the user.

Tip: As with JTextField components, you can use the `requestFocus()` method with JButton components. The surface of the button that has the keyboard focus appears with an outline so it stands out from the other JButtons.

Adding Multiple Components to a JApplet

Previously you have added only a single component to the Swing applet. In the following example you will add three components. To place multiple components at some given position in a container, you must use a **layout manager** to control component positioning. The normal default behavior of a Swing applet is to use a border layout if no layout manager is specified. **Border layouts**, created by using the BorderLayout class, divide a container into five sections: north, south, east, west, and center. A border layout is created with the **BorderLayout()** or **BorderLayout(int, int) constructor methods**. The statement `BorderLayout border = new BorderLayout();` creates a BorderLayout object named border. The statement `BorderLayout gap = new BorderLayout(5, 10);` creates a BorderLayout object named gap with a horizontal gap of five pixels and a vertical gap of 10 pixels between components. The components in the north, south, east, and west areas will take up as much space as needed; the center will use whatever space is left over.

When a component uses BorderLayout, components are placed in the center region. This means when you add multiple components to a JApplet, the components all lie in the center and appear to be on top of each other, so you can only see the last component you have added.

Tip

You will learn more about layout managers in Chapter 14.

When you use a flow layout, components do not lie on top of each other, instead the **flow layout manager** places components in a row, and when a row is filled, it automatically spills components onto the next row. The default positioning of the row of components is centered in the container. In the FlowLayout class there are three row positioning options specified by constants defined in the class. These options are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER. To create a layout manager named flow that positions the components to the right, use the statement `FlowLayout flow = new FlowLayout(FlowLayout.RIGHT);`. The layout of a container named con can be set to FlowLayout with row components positioned to the right using `con.setLayout(flow);`. A more compact syntax that combines the two statements into one is `con.setLayout(new FlowLayout(FlowLayout.RIGHT);`. You can use whichever syntax you prefer.

Next you will add a JTextField and a JButton to your Swing applet.

To add a JTextField and a JButton to the JGreet2 JApplet:

1. Open the **JGreet2.java** file in your text editor and change the class name to `JGreet3`.
2. Position the insertion point at the end of the line that defines the bigFont Font object, and then press **[Enter]** to start a new line of text.

3. Declare a JButton with the label "Press Me" and an empty JTextField by typing the following:

   ```
   JButton pressMe = new JButton("Press Me");
   JTextField answer = new JTextField("",10);
   ```

4. Set the new layout manager to a flow layout with the statement: `FlowLayout flow = new FlowLayout();`.
5. Position the insertion point at the end of the statement **`con.add(greeting);`**, press **[Enter]** to start a new line, and type **`con.setLayout(flow);`**.
6. Add the JTextField and the JButton to the Swing applet by typing the following:

   ```
   con.add(answer);
   con.add(pressMe);
   ```

7. On the next line, request focus for the answer by typing: **`answer.requestFocus();`**
8. Save the file as **JGreet3.java** and compile. Run the applet, changing the TestJGreet2.html document created earlier to **TestJGreet3.html**. Change the class name internally to **`JGreet3`**, and then execute the **`appletviewer TestJGreet3.html`** command. Output is shown in Figure 9-13. Confirm that you can type characters into the JTextField and that you can click the JButton using the mouse. You haven't coded any action to take place as a result of a JButton click yet, but the components should function. Also confirm that the objects in each row are positioned to the right.

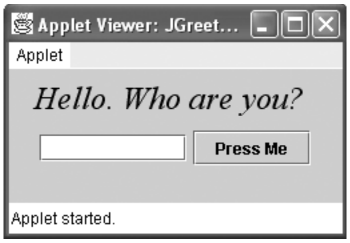


Figure 9-13 Output of the JGreet3.html document

9. Close the Applet Viewer window.

Learning About Event-Driven Programming

An **event** occurs when someone using your applet takes action on a component, such as clicking the mouse on a JButton object. The programs you have written so far in this book have been **procedural**—you dictated the order in which events occurred. You retrieved user input, wrote decisions and loops, and created output. When you retrieved user input, you had no control over how much time the user took to enter a response to a prompt, but you did control the fact that processing went no further until the input was completed. In contrast, with **event-driven programs**, the user might initiate any number of events in any order. For example, if you use a word-processing program, you have dozens of choices at your disposal at any moment in time. You can type words, select text with the mouse, click a button to change text to bold, click a button to change text to italics, choose a menu item, and so on. With each word-processing document you create, you choose options in any order that seems appropriate at the time. The word-processing program must be ready to respond to any event you initiate.

Within an event-driven program, a component on which an event is generated is the **source** of the event. A button that a user can click is an example of a source; a text field that a user can use to enter text is another source. An object that is interested in an event is a **listener**. Not all objects can receive all events—you probably have used programs in which clicking many areas of the screen has no effect. If you want an object, such as your applet, to be a listener for an event, you must register the object as a listener for the source.

Newspapers around the world register with news services, such as the Associated Press or United Press International. The news services maintain a list of subscribers, and send each one a story when important national or international events occur. Similarly, a Java component source object (such as a button) maintains a list of registered listeners and notifies all registered listeners (such as an applet) when any event occurs, such as a mouse click. When the listener "receives the news," an event-handling method that is part of the listener object responds to the event.

Tip

A source object and a listener object can be the same object. For example, a JButton can change its label when a user clicks it.

To respond to user events within any applet you create, you must do the following:

- Prepare your Swing applet to accept event messages.
- Tell your Swing applet to expect events to happen.
- Tell your Swing applet how to respond to any events that happen.

Preparing Your Swing Applet to Accept Event Messages

You prepare your applet to accept mouse events by importing the java.awt.event package into your program and adding the phrase `implements ActionListener` to the class header. The java.awt.event package includes event classes with names such as ActionEvent, ComponentEvent, and TextEvent. ActionListener is an **interface**, or a set of specifications for methods that you can use with Event objects. Implementing ActionListener provides you with standard event method specifications that allow your applet to work with ActionEvents, which are the types of events that occur when a user clicks a button.

You can identify interfaces such as ActionListener because they are implemented, and not imported or extended.

Telling Your Swing Applet to Expect Events to Happen

You tell your applet to expect ActionEvents with the **addActionListener() method**. If you have declared a JButton named aButton, and you want to perform an action when a user clicks aButton, then aButton is the source of a message, and you can think of your applet as a target to which to send a message. You learned in Chapter 4 that the `this` reference means "this current method," so `aButton.addActionListener(this);` causes any ActionEvent messages (button clicks) that come from aButton to be sent to "this current object."

Not all Events are ActionEvents with an addActionListener() method. For example, KeyListeners have an addKeyListener() method and FocusListeners have an addFocusListener() method. Additional event types and methods are covered in more detail in Chapters 13 and 14.

Telling Your Swing Applet How to Respond to Any Events That Happen

The ActionListener interface contains the **actionPerformed(ActionEvent e) method** specification. When a JApplet has registered as a listener with a JButton, and a user clicks the JButton, the actionPerformed() method executes. You must write the actionPerformed() method, which contains a header and a body like all methods. You use the header `public void actionPerformed(ActionEvent e)`, where `e` is any name you choose for the Event (the JButton click) that initiated the notification of the ActionListener (the JApplet). The body of the method contains any statements that you want to execute when the action occurs. You might want to perform mathematical calculations, construct new objects, produce output, or execute any other operation. For example, Figure 9-14 shows an actionPerformed() method that produces a line of output at the operating system prompt.

```
public void actionPerformed(ActionEvent someEvent)
{
   System.out.println
     ("I'm inside the actionPerformed() method!");
}
```

Figure 9-14 The actionPerformed() method that produces a line of output

Tip: When more than one component is added and registered to a Swing applet, it is necessary to determine which component was used for your program to act accordingly. ActionEvent and other event objects are part of the java.awt.event package and are subclasses of the EventObject class.

Every event-handling method is sent an event object of some kind. To determine the source of the event, the getSource() method of the object is used to determine the component that sent the event. For example, if the method header `public void ActionPerformed(ActionEvent e) {` is followed by the statement `Object source = e.getSource();`, the object returned by the getSource() method determines the component that sent the event. Continuing the example, if the source of the event is a JButton named exit, then the following `if` statement evaluates to `true` and the statements contained in its body execute:

```
if(source == exit)
{
//execute these statements
}
```

You can also use the `instanceof` keyword inside an event-handling method to determine the source of the event. The `instanceof` keyword is used when it is necessary to know only the component's type, rather than what component triggered the event. For example, if any JTextField, regardless of name, generates an action event when text is typed in it and [Enter] is pressed, the following `if` statement would execute:

```
void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if (source instanceof JTextField)
   {
   //execute these statements
   }
}
```

Next you will make your applet an event-driven program by adding functionality to your Swing applet. When the user enters a name and clicks the JButton, the JApplet will display a greeting on the command line.

To add functionality to your Swing applet:

1. Open the **JGreet3.java** file in your text editor and change the class name to **`JGreet4`**.
2. Add a third import statement to your program by typing: **`import java.awt.event.*;`**.
3. Position the insertion point at the end of the class header `public class JGreet4 extends JApplet`, press **[Spacebar]**, and then type **`implements ActionListener`**.

9

4. Position the insertion point at the end of the statement in the init() method that adds the pressMe button to the JApplet, and press **[Enter]**. Prepare your Swing applet for JButton-sourced events by typing the statement **`pressMe.addActionListener(this);`**.

5. Position the insertion point to the right of the closing curly brace for the init() method, and then press **[Enter]**. Add the following actionPerformed() method which follows the init() method but comes before the closing brace for the JGreet4 class. Use the object's getSource() method to determine that the source of the event is the JButton. Use an `if` statement to control the events that occur when the event's source is the JButton. You will declare a String to hold the user's name, use the getText() method on the answer JTextField to retrieve the String, and display an on-screen message to the user.

   ```
   public void actionPerformed(ActionEvent thisEvent)
   {
     Object source = thisEvent.getSource();
     if(source == pressMe)
     {
       String name = answer.getText();
       System.out.println("Hi there " + name);
     }
   }
   ```

6. Save the file as **JGreet4.java**, and compile the program. Edit the file TestJGreet.html and change the class reference to **`TestJGreet4.class`**. Save the file as **TestJGreet4.html**. Run the program using the **`appletviewerTestJGreet4.html`** command.

7. Type your name in the JTextField, and then click the **Press Me** button. Examine your command-prompt screen. The personalized message ("Hi there" and your name) should appear on the command prompt screen.

You might need to drag the Applet Viewer window to a new position so you can see the output on the command line.

8. Drag the mouse to highlight the name in the text field in the Applet Viewer window, and then type a different name. Click the **Press Me** button. A new greeting appears on the command-line screen.

9. Close the Applet Viewer window.

When Swing applets contain a JTextField, there are two ways to get the applet to accept user input. You can enter text and click a button, or you can enter text and press [Enter]. If your Swing applet needs to receive an event message from a JTextField, then you must make your applet a registered Event listener with the JTextField.

To add the ability to press [Enter] from within the JTextField for input:

1. In the JGreet4.java text file, change the class name to **`JGreet5`**, position the insertion point at the end of the statement `pressMe.addActionListener(this);`, and then press **[Enter]**.
2. Make the answer field accept input by typing: **`answer.addActionListener(this);`**
3. Add the following statements to the end of the ActionPerformed method that uses the `instanceof` keyword to test for an action event generated by the JTextField named answer:

```
else if(source instanceof JTextField)
    {
      String name = answer.getText();
      System.out.println("Hi there " + name);
    }
```

4. Save the file as **JGreet5.java**, and compile the program. Edit the file TestJGreet4.html and change the class reference to **`TextJGreet5.class`**. Save the file as **TestJGreet5.html**. Run the program using the **`appleviewer TestJGreet5.html`** command. To confirm that you can cause the message to appear, type a name and then press **[Enter]**.
5. Close the Applet Viewer window.

Adding Output to a Swing Applet

A Swing applet that produces output on the command-line screen is not very exciting. Naturally, you will want to make changes as various events occur. For example, rather than using `System.out.println("Hi" + name);` to send a greeting to the command-line screen, you might want to add a greeting to the Swing applet itself. One approach is to create a new JLabel that gets added to the applet with the add() method after the user enters a name. You can declare a new, empty JLabel with the statement `JLabel personalGreeting = new Label("");`. After the name is retrieved, you can use the setText() method to set the JLabel text for personalGreeting to `"Hi there " + name`.

To add a personalGreeting JLabel to the Swing applet:

1. Within the JGreet5.java text file, change the class name to **`JGreet6`**, and then remove both **`System.out.println("Hi" + name);`** statements from the actionPerformed() method.
2. Position the insertion point at the end of the statement `JTextField answer = new JTextField("",10);` and press **[Enter]**. To declare

a new JLabel named personalGreeting, type the statement:
`JLabel personalGreeting = new JLabel("");`

3. Position the insertion point in the init() method after the `con.add(pressMe);` statement, and then add the JLabel personalGreeting with the statement **`con.add(personalGreeting);`**, then press **[Enter]**.
4. Add the following statement to the actionPerformed() method after the `String name = answer.getText();` statement to set the text of the personalGreeting. Be sure to add the statement to the body of both the `if` and `if…else` statements.

   ```
   personalGreeting.setText("Hi " + name);
   ```

5. Save the program as **JGreet6.java**, and compile the file. Edit the file TestJGreet5.html and change the class reference to **`TestJGreet6.class`**. Save the file as **TestJGreet6.html**. Run the program using the **`appletviewer TestJGreet6.html`** command. Type a name in the JTextField, and then press **[Enter]** or click the **Press Me** button.
6. Close the Applet Viewer window.

If you can add components to an applet, you should also be able to remove them; you do so with the **remove() method**. For example, after a user enters a name into the JTextField, you might not want the user to use the JTextField or its JButton again, so you can remove them from the applet. To use the remove() method, you place the component's name within the parentheses. As with the add() method, you must redraw the applet after the remove() method to display the effects.

To remove the JTextField and JButton from the Greet applet:

1. Open the **JGreet6.java** file, if necessary, and rename the class **`JGreet7`**. Place the insertion point after the closing curly brace of the `if…else` statement in the actionPerformed() method, and press **[Enter]**. Then enter the following statements. Note that the repaint() method causes the Swing applet to redraw after the JButton and JTextField are removed from the screen.

   ```
   remove(answer);
   remove(pressMe);
   repaint();
   ```

2. Save the file **as JGreet7.java** and compile the file. Edit the file TestJGreet6.html and change the class reference to `TestJGreet.class`. Save the file as **TestJGreet7.html**. Run the program using the **`appletviewer TestJGreet7.html`** command. Enter a name, and then either press **[Enter]** or click the **Press Me** button. Notice that the JTextField and the JButton disappear from the screen.
3. Close the Applet Viewer window.

UNDERSTANDING THE SWING APPLET LIFE CYCLE

Swing applets are popular because they are easy to use in a Web page. Because applets execute in a browser, the JApplet class contains methods that are automatically called by the browser. Earlier in this chapter you learned the names of four of these methods: init(), start(), stop(), and destroy().

You have already written your own init() methods. When you write a method that has the same method header as an automatically provided method, you replace or **override** the original version. Every time a Web page containing a Swing applet is loaded in the browser or when you run the `appletviewer` command within an HTML document that calls a Swing applet, if you have written an init() method for the Swing applet, that method executes; otherwise the automatically provided init() method executes. You should write your own init() method when you have any initialization tasks to perform, such as setting up user interface components.

Tip

When you override a method, you create your own version that Java uses, instead of using the automatically supplied version with the same name. It is not the same as overloading a method, which is writing several methods that have the same name but take different arguments. You learned about overloading methods in Chapter 4.

The **start() method** executes after the init() method, and it executes again every time the applet becomes active after it has been inactive. For example, if you run a Swing applet using the `appletviewer` command, and then minimize the AppletViewer window, the Swing applet becomes inactive. When you restore the window, the Swing applet becomes active again. On the Internet, users can leave a Web page, visit another page, and then return to the first site. Again, the Swing applet becomes inactive, and then active. When you write your own start() method you must include any actions you want your Swing applet to take when a user revisits the Swing applet. For example, you might want to resume some animation that you suspended when the user left the applet.

When a user leaves a Web page (perhaps by minimizing a window or traveling to a different Web page) the **stop() method** is invoked. You override the existing empty stop() method only if you want to take some action when a Swing applet is no longer visible. You don't usually need to write your own stop() methods.

The **destroy() method** is called when the user closes the browser or Applet Viewer. Closing the browser or AppletViewer releases any resources the Swing applet might have allocated. As with the stop() method, you do not usually have to write your own destroy() methods.

Tip

Advanced Java programmers override the stop() and destroy() methods when they want to add instructions to "suspend a thread," or stop a chain of events that were started by a Swing applet, but which are not yet completed.

Again, every Swing applet has the same life cycle outline, as shown in Figure 9-15. When it executes, the init() method runs, followed by the start() method. If the user leaves the Swing applet's page, the stop() method executes. When the user returns, the start() method executes. The stop() and start() sequence might continue any number of times, until the user closes the browser (or AppletViewer) which invokes the destroy() method.

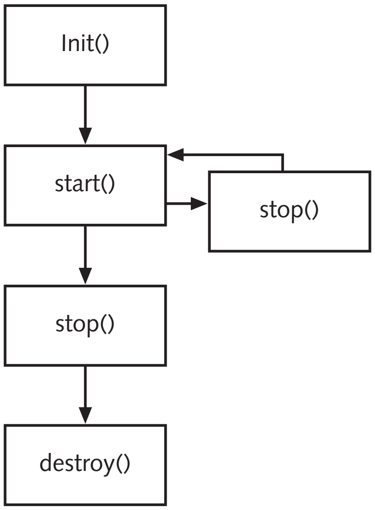


Figure 9-15 Swing applet life cycle

To demonstrate the life cycle methods in action, you can write a Swing applet that overrides all four methods. Note the number of times each method executes.

To demonstrate the life cycle of a Swing applet:

1. Open a new text file in your text editor, and then type the following import statements:

   ```
   import javax.swing.*;
   import java.awt.*;
   import java.awt.event.*;
   ```

2. So the Swing applet will include a JButton that the user can click and, so the ActionListener will be implemented, type the following header for a JLifeCycle applet:

   ```
   public class JLifeCycle extends JApplet implements
     ActionListener
   ```

3. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again to start a new line.

4. Declare the following six JLabel objects that you will use to display each name of the six methods that will execute during the lifetime of the Swing applet:

```
JLabel messageInit = new JLabel("init ");
JLabel messageStart = new JLabel("start ");
JLabel messageDisplay = new JLabel("display ");
JLabel messageAction = new JLabel("action ");
JLabel messageStop = new JLabel("stop ");
JLabel messageDestroy = new JLabel("destroy ");
```

5. Declare a JButton by typing: **`JButton pressButton = new JButton("Press");`**.

6. Declare six integers that will hold the number of occurrences of each of the six methods by typing the following code on one line:

```
int countInit, countStart, countDisplay, countAction,
countStop, countDestroy;
```

7. Start the init() method by adding a container and flow layout manager with the statements:

```
public void init()
{
  Container con = getContentPane();
  con.setLayout (new FlowLayout());
```

8. Add the following statements, which adds one to countInit, places the components within the applet, and then calls the display() method:

```
  ++countInit;
  con.add(messageInit);
  con.add(messageStart);
  con.add(messageDisplay);
  con.add(messageAction);
  con.add(messageStop);
  con.add(messageDestroy);
  con.add(pressButton);
  pressButton.addActionListener(this);
  display();
}
```

9. Add the following start() method, which adds one to countStart and calls display():

```
public void start()
{
  ++countStart;
  display();
}
```

10. Add the following display() method, which adds one to countDisplay, displays the name of each of the six methods with the current count, and indicates how many times the method has executed:

```
public void display()
{
  ++countDisplay;
  messageInit.setText("init " + countInit);
  messageStart.setText("start " + countStart);
  messageDisplay.setText("display " + countDisplay);
  messageAction.setText("action " + countAction);
  messageStop.setText("stop " + countStop);
  messageDestroy.setText("destroy " + countDestroy);
}
```

11. Add the following stop() and destroy() methods, which each add one to the appropriate counter and call display():

```
public void stop()
{
  ++countStop;
  display();
}
public void destroy()
{
  ++countDestroy;
  display();
}
```

12. When the user clicks pressButton, the following actionPerformed() method will execute; it adds one to countAction and displays it. Enter the method:

```
public void actionPerformed(ActionEvent e)
{
  Object source = e.getSource();
  if(source == pressButton)
  {
    ++countAction;
    display();
  }
}
```

13. Add the closing curly brace for the class. Save the program as **JLifeCycle.java** in the Chapter.09 folder on your Student Disk. If necessary, compile, correct any errors, and compile again.

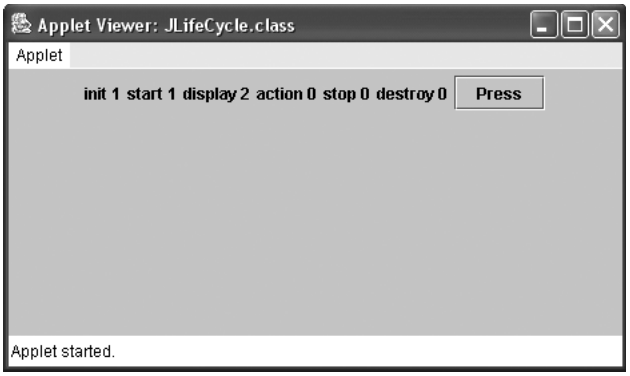
Take a moment to examine the code you created for JLifeCycle.java. Each method adds one to one of the six counters, but you never explicitly call any of the methods except display(); each of the other methods will be called automatically. Next you will create an HTML document so you can test JLifeCycle.java.

To create an HTML document to test JLifeCycle.java:

1. Open a new text file in your text editor.
2. Enter the following HTML code:

```
<HTML>
<APPLET CODE="JLifeCycle.class" WIDTH = 460 HEIGHT = 200>
</APPLET>
</HTML>
```

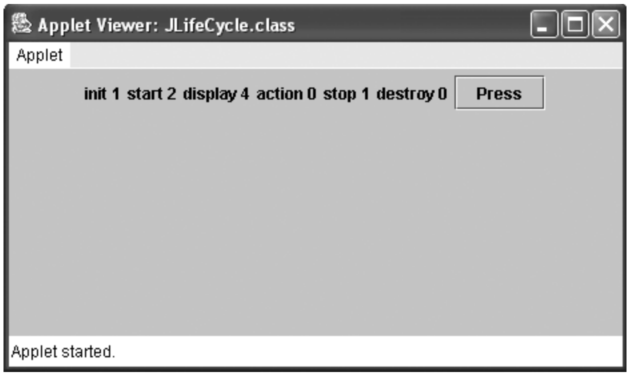
3. Save the file as **TestJLifeCycle.html** in the Chapter.09 folder on your Student Disk.
4. Run the HTML document using the command **`appletviewer TestJLifeCycle.html`**. Figure 9-16 shows the output. When the applet begins, the init() method is called, so one is added to countInit. The init() method calls display(), so one is added to countDisplay. Immediately after the init() method executes, the start() method is executed, and one is added to countStart. The start() method calls display(), so one more is added to countDisplay. The first time you see the applet, countInit is 1, countStart is 1, and countDisplay is 2. The methods actionPerformed(), stop(), and destroy() have not yet been executed.



```
init 1 start 1 display 2 action 0 stop 0 destroy 0
Applet started.
```

Figure 9-16 JLifeCycle Swing applet after start-up

5. Click the **Minimize** button to minimize the Applet Viewer window, and then click the **Taskbar** button to restore it. The applet now looks like Figure 9-17. The init() method still has been called only once, but when you minimized the applet, the stop() method executed, and when you restored it, the start() method executed. Therefore, countStop is now 1 and countStart has increased to 2. Additionally, because start() and stop() call display(),countDisplay() is increased by two and now holds the value 4.

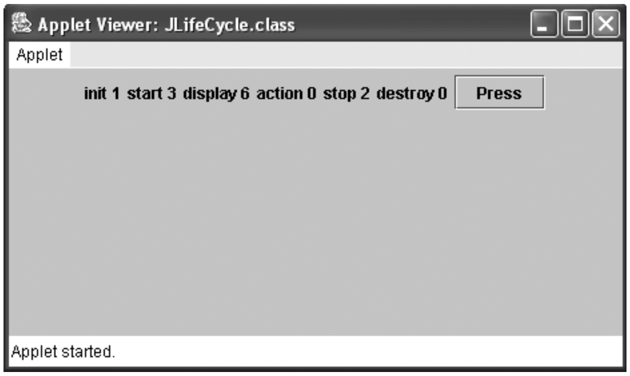


```
init 1 start 2 display 4 action 0 stop 1 destroy 0
```

Applet started.

Figure 9-17 JLifeCycle Swing applet after being minimized and restored

6. Minimize and maximize the Applet Viewer window again. Now the stop() method has executed twice, the start() method has executed three times, and the display() method has executed a total of six times, as shown in Figure 9-18.

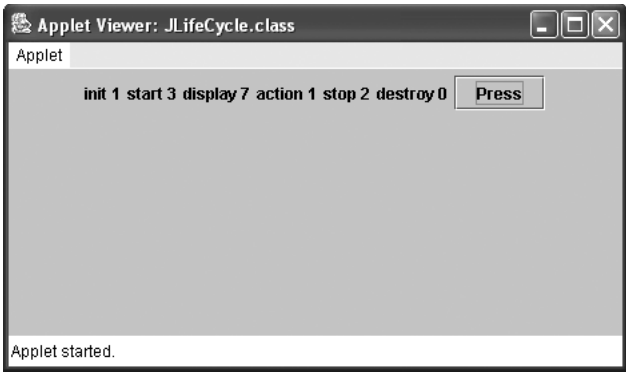


```
init 1 start 3 display 6 action 0 stop 2 destroy 0
```

Applet started.

Figure 9-18 JLifeCycle Swing applet after being minimized and restored twice

7. Click the **Press** button. The count for the actionPerformed() method is now 1, and actionPerformed() calls display(), so countDisplay() is now 7, as shown in Figure 9-19.
8. Continue to minimize, maximize, and click the **Press** button. Note the changes that occur with each activity until you can correctly predict the outcome. Notice that the destroy() method is not executed until you close the applet, and then it is too late to observe an increase in countDestroy.



```
init 1 start 3 display 7 action 1 stop 2 destroy 0
Applet started.
```

Figure 9-19 JLifeCycle Swing applet after clicking the Press button

Creating a More-Sophisticated Interactive Swing Applet

You are now able to create a fairly complex application or applet. Next you will create an applet that contains several components, receives user input, makes decisions, uses arrays, performs output, and reacts to the applet life cycle.

The JPartyPlanner Swing applet lets its user estimate the cost of an event hosted by Event Handlers Incorporated. Event Handlers uses a sliding fee scale so the per-guest cost decreases as the total number of invited guests increases. Table 9-1 shows the fee structure.

Table 9-1 Cost per guest for events

Number of Guests	Cost per Guest
1 to 24	$27
25 to 49	$25
50 to 99	$22
100 to 199	$19
200 to 499	$17
500 to 999	$14
1000 and over	$11

The Swing applet lets the user enter a number of anticipated guests. The user can press [Enter] or click a JButton to perform the fee lookup and event cost calculation. Then the Swing applet displays the cost per person as well as the total cost for the event. The user can continue to request fees for a different number of guests and view the results for any length of time before making another request or leaving the page. When the user leaves the page, however, you will erase the last number of requested guests and ensure that the next user starts fresh with zero guests.

To begin creating an interactive party planner Swing applet:

1. Open a new text file in your text editor.
2. Type the following import statements, the JPartyPlanner class header, and the opening curly brace for the class:

   ```
   import javax.swing.*;
   import java.awt.*;
   import java.awt.event.*;
   public class JPartyPlanner extends JApplet implements
     ActionListener
   {
   ```

3. You will need several components: a JLabel for the company name, a JButton the user can click to perform a calculation, and two more JLabels to display output. Add the following code to implement these components:

   ```
   JLabel companyName = new
     JLabel("Event Handlers Incorporated");
   JButton calcButton = new JButton("Calculate");
   JLabel perPersonResult = new JLabel("Plan with us.");
   JLabel totalResult = new JLabel("The more the merrier");
   ```

4. To enhance the appearance, create a Font object by typing the following:

   ```
   Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
   ```

5. Use the init() method to place components within the applet screen, and then prepare the JButton to receive action messages by typing the following:

   ```
   public void init()
   {
   Container con = getContentPane();
   con.setLayout(new FlowLayout());
   companyName.setFont(bigFont);
   con.add(companyName);
   con.add(calcButton);
   calcButton.addActionListener(this);
   con.add(perPersonResult);
   con.add(totalResult);
   }
   ```

6. Add the following start() method, which executes when the user leaves the Swing applet and resets the JLabel and the data-entry JTextField:

   ```
   public void start()
   {
     perPersonResult.setText("Plan with us.");
     totalResult.setText("The more the merrier");
     repaint();
   }
   ```

7. Save the partially completed Swing applet as **JPartyPlanner.java** in the Chapter.09 folder on your Student Disk.

You finished the init() and start() methods for the JPartyPlanner Swing applet, placed each component in the Swing applet, and reinitialized each component every time a user returns to the Swing applet after leaving. The Swing applet doesn't actually do anything yet; most of the applet's work will be contained in the actionPerformed() method, the most complicated method in this applet.

Next you will create the actionPerformed() method. You begin by declaring two parallel arrays—one array will hold guest limits for each of six event rates, and the other array will hold the actual rates.

To complete the JPartyPlanner Swing applet:

1. Use a dialog box to receive user input for the number of guests. Enter the following method header for actionPerformed() and declare two arrays for guest limits and rates:

   ```
   public void actionPerformed(ActionEvent e)
   {
   Object source = e.getSource();
     if(source == calcButton)
     {
       String response = JOptionPane.showInputDialog(null,
         "Enter the number of guests");
       int[] guestLimit =   {0, 25, 50, 100, 200, 500, 1000};
       int[] ratePerGuest = {27, 25, 22, 19, 17, 14, 11};
   ```

2. Next add the following variable to hold the number of guests. The user will receive input from a dialog box, but you need an integer to perform calculations so you can use the parseInt() method.

   ```
   int guests = Integer.parseInt(response);
   ```

Tip

You learned about the parseInt() method in Chapter 6.

3. You need two variables—one will hold the individual, per-person fee for an event, and the other will hold the fee for the entire event. Enter the following variables:

   ```
   int individualFee = 0, eventFee = 0;
   ```

4. Enter the following variables to use as subscripts for the arrays:

   ```
   int x = 0, a = 0;
   ```

   There are several ways to search through the guestLimit array to discover the appropriate position of the per person fee in the ratePerGuest array. One possibility is to use a `for` loop and decrement from six down to zero. If the number of guests is greater than or equal to any value in the guestLimit array, then the corresponding per person rate in the ratePerGuest array is the correct rate. After finding the correct individual rate, you determine the price for the entire event by multiplying the individual rate by the number of guests. After finding the appropriate individual fee for a given event, you no longer want to search through the guestLimit array, so you set the subscript x equal to zero to force an early exit from the `for` loop.

5. Enter the following `for` loop:

```
for(x = 6; x >= 0; --x)
  if(guests >= guestLimit[x])
    {
      individualFee = ratePerGuest[x];
      eventFee = guests * individualFee;
      x = 0;
    }
```

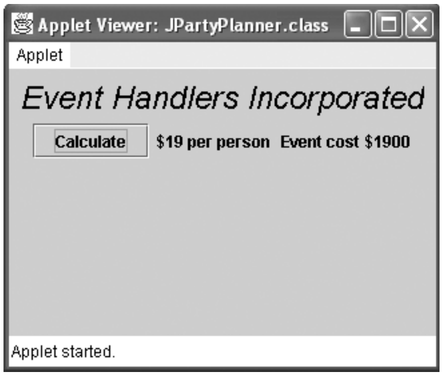
6. The only tasks that still must be included in the actionPerformed() method involve producing output for the user. Enter the following code to accomplish this processing:

```
perPersonResult.setText("$" + individualFee + " per
  person");
totalResult.setText("Event cost $" + eventFee);
```

7. Add three closing curly braces—two for the actionPerformed() method, and one for the entire JPartyPlanner Swing applet.
8. Save the file, and then compile it at the command prompt.
9. Open a new text file in your text editor, and then create the following HTML document to test the applet:

```
<HTML>
<APPLET CODE="JPartyPlanner.class"
  WIDTH = 320 HEIGHT = 200>
</APPLET>
</HTML>
```

10. Save the HTML document as **TestJPartyPlan.html** in the Chapter.09 folder on your Student Disk. Then use the **`appletviewer`** command to execute the file. Test the applet with different numbers of guests until you are sure that the per person rates and event rates are correct. Minimize and restore the Applet Viewer window and observe that any calculated fees are replaced with start() messages. For example, if you enter 100 guests, then your output should resemble Figure 9-20.
11. Close the Applet Viewer window.



Event Handlers Incorporated

$19 per person Event cost $1900

Figure 9-20 Output of JPartyPlanner Swing applet

Using the setLocation() and setEnabled() Methods

A serious shortcoming of the objects you have written so far is that you cannot choose the location of the JLabel and JButton objects you place within your Swing applets. When you use the add() method to add a component to an applet, it seems to have a mind of its own as to where it is physically placed. Although you must learn more about the Java programming language before you can change the initial placement of components when you use the add() method, you can use the setLocation() method to change the location of a component at a later time. The **setLocation() method** allows you to place a component at a specific location within the Applet Viewer window.

Any applet window consists of a number of horizontal and vertical pixels on the screen. You set the pixel values in the HTML document you write to test the Swing applet. Any component you place on the screen has a horizontal, or **x-axis**, position as well as a vertical, or **y-axis**, position in the window. The upper-left corner of any display is position 0,0. The first, or **x-coordinate**, value increases as you travel from left to right across the window. The second, or **y-coordinate**, value increases as you travel from top to bottom.

For example, to position a Label object named someLabel at the upper-left corner of a window, you write `someLabel.setLocation(0,0);`. If a window is 200 pixels wide and 100 pixels tall, then you can place a Button named pressMe in the approximate center of the window with the statement `pressMe.setLocation(100,50);`. Figure 9-21 illustrates the screen coordinate positions.

Tip

You can picture a coordinate as an infinitely thin line that lies between the pixels of the output device.

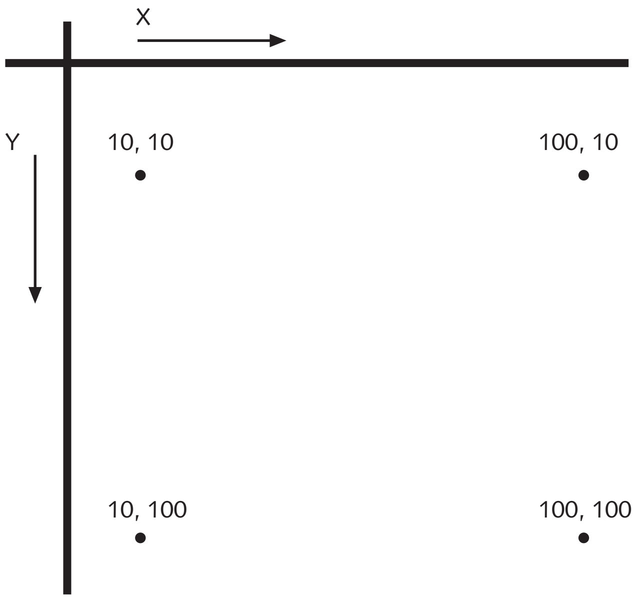


Figure 9-21 Screen coordinate positions

Tip: When you use setLocation(), the upper-left corner of the component is placed at the specified x- and y-coordinates. If a window is 100 by 100 pixels, then `aButton.setLocation(100,100);` places the JButton outside the window, where you cannot see the component.

Next you will create a JLabel that changes its location with each JButton click.

To create a moving JLabel:

1. Open a new text file in your text editor, and then type the following import statements:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

2. Type the following class header and opening curly brace for a class named JMoveLabel. You must implement ActionListener because the Swing applet requires that the user click a JButton as the action event.

```
public class JMoveLabel
  extends JApplet implements ActionListener
```

3. Declare the following JLabel, JButton, and two integers that will hold the horizontal and vertical coordinates of the JLabel:

```
JLabel movingMsg = new JLabel("Event Handlers Inc.");
JButton pressButton = new JButton("Press");
int xLoc = 20, yLoc =20;
```

4. Enter the following init() method to add the components to the Swing applet screen and prepare the JButton to receive messages:

```
public void init()
{
  Container con = getContentPane();
  con.setLayout(new FlowLayout());
  con.add(movingMsg);
  con.add(pressButton);
  pressButton.addActionListener(this);
}
```

5. Enter the following actionPerformed method() to move the message 10 pixels to the right and 10 pixels down each time the user clicks the JButton:

```
public void actionPerformed(ActionEvent e)
{
  Object source = e.getSource();
  if(source == pressButton)
     movingMsg.setLocation(xLoc+=10, yLoc+=10);
}
```

6. Add the closing curly brace to the class.
7. Save the file as **JMoveLabel.java** in the Chapter.09 folder on your Student Disk, and then compile it.
8. Open a new file in your text editor, and then create the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE="JMoveLabel.class" WIDTH = 460 HEIGHT = 300>
</APPLET>
</HTML>
```

9. Save the HTML document as **Test JMoveLabel.html** in the Chapter.09 folder on your Student Disk. Then run the file using the **`appletviewer TestJMoveLabel.html`** command. Click the **pressButton** and observe how the JLabel moves each time you click it. If you click enough times, the JLabel moves off the screen.
10. Close the Applet Viewer window.

The setEnabled() Method

You probably have used computer programs in which a component becomes disabled or unusable. For example, a JButton might become dim and unresponsive when the programmer no longer wants you to have access to the JButton's functionality. You can use the **setEnabled() method** with a component to make it unavailable and, in turn, to make it available again. The setEnabled() method takes an argument of `true` if you want to enable a component, or `false` if you want to disable a component.

Tip: When you create a component, it is enabled by default.

For example, in the JMoveLabel applet, a user can continue to click the JButton until the JLabel moves completely off the screen. If you want to prevent this from happening, you can disable the JButton after the JLabel has advanced as far as you want it to go. Next you will stop the JLabel from moving after it reaches a y-coordinate of 280.

To disable the JButton:

1. Open the **JMoveLabel.java** file in your text editor and change the class name to **JMoveLabel2**.
2. Position the insertion point at the end of the `if` statement in the actionPerformed() method, and then press **[Enter]** to start a new line of text. Add the following statement to disable the JButton when the message has moved to a y-coordinate of 280:

```
if(yLoc==280)
  pressButton.setEnabled(false);
```

3. Save the file as **JMoveLabel2.java** and compile. Change the Test JMoveLabel.html document created earlier to **Test JMoveLabel2.html**, change the class internally to **`JMoveLabel2.class`**, and then type the **`appletviewer TestJMoveLabel2.html`** command to run the applet. Click **pressButton** until the JButton is disabled and the JLabel cannot descend any farther.
4. Close the Applet Viewer window.

Chapter Summary

- Swing applets are programs that are called from within another application. You run them within a Web page, or within another program called Applet Viewer, which comes with the Java Developer's Kit. An applet must be called from within an HTML (Hypertext Markup Language) document.
- A component is not added directly to a Swing applet. Instead, you use the add() method to add a component to a container.
- Every Swing applet includes four methods: `public void init()`, `public void start()`, `public void stop()`, and `public void destroy()`.
- A JTextField is a component into which a user can type a single line of text data. Typically, a user types a line into a JTextField and then inputs the data by pressing [Enter] on the keyboard or clicking a JButton with the mouse. The setText() method allows you to change the text in a previously created TextField. The getText() method allows you to retrieve the String of text in a TextField.

- You can create a JButton with or without a label. You can change a JButton's label with the setLabel() method, or get the JLabel and assign it to a String object with the getLabel() method.
- JLabel is a built-in class that holds text that can be displayed within a Swing applet. The setText() method assigns text to a JLabel or any other component.
- An event occurs when a Swing applet's user takes action on a component, such as using the mouse to click a JButton object. In event-driven programs, the user might initiate any number of events in any order. Within an event-driven program, a component on which an event is generated is the source of the event. An object that is interested in an event is a listener.
- To respond to user events within any Swing applet you create, you must prepare your applet to accept event messages, tell your applet to expect events to happen, and then tell your applet how to respond to any events that happen. Adding `implements ActionListener` to an applet's class header prepares a Swing applet to receive event messages.
- An ActionEvent is the type of event that occurs when a user clicks a JButton. You tell an applet to expect ActionEvents with the addActionListener() method. The ActionListener interface contains the actionPerformed(ActionEvent e) method specification. In the body of the method, you write any statements that you want to execute when an action takes place.
- Every event-handling method is sent an event of some kind. The object's getSource() method can be used to determine the component that sent the event.
- When you use the add() method to add a component to an applet, you do not specify the physical location of the component in the Applet Viewer window. The setLocation() method allows you to place a component at a specific location within an Applet Viewer window. When you include x- and y-coordinates within the setLocation() method, the upper-left corner of the component is placed at the specified location.
- You can use the setEnabled() method with a component to make it unavailable and, in turn, to make it available again. The setEnabled() method takes an argument of `true` if you want to enable a component, or `false` if you want to disable a component.

REVIEW QUESTIONS

1. A major difference between AWT and Swing applets is ________________.
   a. the AWT applet uses a content pane
   b. they are executed using different Java commands
   c. they are executed from within different HTML documents
   d. the Swing applet imports from the .javax.swing package

2. A program that allows you to display HTML documents on your computer screen is a ________________.
   a. search engine
   b. compiler
   c. browser
   d. server
3. The name of any Swing applet called using CODE within an HTML document must use the ________________ extension.
   a. .exe
   b. .code
   c. .java
   d. .class
4. The ________________ is a String representing a font.
   a. point size
   b. style
   c. leading
   d. typeface
5. A JTextField is a Swing component ________________.
   a. into which a user can type a single line of text data
   b. into which a user can type multiple lines of text data
   c. that automatically has focus when the applet runs
   d. whose text cannot be changed
6. The Swing add() method ________________.
   a. adds two integers
   b. adds a component directly to the Swing applet
   c. places a component within a container
   d. places a text value within an applet component
7. The start() method called in any Swing applet is called ________________.
   a. at start-up
   b. when the user closes the browser
   c. when a user revisits an applet
   d. when a user leaves a Web page

8. A Font object contains all of the following arguments except ________________.
   a. language
   b. typeface
   c. style
   d. point size
9. To respond to user events within a Swing applet, you must ________________.
   a. prepare the applet to accept event messages
   b. import the java.applet.* package
   c. tell your applet how to respond to any events that happen
   d. accomplish both a and c
10. The constructor `public JButton("4")` creates ________________.
    a. an unlabeled JButton
    b. a JButton four pixels wide
    c. a JButton four characters wide
    d. a JButton with a "4" on it
11. An event occurs when a ________________.
    a. component requests focus
    b. component is enabled
    c. component sets text
    d. button is clicked
12. ActionListener is an example of a(n) ________________.
    a. import
    b. applet
    c. interface
    d. component
13. When a Swing applet is registered as a listener with a JButton, if a user clicks the JButton, the method that executes is ________________.
    a. buttonPressed()
    b. addActionListener()
    c. start()
    d. actionPerformed()

14. When you write a method that has the same method header as an automatically provided method, you ________________ the original version.
    a. destroy
    b. override
    c. call
    d. copy
15. Which of the following statements creates a JLabel that says "Welcome"?
    a. `JLabel = new JLabel("Welcome");`
    b. `JLabel aLabel = JLabel("Welcome");`
    c. `aLabel = new JLabel("Welcome");`
    d. `JLabel aLabel = new JLabel("Welcome");`
16. Which of the following statements correctly creates a Font object?
    a. `Font aFont = new Font("TimesRoman", Font.ITALIC, 20);`
    b. `Font aFont = new Font(30, "Helvetica", Font.ITALIC);`
    c. `Font aFont = new Font(Font.BOLD,"Helvetica", 24);`
    d. `Font aFont = new Font(22, Font.BOLD,"TimesRoman");`
17. The method that positions a component within an applet is ________________.
    a. position()
    b. setPosition()
    c. location()
    d. setLocation()
18. In a window that is 200 x 200 pixels, position 10, 190 is nearest to the ________________ corner.
    a. upper-left
    b. upper-right
    c. lower-left
    d. lower-right
19. An object's ________________ method can be used to determine the component that sends an event.
    a. getSource()
    b. instanceof()
    c. both of the above
    d. none of the above

20. Which of the following statements disables a component named someComponent?
    a. `someComponent.setDisabled();`
    b. `someComponent.setDisabled(true);`
    c. `someComponent.setEnabled(false);`
    d. `someComponent.setEnabled(true);`

Exercises

1. Create a Swing applet with a JButton labeled "Who's number one?". When the user clicks the button, display your favorite team in a large font. Save the program in the Chapter.09 folder on your Student Disk as **JNumberOne.java**.
2. a. Create a Swing applet that asks a user to enter a password into a JTextField and to then press [Enter]. Compare the password to "Rosebud"; if it matches, display "Access Granted"; if not, display "Access Denied". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordA.java**.
   b. Modify the password applet in Exercise 2a to ignore differences in case between the typed password and "Rosebud". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordB.java**.
   c. Modify the password applet in Exercise 2b to compare the password to a list of five valid passwords: "Rosebud", "Redrum", "Jason", "Surrender", or "Dorothy". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordC.java**.
3. Create a Swing applet with a JButton. When the user clicks the JButton, change the font typeface and style. Save the program in the Chapter.09 folder on your Student Disk as **JChangeFont.java**.
4. Create a Swing applet that displays the date and time in a JTextField with the JLabel "Today is" when the user clicks a JButton. Save the program in the Chapter.09 folder on your Student Disk as **JDayOfYear.java**.
5. Create a Swing applet that displays an employee's title in a JTextField when the user types the employee's first and last names (separated by a space) in another JTextField. Include JLabels for each JTextField. You can use arrays to store the employees' names and titles. Save the program in the Chapter.09 folder on your Student Disk as **JEmployeeTitle.java**.
6. Create a Swing applet that displays an employee's title in a TextField when the user types an employee's first and last names (separated by a space) in another JTextField, or displays an employee's name in a JTextField when the user types the employee's title in a JTextField. Include a JLabel for each JTextField. Add a JLabel at the top of the applet with the text "Enter a name or a title". You can use arrays to store the employees' names and titles. Save the program in the Chapter.09 folder on your Student Disk as **JEmployeeTitle2.java**.

7. Create a Swing applet that uses a dialog box to enter an integer. When the user clicks a JButton, the user is prompted to enter an integer. When the user clicks the OK button, the integer is doubled and the answer is displayed. Save the program in the Chapter.09 folder on your Student Disk as **JDialogDouble.java**.
8. Create a Swing applet that allows the user to enter two integers into two separate dialog boxes. When the user clicks a JButton, the sum of the integers is displayed. Save the program in the Chapter.09 folder on your Student Disk as **JDialogAdd.java**.
9. a. Create an applet named DivideTwo that allows the user to enter two integers in two separate JTextFields. The user can click a JButton to divide the first integer by the second integer and display the result.

   b. Modify the DivideTwo applet created in Exercise 9a so that if a user enters zero for the second integer, when the user clicks the JButton to divide, the Swing applet displays the message "Division by zero not allowed!" Save the final program in the Chapter.09 folder on your Student Disk as **JDivideMe.java**.
10. a. Create a payroll Swing applet named CalcPay that allows the user to enter two double values—hours worked, and an hourly rate. When the user clicks a JButton, gross pay is calculated. Save the program in the Chapter.09 folder on your Student Disk as **JCalculatePay.java**.

    b. Modify the payroll Swing applet created in Exercise 10a so that federal withholding tax is subtracted from gross pay based on the following table:

Income$	Withholding%
0 to 99.99	10
100.00 to 299.99	15
300.00 to 599.99	21
600.00 and up	28

    Save the program in the Chapter.09 folder on your Student Disk as **JCalculatePay2.java**.
11. Create a conversion Swing applet that prompts the user to enter a distance in miles in a dialog box, then converts miles to kilometers and displays the result in a JTextField as "XX.XX kilometers", where XX.XX is the number of kilometers. You can use the formula miles *1.6 to convert miles to kilometers. Save the program in the Chapter.09 folder on your Student Disk as **JConversion.java**.
12. Create a Swing applet that calculates the current balance in a checking account in a JTextField. The user enters the beginning balance, check amount, and deposit amount in separate JTextFields with the appropriate JLabels. After the applet calculates the current balance, reposition the JTextFields and JLabels so that the beginning balance appears on the first line, the check and deposit amounts appear on the second line, and the new balance appears on the third line. Save the program in the Chapter.09 folder on your Student Disk as **JCalculateBalance.java**.

13. Create a Swing applet that displays two of your family members' names, relationships to yourself, and ages, in JTextFields when you click a JButton. Each JTextField should have a JLabel. After clicking the JButton, reposition the JTextFields and JLabels so that your family members' names appear on the second line, and the family members' relationships to you and ages appear on the third line. Save the program in the Chapter.09 folder on your Student Disk as **JFamilyRecord.java**.
14. Each of the following files in the Chapter.09 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugNine1.java will become FixDebugNine1.java. You can test each applet with the TestDebugNine1.html through TestDebugNine4.html files on your Student Disk.
    a. DebugNine1.java
    b. DebugNine2.java
    c. DebugNine3.java
    d. DebugNine4.java

Case Project

Ray's Appliance Store sells a wide variety of kitchen appliances. Customers often ask for an estimate of the annual cost of running an appliance. Ray typically scribbles his calculations with a pencil on a notepad when he can find one. He has asked you to write a Swing applet that will do the calculations on his computer. Ray wants to be prompted to enter the cost per kilowatt hour of electricity and the estimated number of hours the appliance will run annually. After the figures are entered, the applet should display the estimated annual cost.

CHAPTER

10 GRAPHICS

In this chapter, you will:

- ♦ Learn about the paint() and repaint() methods
- ♦ Use the drawString() method to draw Strings
- ♦ Use the setFont() and setColor() Graphics object methods
- ♦ Create Graphics and Graphics 2D objects
- ♦ Draw lines, rectangles, ovals, arcs, and polygons
- ♦ Copy an area
- ♦ Learn more about fonts and their methods
- ♦ Draw with Java 2D graphics
- ♦ Add sound, images, and simple animations to Swing applets

What are you smiling about?" your mentor, Lynn Greenbrier, asks as she walks by your desk at Event Handlers Incorporated.

"I liked Java programming from the start," you say, "but now that I'm creating applets, I'm really having fun."

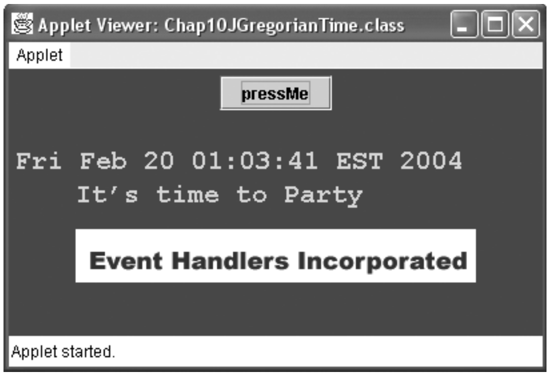
"If you like what you've done with applets so far," Lynn smiles, "just wait until you add colors, shapes, images, and sound. Let me show you how to use graphics and multimedia to add some sizzle."

Previewing the JGregorianTime Swing Applet

The Chap10JGregorianTime Swing applet works as an interactive advertisement for Event Handlers Incorporated and demonstrates several graphics methods. You can now use a completed version of the Swing applet that is saved in the Chapter.10 folder on your Student Disk.

To run the Chap10JGregorianTime Swing applet:

1. Go to the command prompt for the Chapter.10 folder on your Student Disk, type **`appletviewer TestChap10JGregorianTime.html`**, and then press **[Enter]**. It might take a few minutes for the Applet Viewer window shown in Figure 10-1 to open.



```
Fri Feb 20 01:03:41 EST 2004
    It's time to Party
```

Event Handlers Incorporated

Applet started.

Figure 10-1 Chap10JGregorianTime Swing applet

2. Use the Swing applet by clicking the **PressMe** button to see the time display change. Click the **PressMe** button as many times as you like.
3. Click the **Close** button to close the Applet Viewer window.

Learning About the Paint() and Repaint() Methods

In Chapter 9, you learned that every Swing applet uses four methods: init(), start(), stop(), and destroy(). If you don't write these methods, Java provides you with a "do nothing" copy. You can, however, override any of these automatically supplied methods by writing your own versions.

Actually, a fifth method is used within every Swing applet. The **paint() method** runs when Java displays your Swing applet. You can write your own paint() method to override the automatically supplied one whenever you want to paint graphics, such as shapes, on the screen. As with init(), start(), stop(), and destroy() methods, if you don't write a paint() method, you get an automatic version from Java. The paint() method executes automatically every time you minimize, maximize, or resize the Applet Viewer window.

The paint() method header is `public void paint (Graphics g)`. The header indicates that the method requires a Graphics object argument; here it is named g but you can use any legal identifier. However, you don't usually call the paint() method directly. Instead, you call the **repaint() method**, which you can use when a window needs to be updated, such as when it contains new images. The Java system calls the repaint() method when it needs to update a window, or you can call it yourself—in either case, repaint() creates a Graphics object for you. The repaint() method calls another method named update(), which calls the paint() method. The series of events is best described with an example that you will create in the following steps.

To demonstrate how repaint() and paint() operate:

1. Open a new text file in your text editor.
2. Type the following first few lines of a Swing applet named JDemoPaint:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoPaint extends JApplet
 implements ActionListener
{
```

3. The only component in this Swing applet is a JButton that you can create by typing the following code on the next line: `JButton pressButton = new JButton("Press");`.
4. Type the following init() method, which initializes a Container named `con`, sets the `con` layout to FlowLayout, and adds the pressButton to `con`:

```
public void init()
{
 Container con = getContentPane();
 con.setLayout(new FlowLayout() );
 con.add(pressButton);
 pressButton.addActionListener(this);
}
```

5. Override the paint() method by typing the following code, so it prints a message to the screen every time it executes:

```
public void paint(Graphics g)
{
  System.out.println("in paint method");
  pressButton.repaint();
}
```

6. Call the repaint() method when the user clicks the JButton by typing the following:

```
public void actionPerformed(ActionEvent e)
{
 Object source = e.getSource();
 if (source == pressButton)
 {
   repaint();
 }
}
```

7. Add the closing curly brace for the class, and then save the file as **JDemoPaint.java** in the Chapter.10 folder on your Student Disk. Compile the Swing applet using the **`javac JDemoPaint.java`** command.
8. Open a new text file, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JDemoPaint.class"
  WIDTH = 100 HEIGHT = 100>
</APPLET>
</HTML>
```

9. Save the file as **TestJDemoPaint.html** in the Chapter.10 folder on your Student Disk, and then type **`appletviewer TestJDemoPaint.html`** at the command prompt to run the Swing applet using the file. Make sure you can view the command line and the Swing applet on your screen. When the Swing applet starts, the paint() method executes automatically, so the message "in paint method" appears on the command line. Click the **pressButton** in the Swing applet. The actionPerformed() method calls the repaint() method, the repaint() method calls the update() method, which then calls the paint() method, so a second message appears on the command line. Minimize the Applet Viewer window and then restore it. Resize the window by dragging its border. With each action, an "in paint method" message appears on the command line, demonstrating all the conditions under which the paint() method executes.
10. Close the Applet Viewer window.

Tip

The repaint() method only requests that Java repaint the screen. If a second request to repaint() occurs before Java can carry out the first request, then Java executes only the last repaint() method.

USING THE DRAWSTRING() METHOD TO DRAW STRINGS

The **drawString() method** allows you to draw a String in a Swing applet window. The drawString() method requires three arguments: a String, an x-axis coordinate, and a y-axis coordinate.

You are already familiar with x- and y-axis coordinates because you used them with the setLocation() method for components in Chapter 9. However, there is a minor difference in how you place components using the setLocation() method and how you place Strings using the drawString() method. When you use x- and y-coordinates with components, such as JLabels, the upper-left corner of the component is placed at the coordinate position. When you use x- and y-coordinates with drawString(), the lower-left corner of the String appears at the coordinates. Figure 10-2 shows the positions of a JLabel placed at the coordinates 30, 10 and a String placed at the coordinates 10, 30.

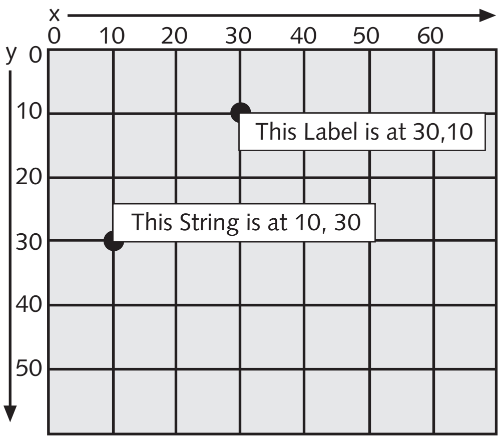


Figure 10-2 JLabel and String coordinates

The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it. Recall that the paint() method header shows that the method receives a Graphics object from the update() method. If you use drawString() within paint(), then the Graphics object you name in the header is available to you. For example, if you write a paint() method with the header `public void paint(Graphics brush)`, then you can draw a String within your paint() method by using a statement such as `brush.drawString("Hi",50,80);`.

To use drawString() to place a String within a Swing applet:

1. Open a new text file, and begin a class definition for a JDemoGraphics class by typing the following:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoGraphics extends JApplet
{
```

2. Declare a String to hold the company name for Event Handlers Incorporated by typing `String companyName = new String("Event Handlers Incorporated");`.

3. Type the following paint() method that uses a Graphics object to draw the companyName String:

```
 public void paint(Graphics gr)
 {
 gr.drawString(companyName,10,100);
 }
}
```

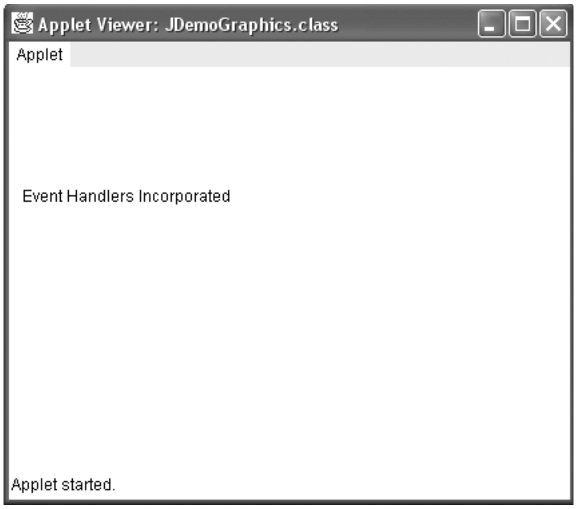
4. Save the file as **JDemoGraphics.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the **`javac`** command.

5. Open a new text file, and then create the following HTML document for the JDemoGraphics1 class:

```
<HTML>
<APPLET CODE =
 "JDemoGraphics.class" WIDTH = 420 HEIGHT = 300>
</APPLET>
</HTML>
```

6. Save the file as **TestJDemoGraphics.html** in the Chapter.10 folder on your Student Disk, and then use the **`appletviewer TestJDemoGraphics.html`** command to run the program. The program's output appears in Figure 10-3.

7. Close the Applet Viewer window.



```
Event Handlers Incorporated
```

```
Applet started.
```

Figure 10-3 Swing applet using the drawString() method

USING THE SETFONT() AND SETCOLOR() GRAPHICS OBJECT METHODS

You can improve the appearance of Graphics objects by using the setFont() and setColor() Graphics object methods. The setFont() method requires a Font object, which, as you recall, you create with a statement such as `Font someFont = new Font("TimesRoman", Font.BOLD, 16);`. Then you can instruct a Graphics object to use the font by using the font as the argument in a setFont() method. For example, if a Graphics object is named brush, then the font is set to someFont with `brush.setFont(someFont);`.

Tip

You learned about the Font object when you changed a JLabel's font in Chapter 9.

You can designate a Graphics color with the setColor() method. The Color class contains 13 constants that appear in Table 10-1. You can use any of these constants as an argument to the setColor() method. For example, you can instruct a Graphics object named brush to apply green paint by using the statement `brush.setColor(Color.green);`. Until you change the color, subsequent graphics output will appear as green.

Tip

Java constants are usually written in all uppercase letters, as you learned in Chapter 4. However, even though the color names of the Color class are constants, Java's creators failed to make them uppercase.

Table 10-1 Color class constants

black	green	red
blue	lightGray	white
cyan	magenta	yellow
darkGray	orange	
gray	pink	

Next you will use your knowledge of fonts and colors to set the color and font style of a Swing applet.

To add a Font and color to your JDemoGraphics class:

1. Open the JDemoGraphics.java text file in your text editor and rename the class **JDemoGraphics2**.
2. Just after the companyName declaration, add a Font object by typing:

```
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
```

3. For the first two statements in the paint() method after the opening curly brace, type the following statements so the gr object uses the bigFont object and the color magenta:

   ```
   gr.setFont(bigFont);
   gr.setColor(Color.magenta);
   ```

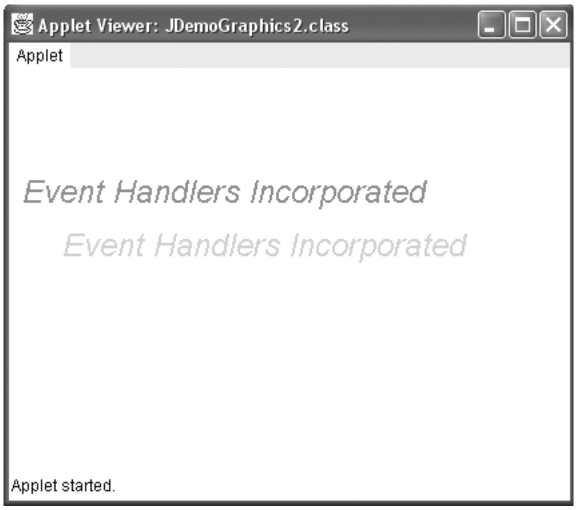
4. Following the existing drawString() method call, type the following lines to change the color and add an additional call to the drawString() method:

   ```
   gr.setColor(Color.orange);
   gr.drawString(companyName,40,140);
   ```

5. Save the file as **JDemoGraphics2.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the **`javac`** command. Modify the **TestJDemoGraphics.html** document for use with the JDemoGraphics2, and then save as **TestJDemoGraphics2.html**. Use the **`appletviewer TestJDemoGraphics2.html`** command to run the program. The program's output appears in Figure 10-4. Although the figure is shown in black and white in this book, notice that the Strings on your screen print as magenta and orange text.

Help: The fonts that appear in your Swing applet might be different, depending on your computer's installed fonts. You will learn about installed fonts later in this chapter.

6. Close the Applet Viewer window.



Event Handlers Incorporated

Event Handlers Incorporated

Applet started.

Figure 10-4 JDemoGraphics2 Swing applet using font and color

You can also create your own Color object with the statement `Color someColor = new Color(r, g, b);`, where r, g, and b are numbers representing the intensity of red, green, and blue you want in your color. The numbers can range from 0 to 255, with 0 being the darkest shade of the color and 255 being the lightest. For example, `color darkPurple = new Color(100, 0, 100);` produces a dark purple color that has red and blue components, but no green. You can create more than 16 million custom colors using this approach.

Tip

Some computers cannot display each of the 16 million possible colors. Each computer will display the closest color it can.

You can discover the red, green, or blue components of any existing color with the methods getRed(), getGreen(), and getBlue(). Each of these methods returns an integer. For example, you can discover the amount of red in a magenta color by printing the value of `Color.magenta.getRed();`.

Next you will use the methods for getting and setting colors to display several hundred colors in a Swing applet.

To create a demonstration program that displays several hundred colors:

1. Open a new text file in your text editor.
2. Type the following import statements, a class header for a JDemoColor program, and the opening curly brace:

   ```
   import javax.swing.*;
   import java.awt.*;
   public class JDemoColor extends JApplet
   {
   ```
3. Define a small font by typing:

   ```
   Font littleFont = new Font("Helvetica", Font.ITALIC, 6);
   ```
4. Add the following paint() method with five integer variables—r, g, b, x, and y.

   ```
   public void paint(Graphics gr)
   {
    int r, g , b;
    int x = 0, y = 0;
   ```
5. Set the Graphics object font by typing **`gr.setFont(littleFont);`**.
6. Create a `for` loop in which the red component will vary from 255 down to 0 in decrements of 20. Within the red `for` loop, vary the intensity of green, and within the green `for` loop, vary the intensity of blue. Although you won't get every possible combination of components, you will get a wide variety.

   ```
   for(r = 255; r >= 0; r -= 20)
    for(g = 255; g >= 0; g -= 20)
     for(b = 255; b >= 0; b -= 20)
         {
   ```

7. Within the body of the innermost `for` loop, create a new color, set the color, and draw an X. After the X is drawn, you increase the x-axis coordinate by 5. When the value of x approaches the horizontal limit of the Swing applet—that is, when it passes 400 or so—increase y and reset x to 0. To accomplish this processing, type the following code:

```
   Color variety = new Color(r, g, b);
   gr.setColor(variety);
   gr.drawString("X",x,y);
   x += 5;
   if (x >= 400)
   {
    x = 0;
    y += 10;
   }//end if
  }//end for
 }//end paint()
}//end JDemoColor class
```

8. Save the file as **JDemoColor.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the **`javac`** command. Modify the **TestJDemoGraphics2.html** document for use with the JDemoColor class, and then save as **TestJDemoColor.html**. When you run the Swing applet, you should see it filled with hundreds of small Xs in many different colors.

9. Close the Applet Viewer window.

The Swing Applet's Background Color

In addition to changing the color of Strings that you display, you can change the background color of your Swing applet. For example, the statement `setBackground(Color.pink);` changes the Swing applet screen color to pink. You do not need a Graphics object to change the Swing applet's background color; it is the Swing applet itself that changes colors. (You could also write `this.setBackground(Color.pink);` because setBackgound refers to "this" Swing applet.)

Creating Graphics and Graphics 2D Objects

When you call the paint() method, you can use the automatically created Graphics object, but you also can instantiate your own Graphics or Graphics 2D objects. For example, you might want to use a Graphics object when some action occurs, such as a mouse event. Because the ActionPerformed() method does not supply you with a Graphics object automatically, you can create your own.

For example, to display a string when the user clicks a JButton, you can code an ActionPerformed() method such as the following:

```
public void actionPerformed(ActionEvent e)
{
 Object source = e.getSource();
 if (source == button1)
 {
  Graphics draw = getGraphics();
  draw.drawString("You clicked the button!",50,100);
 }
}
```

This method instantiates a Graphics object named draw. (You can use any legal Java identifier.) The getGraphics() method provides the draw object with Graphics capabilities. Then the draw object can employ any of the Graphics methods you have learned—setFont(), setColor(), and drawString().

Tip

Notice that when you create the draw object, you are not calling the Graphics constructor directly. (The name of the graphics constructor is Graphics(), not getGraphics().) You are not allowed to call the Graphics constructor because Graphics() is an abstract class. You will learn about abstract classes in Chapter 11.

Next you will create a Graphics object named pen and use the object to draw a String on the screen. The text of the String will appear to move each time a JButton is clicked.

To write a Swing applet in which you create your own Graphics object:

1. Open a new text file in your text editor, and type the following import statements for the Swing applet:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

2. Start typing the following Swing applet that uses the mouse, and defines a String, a JButton, a Font, and two integers to hold x and y coordinates:

```
public class JDemoCreateGraphicsObject extends JApplet
 implements ActionListener
{
 String companyName = new String
  ("Event Handlers Incorporated");
 JButton moveButton = new JButton("Move It");
 Font hell2Font = new Font("Helvetica", Font.ITALIC, 12);
 int x = 10,y = 50;
```

3. Type the following init() method, which changes the background color, adds the JButton, and prepares the Swing applet to listen for JButton events:

```
public void init()
{
   setBackground(Color.yellow);
   Container con = getContentPane();
   con.setLayout(new FlowLayout() );
   con.add(moveButton);
  moveButton.addActionListener(this);
  }
```

4. Within the actionPerformed() method, you can create a Graphics object and use it to draw the String on the screen. Each time a user clicks the JButton, the x- and y-coordinates both increase so a copy of the company name appears slightly below and to the right of the previous company name. Type the following code to accomplish this processing:

```
     public void actionPerformed(ActionEvent e)
     {
      Object source = e.getSource();
      if (source == moveButton)
      {
       Graphics pen = getGraphics();
       pen.setFont(hell2Font);
       pen.setColor(Color.magenta);
       pen.drawString(companyName, x+=20,y += 20);
      }//end if
    }//end actionPerformed()
  }//end JDemoCreateGraphicsObject class
```

5. Save the file as **JDemoCreateGraphicsObject.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the **javac** command. Modify the**TestDemoGraphics2.html** document for use with the JDemoCreateGraphicsObject class, and save it as **TestJDemoCreateGraphicsObject.html** in the Chapter.10 folder on your Student Disk. Use the **appletviewer TestJDemoGraphicsObject.html** command to run the program. Click the **moveButton** several times to see the String message appear and move on the screen.

6. When you finish clicking the moveButton, close the Applet Viewer window.

If you run JDemoCreateGraphicsObject and click the JButton enough times, the "Event Handlers Incorporated" String appears to march off the bottom of the Swing applet. Every time you click the JButton, the x- and y-coordinates used by drawString() increase. You can prevent this error by checking the screen coordinates' values to see if they exceed the applet's dimensions.

To avoid the error of exceeding the applet viewing area:

1. Open the **JDemoCreateGraphicsObject** file, and change the class name to **JDemoCreateGraphicsObject2**.

2. Position your insertion point to the right of the statement `pen.setColor (Color.magenta);` in the actionPerformed() method, and then press **[Enter]** to start a new line.

3. Because you add 20 to the x variable each time you draw the String within the applet, you can ensure that the String appears only 12 times by preventing the x-coordinate from exceeding a value of 250. Type the following `if` statement to check the x-coordinate value:

   ```
   if(x < 250)
   {
   ```

4. Position your insertion point to the right of the line `pen.drawString (companyName,x+=20,y+=20);`, press **[Enter]**, type the closing curly brace for the `if` statement, and then press **[Enter]** again.
5. On the new line, type the following `else` statement that disables the JButton after the x-coordinate becomes too large:

   ```
   else
    moveButton.setEnabled(false);
   ```

6. Save the file as **JDemoCreateGraphicsObject2.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the `javac` command. Modify the**TestJDemoCreateGraphicsObject.html** document for use with the JDemoCreateGraphicsObject2 class, and save as **TestJDemoCreateGraphicsObject2.html** in the Chapter.10 folder on your Student Disk. Now when you click the moveButton until the company name moves to x-coordinate 250, the JButton is disabled, and the company name no longer violates the applet size limits.
7. Close the Applet Viewer window.

Drawing Lines, Rectangles, Ovals, Arcs, and Polygons

Just as you can draw Strings using a Graphics object and the drawString() method, Java provides you with several methods for drawing a variety of lines and geometric shapes.

Tip

Any line or shape will be drawn in the current color you set with the setColor() method.

You can use the **drawLine() method** to draw a straight line between any two points on the screen. The drawLine() method takes four arguments: the x- and y-coordinates of the line's starting point, and the x- and y-coordinates of the line's ending point. For example, if you create a Graphics object named pen, then `pen.drawLine(10,10,100,200);` draws a straight line that slants down and to the right, from position 10, 10 to position 100, 200, as shown in Figure 10-5. Because you can start at either end when you draw a line, an identical line is created with `pen.drawLine(100,200,10,10);`.

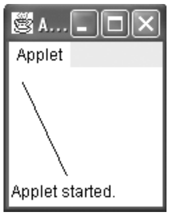


Figure 10-5 Line from position 10, 10 to 100, 200

Tip: It is almost impossible to draw a picture of any complexity without sketching it first on a piece of graph paper to help you determine correct coordinates.

You can use the **drawRect() method** and **fillRect()** method, respectively, to draw the outline of a rectangle or to draw a solid, or filled, rectangle. Each of these methods requires four arguments. The first two arguments represent the x- and y-coordinates of the upper-left corner of the rectangle. The last two arguments represent the width and height of the rectangle. For example, `drawRect(20,100,200,10);` draws a short, wide rectangle that begins at position 20, 100, and is 200 pixels wide by 10 pixels tall.

Tip: For an alternative to the drawRect() method, you can use four calls to drawLine().

The **clearRect() method** also requires four arguments and draws a rectangle. The difference between using the drawRect() and fillRect() methods and the clearRect() method is that the drawRect() and fillRect() methods use the current drawing color, whereas the clearRect() method uses the current background color to draw what appears to be an empty or "clear" rectangle. For example, the JDemoRectangles program shown in Figure 10-6 produces the Swing applet shown in Figure 10-7. The program sets the background color, draws a filled rectangle in a contrasting color, and draws a smaller, clear rectangle (using the background color) within the boundaries of the filled rectangle.

```
import javax.swing.*;
import java.awt.*;
public class JDemoRectangles extends JApplet
{
   public void paint(Graphics gr)
   {
      gr.setColor(Color.red);
      setBackground(Color.blue);
      gr.fillRect(20,20,120,120);
      gr.clearRect(40,40,50,50);
      invalidate();
      validate();
   }
}
```

Figure 10-6 JDemoRectangles program

You can create rectangles with rounded corners when you use the **drawRoundRect method**. The drawRoundRect() method requires six arguments. The first four arguments match the four arguments required to draw a rectangle: the x- and y-coordinates of the upper-left corner, and the width and height. The two additional arguments represent the arc width and height associated with the rounded corners; an **arc** is simply a portion of a circle. If you assign zeros to the arc coordinates, the rectangle will not be rounded; instead, the corners will be square. At the other extreme, if you assign values to the arc coordinates that are at least the width and height of the rectangle, the rectangle is so rounded that it is a circle. The paint() method in Figure 10-8 draws four rectangles with increasingly large corner arcs. Figure 10-9 shows the program's output.

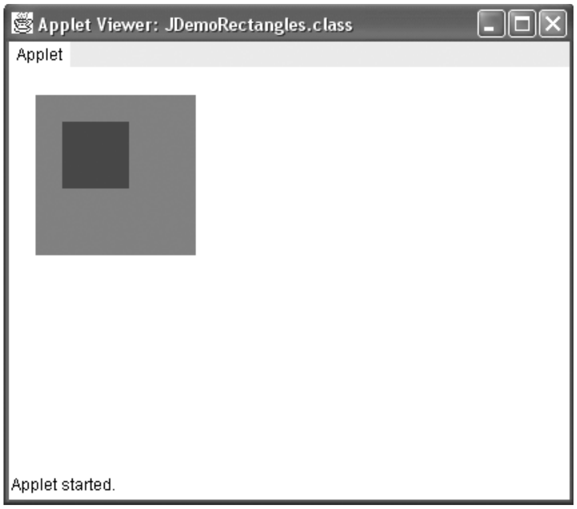


Figure 10-7 Output of the JDemoRectangles program

```
import javax.swing.*;
import java.awt.*;
public class JDemoRectangles2 extends JApplet
{
   public void paint(Graphics gr)
   {
      gr.drawRoundRect(20,20,80,80,0,0);
      gr.drawRoundRect(120,20,80,80,10,10);
      gr.drawRoundRect(220,20,80,80,40,40);
      gr.drawRoundRect(320,20,80,80,80,80);
   }
}
```

Figure 10-8 JDemoRectangles2 program that draws rounded rectangles

Like with the fillRect() method, you can use the fillRoundRect() method to create a filled, rounded rectangle.

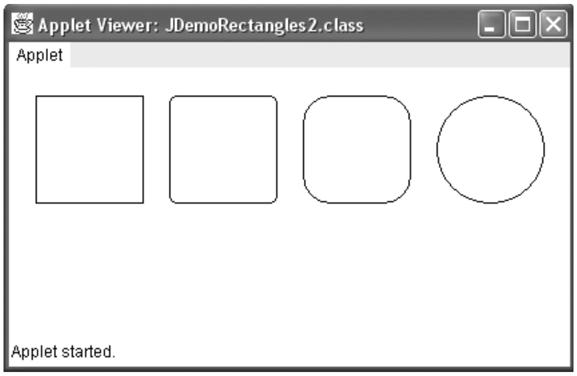


Figure 10-9 Output of the JDemoRectangles2 program that draws rounded rectangles

Drawing Ovals

It is possible to draw an oval using the drawRoundRect() or fillRoundRect() methods, but it is usually easier to use the **drawOval()** and **fillOval() methods**. The drawOval() and fillOval() methods both draw ovals using the same four arguments that rectangles use. When you supply drawOval() or fillOval() with x- and y-coordinates for the upper-left corner and width and height measurements, you can picture an imaginary rectangle that uses the four arguments. The oval is then placed within the rectangle so it touches the rectangle at the center of each of the rectangle's sides. For example, if you create a Graphics object named tool and draw a rectangle with `tool.drawRect(50,50,100,60);`, and an oval with `tool.drawOval(50,50,100,60);`, then the output will appear as shown in Figure 10-10 with the oval edges just skimming the rectangle's sides.

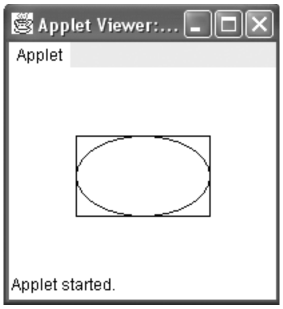


Figure 10-10 Demonstration of the drawOval() method

Tip: If you draw a rectangle with identical height and width, you draw a square. If you draw an oval with identical height and width, you draw a circle.

Next you will add a simple line drawing to the JDemoCreateGraphicsObject2 program. The drawing will appear after the user clicks the JButton enough times to increase the x-coordinate to 250, which disables the JButton.

To add a line drawing to a program:

1. Open the **JDemoCreateGraphicsObject2** program, and rename the class **JDemoCreateGraphicsObject3**.
2. Position your insertion point to the right of the opening curly brace for the actionPerformed() method, and then press **[Enter]** to start a new line.
3. Type the following to define a Graphics object and set its font and color:

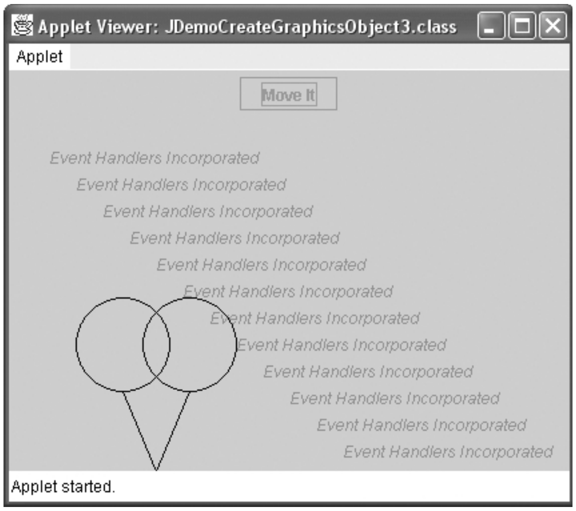
```
Graphics pen = getGraphics();
pen.setFont(hell2Font);
pen.setColor(Color.magenta);
```

4. Replace the current `if...else` structure with the following code that tests the value of x, and either draws the company name or disables the JButton and draws a logo. When you draw the logo, you set the drawing color to black and draw a simple drawing of the Event Handlers Incorporated logo, which is two overlapping balloons with strings attached:

```
if(x < 250)
{
 pen.drawString(companyName,x += 20,y += 20);
}
else
{
 moveButton.setEnabled(false);
 pen.setColor(Color.black);
 pen.drawOval(50,170,70,70);
 pen.drawLine(85,240,110,300);
```

```
   pen.drawOval(100,170,70,70);
   pen.drawLine(135,240,110,300);
  }
```

5. Save the file as **JDemoCreateGraphicsObject3.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the **`javac`** command. Modify the**TestJDemoCreateGraphicsObject2.html** document for use with the JDemoCreateGraphicsObject3 class, and save as **TestJDemoCreateGraphicsObject3.html** in the Chapter.10 folder on your Student Disk. After the company name moves to x-coordinate 250, the JButton is disabled, and the balloon drawing appears, as shown in Figure 10-11.
6. Close the Applet Viewer window.



```
Event Handlers Incorporated
 Event Handlers Incorporated
  Event Handlers Incorporated
   Event Handlers Incorporated
    Event Handlers Incorporated
     Event Handlers Incorporated
      Event Handlers Incorporated
       Event Handlers Incorporated
        Event Handlers Incorporated
         Event Handlers Incorporated
          Event Handlers Incorporated
           Event Handlers Incorporated
Applet started.
```

Figure 10-11 Output of the JDemoCreateGraphicObjects3 program with the JButton disabled

Drawing Arcs

In Java, you can draw an arc using the Graphics **drawArc()** method. To use the drawArc() method, you provide six arguments:

- The x-coordinate of the upper-left corner of an imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The y-coordinate of the same point
- The width of the imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The height of the same imaginary rectangle

- The beginning arc position
- The arc angle

Arc positions and angles are measured in degrees; there are 360 degrees in a circle. The zero-degree position for any arc is at the three o'clock position, as shown in Figure 10-12. The other 359-degree positions increase as you move counterclockwise around an imaginary circle, so that 90 degrees is at the top of the circle in the 12 o'clock position, 180 degrees is opposite the starting position at nine o'clock, and 270 degrees is at the bottom of the circle in the six o'clock position.

The arc angle is the number of degrees over which you want to draw the arc, traveling counterclockwise from the starting position. For example, you could draw a half circle by indicating an arc angle of 180 degrees, or a quarter circle by indicating an arc angle of 90 degrees. If you want to travel clockwise from the starting position, you express the degrees as a negative number. Just as when you draw a line, when drawing any arc you can take one of two approaches: either start at point A and travel to point B, or start at point B and travel to point A. For example, to create an arc object halfarc that looks like the top half of a circle, the statements `halfarc.drawArc(x,y,w,h,0,180);` and `halfarc.drawArc(x,y,w,h,180,-180);` produce identical results. The first statement starts an arc at the three o'clock position and travels 180 degrees counterclockwise to the nine o'clock position. The second statement starts at nine o'clock and travels clockwise to three o'clock.

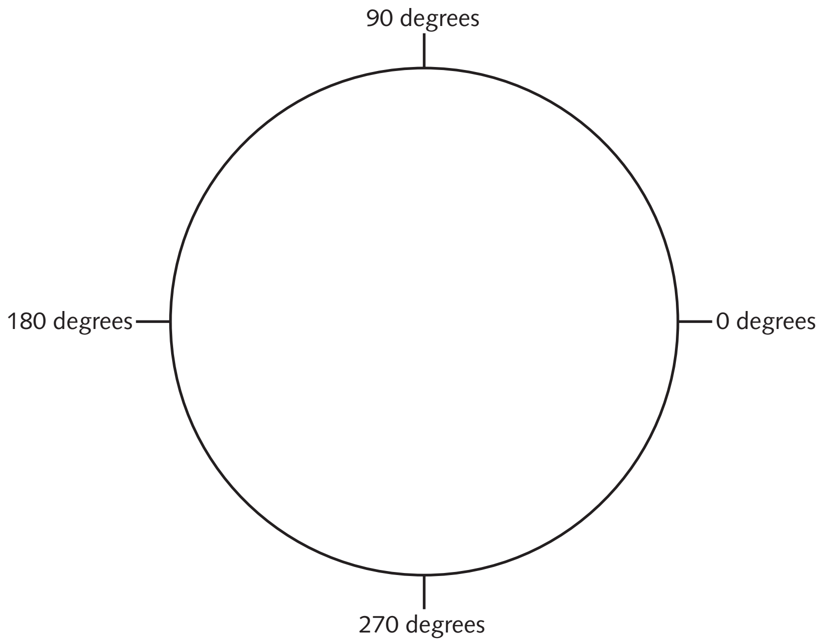


Figure 10-12 Arc positions

The **fillArc() method** creates a solid arc. The arc is drawn and two straight lines are drawn from the arc end points to the center of the imaginary circle whose perimeter the arc occupies. For example, the two statements `solidarc.fillArc(10,50,100,100,20,320);` and `solidarc.fillArc(200,50,100,100,340,40);` together produce the output shown in Figure 10-13. Each of the two arcs is in a circle of size 100 by 100. The first almost completes a full circle, starting at position 20 (near two o'clock) and ending 320 degrees around the circle (at position 340, near four o'clock). The second filled arc more closely resembles a pie slice, starting at position 340 and extending 40 degrees to end at position 40.

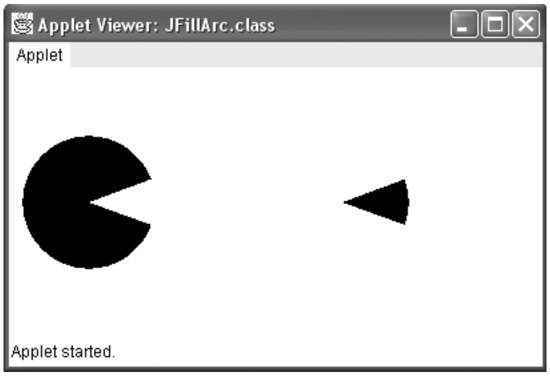


Figure 10-13 Two filled arcs

Creating Three-Dimensional Rectangles

The draw3DRect() method is a minor variation on the drawRect() method. You use the **draw3DRect() method** to draw a rectangle that appears to have "shadowing" on two of its edges—the effect is that of a rectangle that is slightly raised or slightly lowered. The draw3DRect() method requires a fifth argument in addition to the x- and y-coordinates and width and height required by the drawRect() method. The fifth argument is a Boolean value, which is `true` if you want the raised rectangle effect (darker on the right and bottom) and `false` if you want the lowered rectangle effect (darker on the left and top). There is also a **fill3DRect() method** for creating filled three-dimensional rectangles.

Tip: The three-dimensional methods work well only with lighter drawing colors.

Creating Polygons

When you want to create a shape that is more complex than a rectangle, you can use a sequence of calls to the drawLine() method, or you can use the **drawPolygon() method** to draw complex shapes. The drawPolygon() method requires three arguments: two integer arrays and a single integer.

The first integer array holds a series of x-coordinate positions, and the second array holds a series of corresponding y-coordinate positions. The third integer argument is the number of pairs of points you want to connect. If you don't want to connect all the points represented by the array values, you can assign this third argument integer a value that is smaller than the number of elements in each array. However, an error occurs if the third argument is a value higher than the available number of coordinate pairs.

For example, examine the code shown in Figure 10-14, which is a Swing applet that has one task—to draw a red, star-shaped polygon. Two parallel arrays are assigned x- and y-coordinates; the paint() method sets the drawing color to red and draws the polygon. The program's output appears in Figure 10-15.

```
import javax.swing.*;
import java.awt.*;
public class JStar extends JApplet
{
   int xPoints[] = {42, 52, 72, 52,  60, 40,  15, 28,  9, 32, 42};
   int yPoints[] = {38, 62, 68, 80, 105, 85, 102, 75, 58, 60, 38};
   public void paint(Graphics gr)
   {
      gr.setColor(Color.red);
      gr.drawPolygon(xPoints, yPoints, xPoints.length);
   }
}
```

Figure 10-14 JStar Swing applet

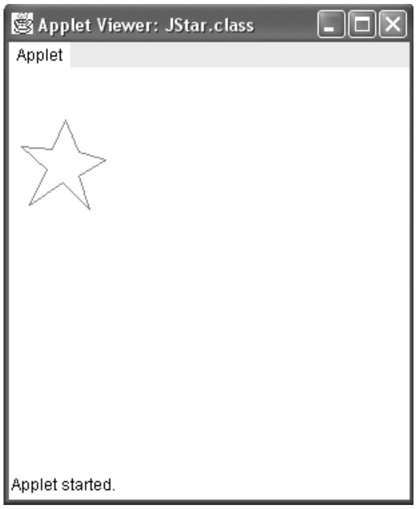


Figure 10-15 Output of the JStar Swing applet

Tip: In Chapter 8, you learned that you can use `length` for the length of an array. Rather than using a constant integer value, such as 11, it is convenient to use the length of one of the coordinate point arrays, as in `xPoints.length`.

You can use the **fillPolygon() method** to draw a solid shape. The major difference between the drawPolygon() and fillPolygon() methods is that if the beginning and ending points used with the fillPolygon() method are not identical, then the two end points will be connected by a straight line before the polygon is filled with color.

The difference is subtle, but rather than providing the fillPolygon() method with three arguments, you can create a Polygon object that defines a polygon, and then pass the constructed object as a single argument to the fillPolygon() method. Note the following statements:

```
Polygon someShape = new Polygon(xPoints, yPoints, size);
fillPolygon(someShape);
```

These statements have the same result as the following:

```
fillPolygon(xPoints, yPoints, size);
```

Additionally, you can instantiate an empty Polygon object (with no points) using the statement `Polygon someFutureShape = new Polygon();`. You use the following statements to add points to the polygon later using a series of calls to the addPoint() method:

```
someFutureShape.addPoint(100,100);
someFutureShape.addPoint(150,200);
someFutureShape.addPoint(50,250);
```

It is practical to use addPoint() instead of coding the point values when you want to write a program in which the user enters polygon point values. Whether the user does so from the keyboard or with a mouse, you can continue to add points to the polygon indefinitely.

COPYING AN AREA

After you create a graphics image, you might want to create copies of the image. For example, you might want a company logo to appear several times in an applet. Of course, you can redraw the picture, but you can also use the **copyArea() method** to copy any rectangular area to a new location. The copyArea() method requires six parameters:

- The x- and y-coordinates of the upper-left corner of the area to be copied
- The width and height of the area to be copied
- The horizontal and vertical displacement of the destination of the copy

Next you will learn how to copy an area containing a logo that you want to appear several times on a Swing applet without re-creating the logo each time.

To copy an image:

1. Open a new text file in your text editor, and then enter the beginning statements for a Swing applet that uses the copyArea() method:

```
import javax.swing.*;
import java.awt.*;
public class JThreeStars extends JApplet
{
```

2. Add the following statements, which will create a polygon in the shape of a star:

```
int xPoints[] = {42, 52, 72, 52,
                60, 40,  15, 28,  9, 32, 42};
int yPoints[] = {38, 62, 68, 80,
                105, 85, 102, 75, 58, 60, 38};
Polygon aStar =
 new Polygon(xPoints, yPoints, xPoints.length);
```

3. Add the following paint() method, which sets a color, draws a star, and then draws two additional identical stars:

```
 public void paint(Graphics star)
 {
  star.setColor(Color.red);
  star.drawPolygon(aStar);
  star.copyArea(0,0,75,105,125,130);
  star.copyArea(0,0,75,105,180,70);
 }
}
```

4. Save the program as **JThreeStars.java** in the Chapter.10 folder on your Student Disk, and then compile the program.

5. Open a new file in your text editor, and then enter the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE = "JThreeStars.class"
  WIDTH = 420 HEIGHT = 300>
</APPLET>
</HTML>
```

6. Save the HTML document as **TestJThreeStars.html** in the Chapter.10 folder on your Student Disk, and then run it using the **`appletviewer TestJThreeStars.html`** command. The output should look like Figure 10-16.

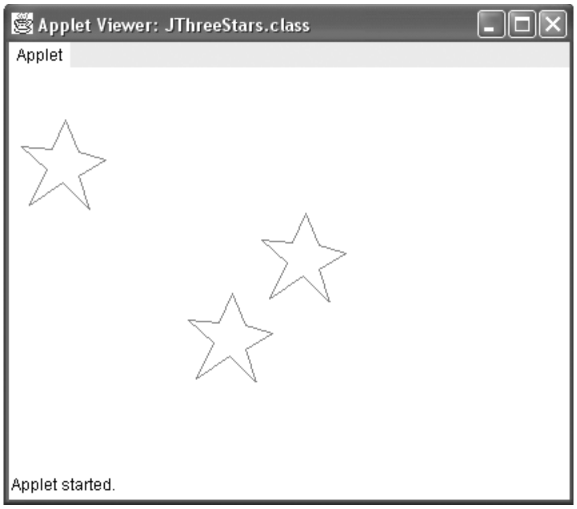


Figure 10-16 Output of the JThreeStars applet

7. Close the Applet Viewer window.
8. Modify the program to add more stars in any location you choose, save and compile the program, and then run the HTML document to confirm that the stars are copied to your desired locations.
9. Close the Applet Viewer window.

Learning More About Fonts and Their Methods

As you add more components in your Swing applet, positioning becomes increasingly important. In particular, when you draw Strings using different fonts, if you do not place the Strings correctly, they overlap and become impossible to read. Additionally, even when you define a font, such as `Font myFont = new Font("TimesRoman",Font.PLAIN,10);`, you have no guarantee that the font will be available on every computer that runs your Swing applet. If your user's computer does not have the font loaded, then Java chooses a default replacement font, so you are never completely sure of how your output will look. Fortunately, Java provides many useful methods for obtaining information about the fonts you use.

You can discover the fonts that are available on your system by using the **getAllFonts() method** that is part of the GraphicsEnvironment class defined in the java.awt package. The GraphicsEnvironment class structure shown in Figure 10-17 describes the collection of GraphicsDevice objects and Font objects available to a Java application on a particular platform. The getAllFonts() method returns an array of String objects that are the names of available fonts, as shown in the following code: `GraphicsEnvironment myFonts = GraphicsEnvironment.getLocalGraphicsEnvironment(); Font myFonts = myFonts.getAllFonts()`.

```
java.lang.Object
  |
  +--java.awt.GraphicsEnvironment
```

Figure 10-17 Structure of the GraphicsEnvironment class

Notice in the example above that you can't call the GraphicsEnvironment object directly. Instead, you must get a reference object to the current computer by calling the static getLocalGraphicsEnvironment() method. In the previous example, myFonts is the reference object. Then in the second statement, the myFonts object is used as the Font reference to the getAllFonts() method. The getAllFonts method returns an array of Font objects existing on the current system.

You can discover the resolution and screen size on your system by using the **getScreenResolution() method** and **getScreenSize() method** that is part of the Toolkit class. The structure of this helpful class is shown in Figure 10-18.

```
java.lang.Object
  |
  +--java.awt.Toolkit
```

Figure 10-18 Structure of the Toolkit class

The **getDefaultToolkit() method** provides information about the system in use. The **getScreenResolution() method** returns the number of pixels as an int type. You can create a Toolkit object and get the screen resolution with the following code:

```
Toolkit tk = Toolkit.getDefaultToolkit();
int resolution = tk.getScreenResolution()
```

The Dimension class structure is shown in Figure 10-19. A Dimension object is useful for representing the width and height of a user interface component. For example, calling the Dimension(int, int) constructor in the following example creates a Dimension object representing the width and height specified as arguments. The Dimension class has three constructors:

- Dimension() method creates an instance of Dimension with a width of zero and a height of zero.
- Dimension(Dimension d) creates an instance of Dimension whose width and height are the same as for the specified dimension.
- Dimension(int width, int height) constructs a Dimension and initializes it to the specified width and specified height.

```
java.lang.Object
  |
  +--java.awt.geom.Dimension2D
        |
        +--java.awt.Dimension
```

Figure 10-19 Structure of the Dimension class

The **getScreenSize() method** that is a member of the Toolkit object returns an object of type Dimension which specifies the width and height of the screen in pixels. Knowing the number of pixels for the width and height of your display is useful to set the coordinates for the position of the window and also set the width and height of the window.

```
Dimension screen = tk.getScreenSize();
String width = screen.width;
String height = sd.height;
```

Next you will write a Swing applet that lists the resolution and screen size along with the fonts available on your system.

To write an applet that lists the resolution, screen size, and fonts on your system:

1. Open a new file in your text editor, and then enter the first few lines of the JFontList Swing applet:

   ```
   import javax.swing.*;
   import java.awt.*;
   public class JFontList extends JApplet
   {
   ```

2. Add the following statement to create two integer variables to hold the x- and y-coordinate positions you will use to draw Strings within the applet: **`int x = 10, y = 15;`**.

3. Add the following paint() method header and an opening curly brace. Within the method, create a Toolkit object by calling the getDefaultToolkit() method. Call the toString() method to return the value stored in a String named resAndSize. Draw the first String named resAndSize at horizontal position 10 and vertical position 15.

   ```
   public void paint(Graphics gr)
    {
     Toolkit tk = Toolkit.getDefaultToolkit();
     String resAndSize = toString();
     gr.drawString(resAndSize,x, y += 15);
   ```

4. Enter the statement that creates the GraphicsEnvironment object named ge. Enter the following `for` loop to the paint() method. This loop will draw each String in the array that was filled using the getAvailableFontFamilyNames() method. You will draw each subsequent String 15 pixels lower within the applet. Finally, type the closing curly brace for the paint() method.

```
GraphicsEnvironment ge =
  GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontnames = ge.getAvailableFontFamilyNames();
for (int i = 0; i < fontnames.length; it=4)
  {
 gr.drawString(fontnames[i], x, y);
 gr.drawString(fontnames[i+1], x+190, y);
 gr.drawString(fontnames[i+2], x+380, y);
 gr.drawString(fontnames[i+3], x+570, y+=15);
  }
```

5. Create a toString() method that constructs and returns information about the screen resolution and size. Within the method, create a Toolkit object named tk and a Dimension object named sd. You can use the methods and fields of these objects to construct a return String containing screen information. Finally add a closing curly brace for the paint() method.

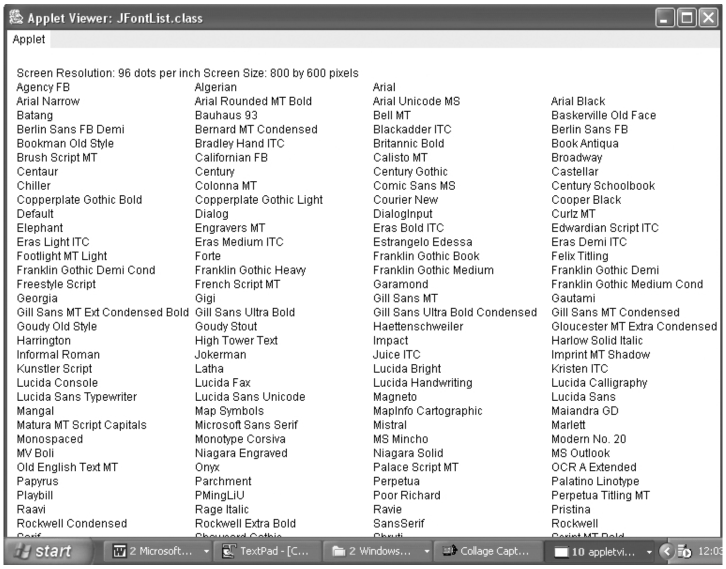
```
 public String toString()
 {
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension sd = tk.getScreenSize();
  return "Screen Resolution: " + tk.getScreenResolution()
          + " dots per inch" +
      " Screen Size: " + sd.width + " by " + sd.height +
      " pixels";
 }
}
```

Tip: In Chapter 7, you first used the automatically included toString() method that converts objects to Strings. Now, you override that method for this class by writing your own version. You will learn more about the toString() method in Chapter 12.

6. Save the file as **JFontList.java** in the Chapter.10 folder on your Student Disk, and then compile it using the **javac** command.
7. Open a new text file, and then enter the following text to create an HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JFontList.class" WIDTH = 760
   HEIGHT = 600>
</APPLET>
</HTML>
```

8. Save the HTML document as **TestJFontList.html** in the Chapter.10 folder on your Student Disk, and then run the program using the **appletviewer** command. Your output should look like Figure 10-20. Notice that the Swing applet is not large enough to display all of the fonts that are installed. (Your font list might be different, depending on the fonts installed on your computer. Your list might even be so long that it cannot fully display in the applet.



```
Screen Resolution: 96 dots per inch Screen Size: 800 by 600 pixels
Agency FB                       Algerian                        Arial
Arial Narrow                    Arial Rounded MT Bold           Arial Unicode MS                Arial Black
Batang                          Bauhaus 93                      Bell MT                         Baskerville Old Face
Berlin Sans FB Demi             Bernard MT Condensed            Blackadder ITC                  Berlin Sans FB
Bookman Old Style               Bradley Hand ITC                Britannic Bold                  Book Antiqua
Brush Script MT                 Californian FB                  Calisto MT                      Broadway
Centaur                         Century                         Century Gothic                  Castellar
Chiller                         Colonna MT                      Comic Sans MS                   Century Schoolbook
Copperplate Gothic Bold         Copperplate Gothic Light        Courier New                     Cooper Black
Default                         Dialog                          DialogInput                     Curlz MT
Elephant                        Engravers MT                    Eras Bold ITC                   Edwardian Script ITC
Eras Light ITC                  Eras Medium ITC                 Estrangelo Edessa               Eras Demi ITC
Footlight MT Light              Forte                           Franklin Gothic Book            Felix Titling
Franklin Gothic Demi Cond       Franklin Gothic Heavy           Franklin Gothic Medium          Franklin Gothic Demi
Freestyle Script                French Script MT                Garamond                        Franklin Gothic Medium Cond
Georgia                         Gigi                            Gill Sans MT                    Gautami
Gill Sans MT Ext Condensed Bold Gill Sans Ultra Bold            Gill Sans Ultra Bold Condensed  Gill Sans MT Condensed
Goudy Old Style                 Goudy Stout                     Haettenschweiler                Gloucester MT Extra Condensed
Harrington                      High Tower Text                 Impact                          Harlow Solid Italic
Informal Roman                  Jokerman                        Juice ITC                       Imprint MT Shadow
Kunstler Script                 Latha                           Lucida Bright                   Kristen ITC
Lucida Console                  Lucida Fax                      Lucida Handwriting              Lucida Calligraphy
Lucida Sans Typewriter          Lucida Sans Unicode             Magneto                         Lucida Sans
Mangal                          Map Symbols                     MapInfo Cartographic            Maiandra GD
Matura MT Script Capitals       Microsoft Sans Serif            Mistral                         Marlett
Monospaced                      Monotype Corsiva                MS Mincho                       Modern No. 20
MV Boli                         Niagara Engraved                Niagara Solid                   MS Outlook
Old English Text MT             Onyx                            Palace Script MT                OCR A Extended
Papyrus                         Parchment                       Perpetua                        Palatino Linotype
Playbill                        PMingLiU                        Poor Richard                    Perpetua Titling MT
Raavi                           Rage Italic                     Ravie                           Pristina
Rockwell Condensed              Rockwell Extra Bold             SansSerif                       Rockwell
```

Figure 10-20 Output of the JFontList program

9. Close the Applet Viewer window.

Typesetters and desktop publishers measure the height of every font in three parts: leading, ascent, and descent. **Leading** is the amount of space between baselines. **Ascent** is the height of an uppercase character from a baseline to the top of the character. **Descent** measures the size of characters that "hang below" the baseline, such as the tails on the lowercase letters g and j. The **height** of a font is the sum of the leading, ascent, and descent. Figure 10-21 shows each of these measurements.

Tip: Leading is pronounced "ledding."

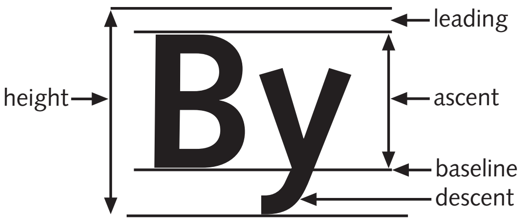


Figure 10-21 Parts of a font's height

You can discover a font's height by using the **getFontMetrics() method**. The getFontMetrics() method is part of the Graphics class and returns a FontMetrics object. The FontMetrics class contains the following methods that return a font's statistics:

- `public int getLeading()`
- `public int getAscent()`
- `public int getDescent()`
- `public int getHeight()`

Each of these methods returns an integer value representing the font size in points (one point measures 1/72 of an inch) of the requested portion of the Font object. For example, if you define a Font object named myFont, and a Graphics object named paintBrush, then you can set the current font for the Graphics object with the statement `paintBrush.setFont(myFont);`. When you code `int heightOfFont = paintBrush.getFontMetrics().getHeight();`, then the heightOfFont variable holds the total height of myFont characters.

Tip: Notice the object-dot-method-dot-method construction of the getHeight() statement. Alternately, if it is clearer to you, you can write two statements. The first statement declares a FontMetrics object: `FontMetrics fmObject = paintBrush.getFontMetrics();`. The second statement assigns a value to heightOfFont: `int heightOfFont = fmObject.getHeight();`.

Tip: When you define a Font object, you use point size. However, when you use the getFontMetrics() methods, the sizes are returned in pixels.

Next you will write a Swing applet to demonstrate the FontMetrics() methods. You will create three Font objects and display their metrics.

To demonstrate the FontMetrics methods:

1. Open a new text file in your text editor, and then enter the first few lines of the JDemoFontMetrics program:

```
import javax.swing.*;
import java.awt.*;
public class JDemoFontMetrics extends JApplet
{
```

2. Type the following code to create a String and a few fonts to use for demonstration purposes:

```
String companyName =
 new String("Event Handlers Incorporated");
Font
 courierItalic = new Font("Courier", Font.ITALIC, 16),
 timesPlain = new Font("TimesRoman", Font.PLAIN, 16),
 helvetBold = new Font("Helvetica", Font.BOLD, 16);
```

Tip: If your JFontList program showed that you do not have one of these fonts, then substitute another font that you do have.

3. Add the following code to define four integer variables to hold the four font measurements, and two integer variables to hold the current horizontal and vertical output positions within the Swing applet:

```
int ascent, descent, height, leading;
int x = 10, y = 15;
```

4. Within the Swing applet, you will draw Strings for output that you will position 15 pixels apart vertically on the screen. Type the following statement to create a constant to hold this vertical increase value:

```
static final int INCREASE = 15;
```

5. Add the following statements to start writing a paint() method. Within the method, you set the Font to courierItalic, draw the companyName String to show a working example of the font, and then call a displayMetrics() method that you will write in Step 6. You will pass the Graphics object to the displayMetrics() method, so the displayMetrics() method can discover the sizes associated with the current font. Perform the same three steps using the timesPlain and helvetBold fonts.

```
public void paint(Graphics pen)
{
 pen.setFont(courierItalic);
 pen.drawString(companyName, x, y);
 displayMetrics(pen);
 pen.setFont(timesPlain);
 pen.drawString(companyName, x, y += 40);
 displayMetrics(pen);
 pen.setFont(helvetBold);
 pen.drawString(companyName, x, y += 40);
 displayMetrics(pen);
}
```

6. Next add the header and opening curly brace for the displayMetrics() method. The method will receive a Graphics object from the paint() method. Add the following statements to call the four getFontMetrics() methods to obtain values for the leading, ascent, descent, and height variables:

```
public void displayMetrics(Graphics metrics)
{
 leading = metrics.getFontMetrics().getLeading();
 ascent = metrics.getFontMetrics().getAscent();
 descent = metrics.getFontMetrics().getDescent();
 height = metrics.getFontMetrics().getHeight();
```

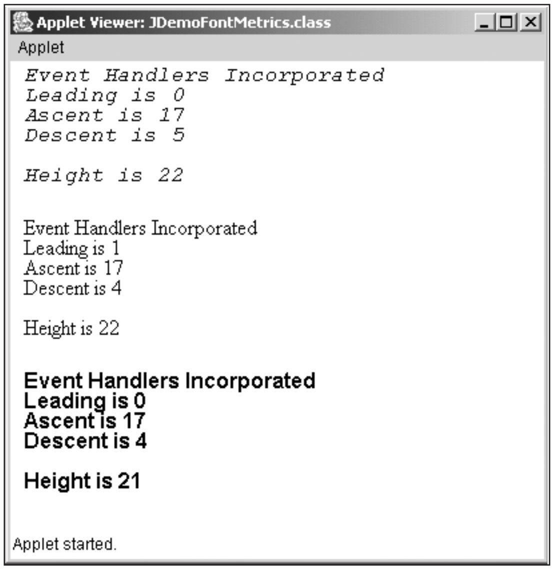
7. Add the following five drawString() statements to display the values. Use the expression `y += INCREASE` to change the vertical position of each String by the INCREASE constant.

```
     metrics.drawString("Leading is " + leading,
        x, y +=  INCREASE);
     metrics.drawString("Ascent is " + ascent,
        x, y +=  INCREASE);
     metrics.drawString("Descent is " + descent,
        x, y +=  INCREASE);
     metrics.drawString("     ", x, y += INCREASE);
     metrics.drawString("Height is " + height,
        x, y +=  INCREASE);
    }
   }
```

8. Save the program as **JDemoFontMetrics.java** in the Chapter.10 folder on your Student Disk, and then compile it using the **`javac`** command.
9. Open a new text file in your text editor, and then enter the following HTML document to host the JDemoFontMetrics Swing applet:

```
<HTML>
<APPLET CODE =
 "JDemoFontMetrics.class" WIDTH = 400 HEIGHT = 350>
</APPLET>
</HTML>
```

10. Save the HTML document as **TestJFontMetrics.html** in the Chapter.10 folder on your Student Disk. At the command prompt, type **`appletviewer TestJFontMetrics.html`**. Your output should look like Figure 10-22. Notice that even though each Font object was constructed with a size of 16, the individual statistics vary for each Font object.



```
Event Handlers Incorporated
Leading is 0
Ascent is 17
Descent is 5

Height is 22


Event Handlers Incorporated
Leading is 1
Ascent is 17
Descent is 4

Height is 22


Event Handlers Incorporated
Leading is 0
Ascent is 17
Descent is 4

Height is 21
```

Figure 10-22 Output of the JDemoFontMetrics Swing applet

11. Close the Applet Viewer window.

A practical use for discovering the height of your font is to space Strings correctly as you print them. For example, instead of placing every String in a series vertically equidistant from the previous String with a statement, such as `pen.drawString("Some string", x, y += INCREASE);` (where INCREASE is always the same), you can make the actual increase in the vertical position dependent on the font. If you code `pen.drawString("Some string", x, y += pen.getFontMetrics().getHeight());`, then you are assured that each String has enough room, and will appear regardless of the font currently in use by the Graphics pen object.

When you create a String, you know how many characters are in the String. However, you cannot be sure which font Java will use or substitute, and because fonts have different measurements, it is difficult to know the exact width of the String in a Swing applet. Fortunately, the FontMetrics class contains a **stringWidth() method** that returns the integer width of a String. As an argument, the stringWidth() method requires the name of a String. For example, if you create a String named myString, then you can retrieve the width of myString with `int width = gr.getFontMetrics().stringWidth(myString);`.

Next you will use the FontMetrics methods to draw a rectangle around a String. Instead of guessing at appropriate pixel positions, you can use the height and width of the String to create a box with borders placed symmetrically around the String.

To draw a rectangle around a String:

1. Open a new file in your text editor, and then enter the first few lines of a JBoxAround Swing applet:

```
import javax.swing.*;
import java.awt.*;
public class JBoxAround extends JApplet
{
```

2. Enter the following statements to add a String, a Font, and variables to hold the font metrics and x- and y-coordinates:

```
String companyName =
 new String("Event Handlers Incorporated");
Font serifItalic = new Font("Serif", Font.ITALIC, 20);
int leading, ascent, height, width;
int x = 40, y = 60;
```

3. Create the following constant variable that holds a number of pixels indicating the dimensions of the rectangle that you will draw around the String:

```
static final int BORDER = 5;
```

4. Add the following paint() method, which sets the font, draws the String, and obtains the font metrics:

```
public void paint(Graphics gr)
{
    gr.setFont(serifItalic);
    gr.drawString(companyName,x,y);
    leading = gr.getFontMetrics().getLeading();
    ascent = gr.getFontMetrics().getAscent();
    height = gr.getFontMetrics().getHeight();
    width = gr.getFontMetrics().stringWidth(companyName);
```

5. Draw a rectangle around the String using the following drawRect() method. In Figure 10-23, the x- and y-coordinates of the upper-left edge are set at 40-border, 60-(ascent + leading + border). The proper width and height are then determined to draw a uniform rectangle around the string.

The values of the x- and y-coordinates used in the drawString() method indicate the left side of the baseline of the first character in the String. You want to position the upper-left corner of the rectangle five pixels to the left of the String, so the first argument to drawRect() is five less than x, or `x - BORDER`. The second argument to drawRect() is the y-coordinate of the String minus the ascent of the String, minus the leading of the String, minus five, or `y - (ascent + leading + BORDER)`. The last two arguments to drawRectangle() are the width and the height of the rectangle. The width is the String's width plus five pixels on the left and five pixels on the right. The height of the rectangle is the String's height, plus five pixels above the String and five pixels below the String.

```
  gr.drawRect(x - BORDER, y - (ascent + leading + BORDER),
  width + 2 * BORDER, height + 2 * BORDER);
  repaint ();
  }
 }
```

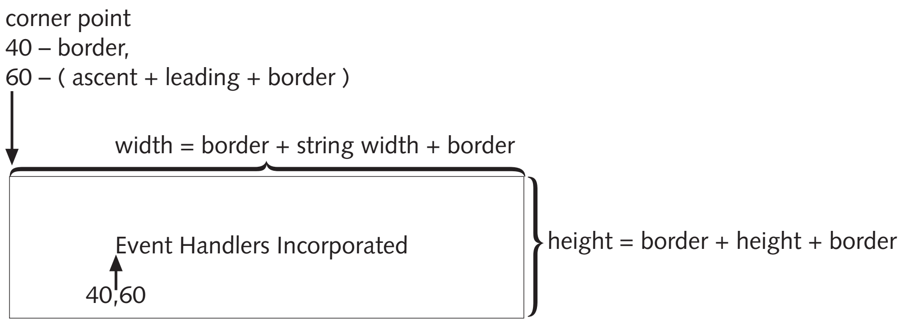

corner point
40 – border,
60 – (ascent + leading + border)
width = border + string width + border
Event Handlers Incorporated
40,60
height = border + height + border

Figure 10-23 Rectangle surrounding a String

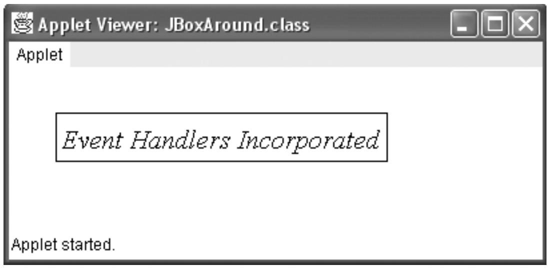
6. Save the file as **JBoxAround** in the Chapter.10 folder on your Student Disk, and then compile it using the **javac** command.

7. Open a new text file in your text editor, and then enter the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "JBoxAround.class" WIDTH = 400 HEIGHT =
120>
</APPLET>
</HTML>
```

8. Save the HTML document as **TestJBoxAround.html** in the Chapter.10 folder on your Student Disk, and then run the program using the **`appletviewer TestJBoxAround.html`** command. Your output should look like Figure 10-24.

9. Close the Applet Viewer window, and then experiment with changing the contents of the String and the size of the BORDER constant. Confirm that the rectangle is drawn symmetrically around any String object. When you finish, close the Applet Viewer window.



```
Event Handlers Incorporated
```

```
Applet started.
```

Figure 10-24 Output of the JBoxAround program

Drawing with Java 2D Graphics

Drawing operations earlier in this chapter are called using an object. In addition, you can call drawing operations on a Graphics2D object. The structure of the Graphics2D class is shown in Figure 10-25.

```
java.lang.Object
  |
  +--java.awt.Graphics
        |
        +--java.awt.Graphics2D
```

Figure 10-25 Structure of the Graphics2D class

The advantage of using Java 2D is the enhanced classes offered to create higher-quality two-dimensional (2D) graphics, images, and text for use in your programs. They don't replace the existing java.awt classes, though—you can still use the other classes and programs that use them.

One of the advantages of Java 2D is a set of high-quality classes to offer enhanced 2D graphics, images, and text. Some of these classes include features such as:

- Fill patterns such as gradients
- Strokes that define the width and style of a drawing stroke
- Anti-aliasing, a graphics technique for producing smoother on-screen graphics

Graphics2D is found in the java.awt package. A Graphics2D object is produced by casting a Graphics object and is commonly referred to as a graphics context. For example, the void paint()method of the previous JBoxAround applet could be cast to create a Graphics2D object as follows:

```
public void paint(Graphics pen)
{
 Graphics2D newpen = (Graphics2D)pen;
```

The JBoxAround2 program created by casting is shown in Figure 10-26. It produces identical output to the JBoxAround program, as shown earlier in Figure 10-24.

```
import javax.swing.*;
import java.awt.*;
public class JBoxAround2 extends JApplet
{
   String companyName =
     new String("Event Handlers Incorporated");
   Font serifItalic = new Font("Serif". Font.ITALIC. 20);
   int leading, ascent, height, width;
   int x = 40, y = 60;

   static final int BORDER = 5

   public void paint(Graphics pen)
   {
     Graphics2D two = (Graphics2D)pen;
     two.setFont(serifItalic);
     two.drawString(companyName,x,y);
     leading = two.getFontMetrics().getLeading();
     ascent = two.getFontMetrics().getAscent();
     height = two.getFontMetrics().getHeight();
     width = two.getFontMetrics().stringWidth(companyName);
     two.drawRect(x - BORDER, y - (ascent + leading + BORDER),
       width + 2 = BORDER, height +2 = BORDER);
     repaint();
   }
}
```

Figure 10-26 JBoxAround2 program

One concept introduced with Java 2D distinguishes between an output device's coordinate space and the coordinate space you refer to when drawing an object. **Coordinate space** is any 2D area that can be described using x- and y-coordinates. For all drawing operations so far in this chapter, the only coordinate space used is for the **device coordinate space**. Recall that you have specified the x- and y-coordinates for an output area such as a container on a Swing applet, and those coordinates have been used to draw lines, text, and other objects. Java 2D adds a **user coordinate space** that you refer to when creating and drawing a 2D drawing object. The upper-left corner 0,0 of the drawing area represents both the device space and the user space. Whereas the device space coordinate is constant, the user space coordinate (0,0) can move as a result of a 2D drawing such as a 2D rotation operation. You will learn more about the two coordinate systems as you work with the 2D examples in this chapter.

You can think of the process of drawing with Java 2D objects as involving:

- Specifying the rendering attributes
- Setting a drawing stroke
- Creating objects to draw

Specifying the Rendering Attributes

The first step in drawing a 2D object is to specify how a drawn object will be rendered. Whereas drawings that are not 2D can only specify the attribute Color, 2D can designate other attributes such as line width and fill patterns. You specify 2D colors by using the setColor() method, which works like the Graphics method of the same name. Using a Graphics2D object, the color can be set to black using the code:

```
gr2D.setColor(Color.black);
```

Fill patterns control how a drawing object will be filled in. In addition to using a solid color, 2D fill patterns can be a gradient fill, texture, or even a pattern that you devise. A fill pattern is created by using the setPaint() method of Graphics2D with a bPaint object as the only argument. Classes that can be a fill pattern include GradientPaint, TexturePaint, and Color.

A **gradient fill** is a gradual shift from one color at one coordinate point to a different color at a second coordinate point. If the color shift occurs once between the points, it is called an **acyclic gradient**. If the shift occurs repeatedly, it is called a **cyclic gradient**. Figure 10-28 shows the top rectangle with an acyclic shift, and the bottom rectangle with a cyclic shift between white and darkGray colors.

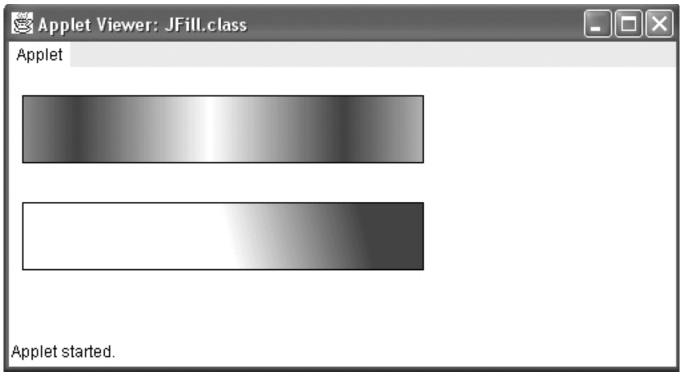


Figure 10-27 One rectangle has an acyclic gradient; the other has a cyclic gradient

Setting a Drawing Stroke

All lines in non-2D graphics operations are drawn solid, with square ends and a line width of 1 pixel. With the new 2D methods, the drawing line width can be changed using the setStroke() method. The Stroke is actually an interface; the class that defines line types and implements the Stroke interface is named **BasicStroke**. A BasicStroke constructor takes three arguments:

- A `float` value representing the line width, with 1.0 as the norm
- An `int` value determining the type of cap decoration at the end of a line
- An `int` value determining the style of juncture between two line segments

BasicStroke class variables determine the endcap and juncture style arguments. **Endcap** styles apply to the end of lines that do not join with other lines, and include CAP_BUT, CAP_ROUND, and CAP_SQUARE. Juncture styles, for lines that join, include JOIN_MITER, JOIN_ROUND, and JOIN_BEVEL.

Tip

Technically, the term stroke has been defined as a single movement using or as if using, a tool or implement such as a pen or pencil.

The following statements create a BasicStroke object and make it the current stroke:

```
BasicStroke aLine = new BasicStroke(1.0f,
 BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
gr2D.setStroke(aLine);
```

The `f` indicates that the 1.0 argument is a floating-point type.

Creating Objects to Draw

After you have created a Graphics2D object and specified the rendering attributes, the final steps are to create the different draw objects and then draw them. Objects that are drawn in Java 2D are first created by defining them as geometric shapes using the java.awt.geom package classes. You can define the shape of lines, rectangles, ovals, and arcs; after you define the shape, you use it as an argument to the draw() or fill() methods. The Graphics2D class does not have different methods for each shape you can draw.

Lines

Lines are created using the Line2D.Float class that takes four arguments. The arguments are the x- and y-coordinates of the two endpoints of the line. For example, to create a line from the endpoint (60,5) to the endpoint (13,28), the arguments are:

```
Line2D.Float line = new Line2D.Float(60F, 5F, 13F, 28F);
```

Tip

Note that F or f is used with the literal arguments so that the Java compiler will not mistake them for integers.

It is possible to create lines based on x and y points defined in the Point2D class. The **Point.2D.Float class** defines a point from a pair of x- and y-coordinates of type `float`. In the following example, you replace the x- and y-coordinates with Point2D objects:

```
Point2D.Float pos1 = new Point2D.Float(60,5);
Point2D.Float pos2 = new Point2D.Float(13,28);
```

The code to create a line then becomes `Line2D.Float line = new Line2D.Float (pos1, pos2);`.

Next you will create a line with a drawing stroke to illustrate how a drawing stroke can be created with different end types and juncture types where lines intersect.

To create a line with a drawing stroke:

1. Open a new file in your text editor, and then enter the first few lines of a J2DLine Swing applet. (Note that you are importing the java.awt.geom package.)

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class J2DLine extends JApplet
{
```

2. Enter the following statements to create a paint() method, create a Graphics environment gr, and cast the Graphics environment to a Graphics2D environment gr2D. Create x and y points with the Point2D.Float class.

```
public void paint(Graphics gr)
{
 Graphics2D gr2D = (Graphics2D)gr;
 Point2D.Float pos1 = new Point.2D.Float(50,10);
 Point2D.Float pos2 = new Point2D.Float(13,28);
```

3. Create a BasicStroke object, and then create a drawing stroke named aLine. Note that the line width is set to 5 pixels and the endcap style and juncture style are set to CAP_ROUND.

```
BasicStroke aLine = new BasicStroke(5.0f,
BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
```

4. Add the following code to create a line between the points posx and posy, and draw the line:

```
  gr2D.setStroke(aline);
  Line2D.Float line = new Line2D.Float(pos1, pos2);
  gr2D.draw(line);
  repaint();
  }
}
```

5. Save the file as **J2DLine.java** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.

6. Open a new file in your text editor, and then enter the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "J2DLine.class" WIDTH = 50 HEIGHT = 50>
</APPLET>
</HTML>
```

7. Save the HTML document as **TestJ2DLine.html** in the Chapter.10 folder on your Student Disk, and then run the program using the **`appletviewer TestJ2DLine.html`** command. Your output should look like Figure 10-28.

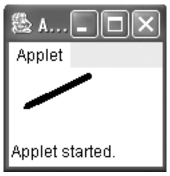


Figure 10-28 Output of the J2DLine program

Rectangles

Rectangles can be created by using a Rectangle2D.Float or a Rectangle2D.Double class. The two classes are distinguished by the type of arguments used in the constructor—float or double. Rectangle2D.Float and Rectangle2D.Double both require four arguments representing the x-coordinate, y-coordinate, width, and height. The code to create a Rectangle2D.Float object named rect at (10,10) with a width of 50 and height of 40 is `Rectangle2D. Float rect = new Rectangle2D.Float(10F, 10F, 50F, 40F);`.

Ovals

Oval objects can be created with the Ellipse2D.Float class. The Ellipse2D.Float constructor requires four arguments representing the x-coordinate, y-coordinate, width, and height. The code to create an Ellipse2D.Float object named ell at (10,73) with a width of 40 and height of 20 is `Ellipse2D.Float ell = new Ellipse2D.Float(10F,73F,40F,20F);`.

Arcs

Arcs can be created with the Arc2D.Float class. The Arc2D.Float constructor takes seven arguments. The first four are arguments representing the x-coordinate, y-coordinate, width, and height that apply to the ellipse of which the arc is a part. The remaining three arguments are:

- The starting degree of the arc
- The number of degrees it travels
- An integer indicating how it is closed

The number of degrees traveled by the arc is specified in a counterclockwise direction using positive numbers. The last argument uses one of the three class variables:

- Arc2D.PIE connects the arc to the center of an ellipse and looks like a pie slice.
- Arc2D.CHORD connects the arc's endpoints with a straight line.
- Arc2D.OPEN is an unclosed arc.

To create an Arc2D.Float object named ac at (10,133) with a width of 30 and height of 33, a starting degree of 30, 120 degrees traveled, and using the class variable Arc.2D.PIE, you use the following statment: `Arc2D.Float ac = new Arc2D.Float(10,133,30,33, 30,120,Arc2D.PIE);`.

Polygons

A Polygon is created by defining the movements from one point to another. The movement that creates a polygon is defined as a GeneralPath object found in the java.awt.geom package.

- The statement `GeneralPath pol = new GeneralPath();` creates a GeneralPath object named pol.
- The moveTo() method of GeneralPath is used to create the beginning point on the polygon. Thus, the statement `pol.moveTo(10F,193F);` starts the polygon named pol at the coordinates (10,193).

- The lineTo() method is used to create a line that ends at a new point. The statement `pol.lineTo(25F,183F);` creates a second point using the arguments of 25 and 183 as the x- and y-coordinates of the new point.
- The statement `pol.lineTo(100F,223F);` creates a third point. The lineTo() method can be used to connect the current point to the original point or, alternately, you can use the closePath() method without any arguments. Here we use the statement `pol.closePath()` to close the polygon.

Next you will you use the Java 2D drawing object types to create a Swing applet that illustrates sample rectangles, ovals, arcs, and polygons.

To create the JShapes2D Swing applet:

1. Open a new file in your text editor, and then enter the first few lines of a JShapes2D Swing applet:

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class JShapes2D extends JApplet
{
```

2. Enter the following statements to create a paint() method, create a Graphics environment gr, and cast the Graphics environment to a Graphics2D environment gr2D:

```
public void paint(Graphics gr)
{
 Graphics2D gr2D = (Graphics2D)gr;
```

3. Create two Rectangle2D.Float objects named rect and rect2. Draw the rect object, and then fill the rect2 object.

```
Rectangle2D.Float rect = new Rectangle2D.Float(10F, 10F,
  40F, 20F);
Rectangle2D.Float rect2 = new Rectangle2D.Float(10F, 40F,
  40F, 20F);
gr2D.draw(rect);
gr2D.fill(rect2);
```

4. Create two Ellipse2D.Float objects named ell and ell2. Draw the ell object and fill the ell2 object.

```
Ellipse2D.Float ell = new Ellipse2D.Float(10F,73F,40F,
  20F);
Ellipse2D.Float ell2 = new Ellipse2D.Float(10F,103F,40F,
  20F);
gr2D.draw(ell);
gr2D.fill(ell2);
```

5. Create two Arc2D.Float objects named ac and ac2. Draw the ac object and fill the ac2 object.

```
Arc2D.Float ac = new Arc2D.Float(10,133,30,33,30,120,
  Arc2D.PIE);
Arc2D.Float ac2 = new Arc2D.Float(10,163,30,33,30,120,
  Arc2D.PIE);
gr2D.draw(ac);
gr2D.fill(ac2);
```

6. Create a new GeneralPath object named pol. Set the starting point of the polygon and create two additional points. Use the closePath() method to close the polygon by connecting the current point to the starting point. Draw the pol object.

```
  GeneralPath pol = new GeneralPath();
  pol.moveTo(10F,193F);
  pol.lineTo(25F,183F);
  pol.lineTo(100F,223F);
  pol.closePath();
  gr2D.draw(pol);
 }
}
```

7. Save the file as**JShapes2D.java** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.

8. Open a new file in your text editor, and then enter the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "JShapes2D.class" WIDTH = 250
   HEIGHT = 250>
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJShapes2D.html** in the Chapter.10 folder on your Student Disk, and then run the program using the `appletviewer TestJShapes2D.html` command. Your output should look like Figure 10-29.

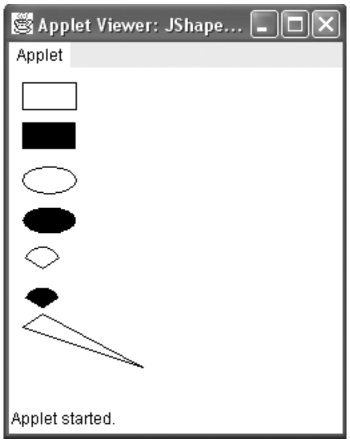


Figure 10-29 Output of the JShapes2D program

Adding Sound, Images, and Simple Animation to Swing Applets

Java 2 still supports sound using applet methods that have been available since the introduction of Java. This allows you to make Java programs audible using the methods of the Applet class to retrieve and play sound files in programs using various sound formats. Sound formats include the Windows Wave file format(.wav), Sun Audio file format(.au), and Music and Instrument Digital Interface file format(.midi).

The simplest way to retrieve and play a sound is to use the play() method of the Applet class. The **play() method** retrieves and plays the sound as soon as possible after it is called. The play() method takes one of two forms:

- play() with one argument—the argument is a Uniform Resource Locater (URL) object that loads and plays an audio clip when both the URL object and the audio clip are stored at the same URL.
- play() with two arguments—loads and plays the audio file (the first argument is a URL object and the second argument is a folder path name). The first argument will often be a call to a getCodeBase() method or getDocumentBase() method to retrieve the URL object; the second argument is the name of the audio clip within the folder path that is stored at that URL.

Tip

The `<APPLET>` tag was introduced in Chapter 9 to run an Applet from within an HTML document using the attributes CODE, HEIGHT, and WIDTH. The getCodeBase() and getDocumentBase()methods are Applet methods. By using these methods when loading sound or images, you make it possible for the applet to work even if you move it to another Web server.

Used with the CODEBASE attribute, which indicates the filename of the applet's main class file, the above methods direct the browser to look in a different folder for the applet and other files it uses. This is necessary when the desired files are in a different location than the Web page containing the applet. By calling getCodeBase() in an applet, you get a URL object that represents the folder where the applet's class file is stored. For example, the following statement retrieves and plays the event.au sound file which is stored at the same place as the applet: `play(getCodeBase(),"event.au");`.

Tip: The getDocumentBase() method returns an absolute URL naming the directory of the document in which the applet is stored. It is sometimes used instead of getCodeBase() as a matter of preference.

To play a sound more than once, or start or stop the sound, you must load the sound into an AudioClip object using the applet's newAudioClip() method. AudioClip is part of the java.awt.Applet class and must be imported into your program. The getAudioClip() method can take one or two arguments similar to the play() method. The first argument (or only argument, if there is only one) is a URL argument that identifies the sound file; the second argument is a folder path reference needed for locating the file.

The following statement loads the sound file from the previous example into the clip object: `AudioClip aClip = newAudioClip(getCodeBase(), "audio/event.au");`. Here the sound file reference indicates that the event.au sound file is located in the audio folder. After you have created an AudioClip object, you can use the play() method to call and play the sound, the stop() method to halt the playback, and the loop() method to play the sound repeatedly.

Next you will use the play() method and AudioClip to play a sound in a Swing applet. You will also create and add a Graphics2D object.

To play a sound and add a Graphics2D object in a Swing applet for Event Handlers, Inc.:

1. Open a new file in your text editor, and then enter the first few lines of the JEventSound Swing applet:

   ```
   import java.awt.*;
   import java.applet.*;
   import javax.swing.*;
   public class JEventSound extends JApplet
   {
   ```

2. Enter the following statement to declare an AudioClip object named sound:

   ```
   AudioClip sound;
   ```

3. Create the init() method and an AudioClip object to play the event.au sound file with the code:

```
public void init()
 {
  sound = getAudioClip(getCodeBase(),"event.au");
 }
```

4. Create the following start() method. The start method uses the loop() method to play the event.au sound file continually:

```
public void start()
 {
  sound.loop();
 }
```

5. Create the following stop() method to halt the event.au sound file.

```
public void stop()
{
 sound.stop();
}
```

6. Create a Graphics object using paint (Graphics g), and then use a cast to change the graphics context to a Graphics2D object. Use the drawstring() method to create a message which appears on the screen while the Swing applet play. Add a closing curly brace for the class.

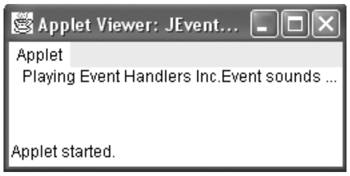
```
public void paint(Graphics g)
  {
  Graphics2D g2D = (Graphics2D)g;
  g2D.drawString
     ("Playing Event Handlers Inc. Event
  sounds ...", 10, 10);
 }
}
```

7. Save the file as **JEventSound.java** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.

8. Open a new file in your text editor, and then enter the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE = "JEventSound.class" WIDTH = 400
   HEIGHT = 250 >
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJEventSound.html** in the Chapter.10 folder on your Student Disk, and then run it using the `appletviewer TestJEventSound.html` command. The output should look like Figure 10-30. You should also be able to hear sound playing continually if speakers are installed for your system.



```
Playing Event Handlers Inc.Event sounds ...
```

Figure 10-30 Output of the JEventSound Swing applet

Adding Images

An **image** is a likeness of a person or thing. Images abound on the Internet in all shapes, colors and sizes. Images formats supported by Java include:

- Graphics Interchange Format (GIF), which can contain a maximum of 256 different colors
- Join Photographic Experts Group (JPEG) which is commonly used to store photographs, and is a more sophisticated way to represent a color image
- Portable Network Graphics (PNG), which is more flexible than the GIF, and stores its images in a lossless form (It was originally designed to be a portable image storage form for computer-originated images.)

Java's image capabilities include using the Image class and ImageIcon class to load images in one of the formats discussed earlier. The Image class is an abstract class. An **abstract** class is one from which you cannot create any objects, but from which you can inherit, so you must create Image objects indirectly. The Image class is found in the java.awt package, as shown in Figure 10-31. The ImageIcon class is particularly useful because it can be used to easily load an image into either an applet or an application. The ImageIcon class is part of the Swing package, as shown in Figure 10-32.

```
java.lang.Object
  |
  +--java.awt.Image
```

Figure 10-31 Structure of Image class

```
java.lang.Object
  |
  +--javax.swing.ImageIcon
```

Figure 10-32 Structure of ImageIcon class

To declare an Image with the name eventLogo1, you use the declaration `Image eventLogo1;`. The getImage() method is used to load an Image into the applet. Like the AudioClip method used for loading sound, one version of the getImage() method can take up to two arguments—a location where the image is stored and the filename of the image. You create and load the Image named eventImage with the statement:

```
eventLogo1 = getImage(getCodeBase(),"event.gif");
```

Notice that the Image object is not created directly. Instead, you request that an Image be loaded and returned to you.

Because the ImageIcon class is not an abstract class, you can create the ImageIcon eventLogo2 with the following statement:

```
eventLogo2 = new ImageIcon("event.gif");
```

This creates an ImageIcon object that holds the same event.gif as the eventLogo1 Image.

Tip

The ImageIcon class provides several constructors that allow an ImageIcon object to be initialized with an image from a local computer or an image stored on a Web server.

The applet's paint() method is used to display both Image and ImageIcon object images. The drawImage() method is a Graphics method which uses the following four arguments:

- The first argument is a reference to the Image object in which the image is stored.
- The second argument is the x-coordinate where the image will appear on the applet.
- The third argument is the y-coordinate where the image will appear on the applet.
- The fourth argument is a reference to an ImageObserver object.

An ImageObserver object can be any object that implements the ImageObserver interface. The Component class implements the ImageObserver interface so all components will inherit this implementation. Usually, the ImageObserver object is the object on which the image appears. For example, when the JApplet class implements this interface, it can track the progress of an image. This useful feature allows you to display a message such as "Please wait. Loading images." while graphics files are being loaded. Recall from Chapter 4 what you learned about the `this` reference. Here you use the `this` reference to refer to the applet in the following example. The code to display the eventLogo1 image is:

```
g.drawImage(eventLogo1,0,0,this);
```

A second version of the Graphics method, the drawImage() method, is unavailable to Image objects, but can be used with ImageIcon images. In this version, the drawImage()

method is used to output a scaled image. This method takes six arguments. Note that the first three arguments are the same as those for the other drawImage() method.

- The first argument is a reference to the Image object in which the image is stored.
- The second argument is the x-coordinate where the image will appear on the applet.
- The third argument is the y-coordinate where the image should appear on the applet.
- The fourth argument is a call to the getWidth() method to specify the image width for display purposes.
- The fifth argument is a call to the getHeight() method to specify the image height for display purposes (in the following example, the height should be 100 pixels fewer than the height of the applet).
- The sixth argument uses the `this` reference to implement the ImageObserver object.

The code to display the eventLogo2 image is:

```
g.drawImage(eventLogo1,0,120, getWidth(), getHeight() -
100, this);
```

You can also use the paintIcon() method to display ImageIcon images. This method requires four arguments:

- The first argument is a reference to the Component on which the image will appear—`this` in the following example.
- The second argument is a reference to the Graphics object that will be used to render the image—g in the following example.
- The third argument is the x-coordinate for the upper-left corner of the image.
- The fourth argument is the y-coordinate for the upper-left corner of the image.

The code to display the eventLogo2 image using the paintIcon() method is:

```
eventLogo2.paintIcon(this, g, 180, 0);
```

The completed Swing applet is shown in Figure 10-33 with the class name JEventImage. Output of the JEventImage is shown in Figure 10-34. You should review the preceding paragraphs while viewing the program and program output.

```
import javax.applet.*;
import java.awt.*;
import javax.swing.*;

public class JEventImage extends JApplet
{
   Image eventLogo1;
   ImageIcon eventLogo2;

   public void init()
   {
     eventLogo1 = getImage(getCodeBase(),"event.gif");
     eventLogo2 = new ImageIcon("event.gif");
   }
   public void paint (Graphics g)
     {
     g.drawImage(eventLogo1,0,0,this);
     g.drawImage(eventLogo1,0,120, getWidth(), getHeight()-100, this);
     eventLogo2.paintIcon(this, g, 180, 0);
   }
}
```

Figure 10-33 The JEventImage program

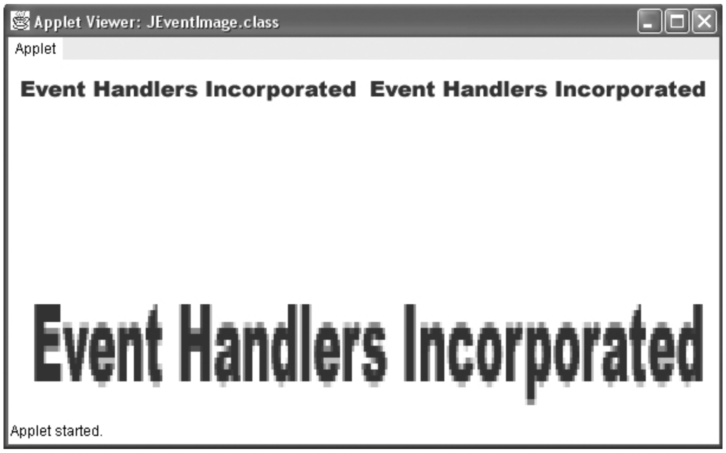

Applet Viewer: JEventImage.class
Applet
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Applet started.

Figure 10-34 Output of the JEventImage program

Adding Simple Animation

At some time in your life, you probably created simple animation by drawing a series of figures on the pages of a book, and slightly changing each version of the figure from the previous one. When you flipped through the pages of the book, the figure appeared to move. Movies, whether animated or not, are created in a similar manner—you see a succession of film frames, and each one contains a slightly modified image. Computer animation is achieved in the same fashion—a series of images appear on your screen in rapid succession. You will be able to create fairly sophisticated animation after you have covered Chapter 17; for now, you can use an ActionListener to control drawing different images using the paint() method. Although the results will not be truly animated, you can achieve dynamic results in which the time appears to change. This change is accomplished by displaying time as a String and updating the String time contents with successive clicks of a JButton.

Next you will create a Swing applet for Event Handlers Incorporated that contains a graphical representation of the current day, date, and time, which changes when the user clicks a JButton. In addition, a sound plays while the message " It's time to Party…" appears under the graphical time representation. An image of a banner of the Event Handlers company name appears under the message. A snapshot of the Swing applet at a random time appears in Figure 10-35.

To create the JGregorianTime applet:

1. Open a new text file, and enter the first few lines of the JGregorianTime Swing applet:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
public class JGregorianTime extends JApplet implements
ActionListener
{
```

2. Add the following statements to create an AudioClip named sound, a new Color named tan, an empty String named lastTime, and an ImageIcon named eventLogo:

```
private AudioClip sound;
private Color tan = new Color(255, 204, 102);
private String lastTime = "";
private ImageIcon eventLogo ;
```

3. Add the following statements to create a JButton for the user to click:

```
JButton pressMe = new JButton("pressMe");
```

4. Begin the init() method, and enter the following statements to add a sound object and ImageIcon object:

```
public void init()
{
  sound = getAudioClip(getCodeBase(),"event.au");
  eventLogo = new ImageIcon("event.gif");
```

5. Add the following statements to set the background of the applet, create a container name `con`, and change the default layout from BorderLayout to FlowLayout:

```
setBackground(Color.blue);
Container con = getContentPane();
con.setLayout(new FlowLayout());
```

6. Add the following statements, which add the JButton named pressMe to the container and add an ActionListener() for pressMe. Then add the closing curly brace to the init() method.

```
  con.add(pressMe);
  pressMe.addActionListener(this);
}
```

7. Begin the paint() method and cast the Graphics context to Graphics 2D. Create a new font named monoFont and set the font of the Graphics2D object to monoFont. (Note the syntax for setting monoFont in a Graphics 2D environment.)

```
public void paint(Graphics g)
{
  Graphics2D g2D = (Graphics2D)g;
  Font monoFont = new Font("Monospaced", Font.BOLD, 20);
  g2D.setFont(monoFont);
```

8. Create a new GregorianCalendar object named day, create a String named time using the getTime() method, and then convert the result using the toString() method.

```
GregorianCalendar day = new GregorianCalendar();
String time = day.getTime().toString();
```

9. Use the setColor()method and drawString() method to set the color and draw the strings lastTime and time. (Note that when the animation starts, the String lastTime is empty.) Set the g2D object's color to tan, and then draw the time string. Finally, reference the String lastTime to the String time.

```
g2D.setColor(Color.blue);
g2D.drawString(lastTime, 5, 75);
g2D.setColor(tan);
g2D.drawString(time, 5, 75);
lastTime = time;
```

10. Add the ImageIcon eventLogo to the paint() method below the graphic representation of the day, date, and time. Add the drawString() method under the eventLogo Image to display the string "It's time to Party...". Add a repaint() method so that the JButton will be redrawn each time the paint() method is called, and then add the closing curly brace to the paint() method.

```
  eventLogo.paintIcon(this, g, 50, 120);
  g2D.drawString("It's time to Party...", 50, 100);
  pressMe.repaint();
}
```

11. Add the Swing applet's start() and stop() methods. Add the loop() method to the applet's start() method to play the sound continually. Then add the stop method to the applet's stop() method.

```
public void start()
{
  sound.loop();
}
public void stop()
{
  sound.stop();
}
```

12. At this point, the Swing applet is almost completed. You still must add the actionPerformed() method that executes when the user clicks the JButton. The only task performed by the method is to call the repaint() method. Add the following method to your applet. Then add the closing curly braces for the acitonPerformed() method and the class.

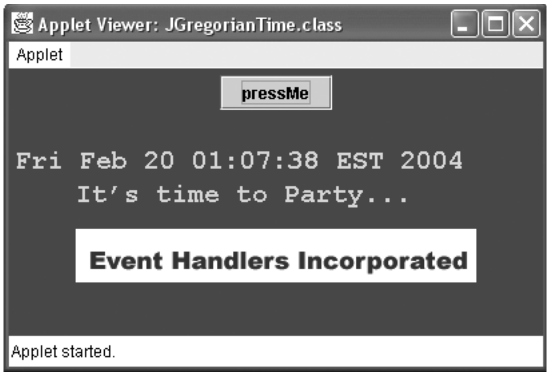
```
public void actionPerformed(ActionEvent e)
 {
  Object source = e.getSource();
  if(source == pressMe)
   repaint();
  }
}
```

13. Save the file as **JGregorianTime** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.

14. Open a new file in your text editor, and then enter the code for an HTML document to host the applet:

```
<HTML>
<APPLET CODE = "JGregorianTime.class"
   WIDTH = 400 HEIGHT = 200>
</APPLET>
</HTML>
```

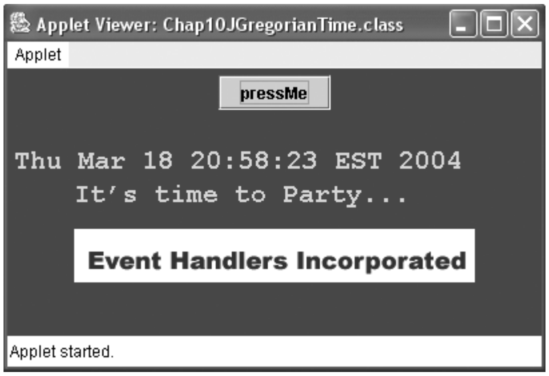
15. Save the HTML document as **TestJGregorianTime.html** in the Chapter.10 folder on your Student Disk, and then run it using the `appletviewer TestJGregorianTime.html` command. The output appears in Figure 10-35. Wait at least a couple of minutes, click the **pressMe** button, wait a period of time, and observe the time changes. Figure 10-36 shows the output of the program after the JButton has been clicked.



```
Fri Feb 20 01:07:38 EST 2004
    It's time to Party...
```

Event Handlers Incorporated

Figure 10-35 A snapshot of the JGregorianTime applet



```
Thu Mar 18 20:58:23 EST 2004
    It's time to Party...
```

Event Handlers Incorporated

Figure 10-36 Output of the JGregorianTime program after clicking the JButton

16. Close the Applet Viewer window.

CHAPTER SUMMARY

- The paint() method header, `public void paint (Graphics g)`, requires a Graphics object argument. The paint() method runs when you display, maximize, minimize, or restore an applet. You can use the paint() method automatically supplied by Java, or you can write your own. You don't usually call the paint() method directly. Instead, you call the repaint() method, which calls the update() method, which then calls the paint() method.
- You use the drawString() method to draw a String in an applet. The drawString() method requires three arguments: a String, an x-coordinate, and a y-coordinate. The x- and y-coordinates represent the lower-left position of the String. The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it.

- You can designate a Graphics color with the setColor() method. The Color class contains 13 constants: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, and yellow. You can create your own Color object with the statement `Color someColor = new Color(r, g, b);`, where r, g, and b are numbers representing the intensity of red, green, and blue you want in your color.
- You can use the drawLine() method to draw a straight line between any two points on the screen. The drawLine() method takes four arguments: the x- and y-coordinates of the line's starting point, and the x- and y-coordinates of the line's ending point.
- You can use the drawRect() and fillRect() methods, respectively, to draw the outline of a rectangle or to draw a solid, filled rectangle. Each of these methods requires four arguments. The first two arguments represent the x- and y-coordinates of the upper-left corner of the rectangle. The last two arguments represent the width and height of the rectangle.
- The drawOval() and fillOval() methods draw ovals using four arguments—the x- and y-coordinates for the upper-left corner, and the width and height measurements of an imaginary rectangle that surrounds the oval.
- An arc is a portion of a circle that you can draw using the Graphics method drawArc(). The drawArc() method requires six arguments: the x-coordinate of the upper-left corner of an imaginary rectangle that represents the bounds of the imaginary circle that contains the arc, the y-coordinate of the same point, the width of the imaginary rectangle that represents the bounds of the imaginary circle that contains the arc, the height of the same point, the beginning arc position, and the arc angle.
- You can discover the fonts that are available on your system by calling the getLocalGraphicsEnvironment() method and then the resulting object to reference its getAllFonts() method. You can also discover the resolution and screen size on your computer system by using the getScreenResolution()and getScreenSize() methods that are part of the Toolkit class.
- A Graphics2D environment can be created in a paint() method by casting a Graphics environment.
- Drawn objects in Java 2D are created by defining them as geometric shapes using the java.awt.geom package classes. You can draw lines, rectangles, ovals, arcs, and polygons. The Graphics2D class does not have different methods for each of the shapes you can draw. Instead, you define the shape and use it as an argument to the draw() or fill() methods.
- Images can be added to an applet using the Image and ImageIcon classes.
- Sound can be added to an applet by first retrieving a sound file with WAV, AU, or MIDI formats. The sound can then be played using an AudioClip object.

REVIEW QUESTIONS

1. The method that calls the paint() method for you is ________________.
   a. callPaint()
   b. repaint()
   c. requestPaint()
   d. draw()
2. The paint() method header requires a(n) ________________ argument.
   a. void
   b. integer
   c. String
   d. Graphics
3. The statement `g.drawString(someString, 50, 100);` places someString's ________________ corner at position 50, 100.
   a. upper-left
   b. lower-left
   c. upper-right
   d. lower-right
4. If you use the setColor() method to change a Graphics object's color to yellow, ________________ will appear in yellow.
   a. only the next graphics output
   b. all graphics output for the remainder of the method
   c. all graphics output for the remainder of the applet
   d. all graphics output until you change the color
5. The correct statement to instantiate a Graphics object named picasso is ________________.
   a. `Graphics picasso;`
   b. `Graphics picasso = new Graphics();`
   c. `Graphics picasso = getGraphics();`
   d. `Graphics picasso = getGraphics(new);`
6. The statement `g.drawRoundRect(100,100,100,100,0,0);` draws a shape that looks most like a ________________.
   a. square
   b. round-edged rectangle
   c. circle
   d. straight line

7. If you draw an oval with the same value for width and height, then you draw a(n) ________________.
    a. circle
    b. square
    c. rounded square
    d. ellipsis
8. The zero-degree position for any arc is at the ________________ o'clock position.
    a. three
    b. six
    c. nine
    d. 12
9. The method you use to create a solid arc is ________________.
    a. solidArc()
    b. fillArc()
    c. arcSolid()
    d. arcFill()
10. You use the ________________ method to copy any rectangular area to a new location.
    a. copyRect()
    b. copyArea()
    c. repeatRect()
    d. repeatArea()
11. The measurement of an uppercase character from the baseline to the top of the character is its ________________.
    a. ascent
    b. descent
    c. leading
    d. height
12. To be sure that a vertical series of Strings has enough room to appear in an applet, you would use which of the following statements?
    a.
    ```
    g.drawString("Some string", x,
       y += g.getFontMetrics(). getHeight());
    ```
    b.
    ```
    g.drawString("Some string", x,
       y += g.getFontMetrics(). getLeading());
    ```

c. `g.drawString("Some string", x,`
   `y += g.getFontMetrics(). getAscent());`

d. `g.drawString("Some string", x,`
   `y += g.getFontMetrics(). getDescent());`

13. You can discover the fonts that are available on your system by using the ________________.
    a. getAllFonts() method of the GraphicsEnvironment class
    b. getAllFonts() method of the Graphics class
    c. setAllFonts() method of the GraphicsEnvironment class
    d. getAllFonts() method of the ImageEnvironment class
14. The getScreenResolution() method and getScreenSize() method ________________.
    a. both return the number of pixels as an int type
    b. return the number of pixels as an int type and an object of type Dimension
    c. both return an object of type Dimension
    d. return the number of pixels as a double type and an object of type Dimension
15. A Graphics2D object is produced by ________________.
    a. the setGraphics2D() method
    b. the `Graphics2D newpen = Graphics2D()` statement
    c. the `Graphics2D = Graphics(g)` statement
    d. casting a Graphics object
16. Java 2D uses ________________ when creating and drawing a 2D drawing object.
    a. only coordinate space
    b. only user coordinate space
    c. both coordinate and user coordinate space
    d. only 2D coordinate space
17. A gradient fill is a gradual change in ________________.
    a. color
    b. font size
    c. drawing style
    d. line thickness
18. After the getAudioClip() method retrieves a sound object named mysound, the ________________ plays a sound continually in a Swing applet.
    a. sound.loop() method
    b. loop() method
    c. mysound.loop() method
    d. mysound.continuous() method

19. The ______________ is particularly useful for loading an image into either an applet or application.
    a. Image class
    b. ImageLogo class
    c. ImageIcon class
    d. GetImage class
20. Showing successive images on the screen is called ______________.
    a. action-oriented
    b. object-oriented
    c. animation
    d. volatility

Exercises

1. Write a Swing applet that demonstrates displaying your first name in every even-numbered font size from 4 through 24. Save the program as **JFontSizeDemo.java** in the Chapter.10 folder on your Student Disk.
2. Write a Swing applet that displays your name in blue the first time the user clicks a JButton, and then displays your name larger and in gray the second time the user clicks the JButton. Save the program as **JBlueGray.java** in the Chapter.10 folder on your Student Disk.
3. Write a Swing applet that displays a form for creating an e-mail directory. The form should contain three JTextFields and three JLabels for first name, last name, and e-mail address. After the user enters an e-mail address and presses [Enter], the program should display the information that was entered. Use the drawString() method to display the information. Use the paint() method to display a heading line for the information display, such as "The e-mail information you entered is: ". Save the program as **JEmailForm.java** in the Chapter.10 folder on your Student Disk.
4. a. Write a Swing applet that displays a yellow smiling face on the screen. Save the program as **JSmileFace.java** in the Chapter.10 folder on your Student Disk.
   b. Add a JButton to the JSmileFace Swing applet so the smile changes to a frown when the user clicks the JButton. Save the program as **JSmileFace2.java** in the Chapter.10 folder on your Student Disk.
5. a. Use polygons and lines to create a graphics image that looks like a fireworks display. Write a Swing applet that displays the fireworks. Save the program as **JFireworks.java** in the Chapter.10 folder on your Student Disk.
   b. Add a JButton to the JFireworks Swing applet. Do not show the fireworks until the user clicks the JButton. Save the program as **JFireworks2.java** in the Chapter.10 folder on your Student Disk.

6. a. Write a Swing applet to display your name. Place boxes around your name at intervals of 10, 20, 30, and 40 pixels. Save the program as **JBorders.java** in the Chapter.10 folder on your Student Disk.

   b. Make each of the four borders in the JBorders.java applet display a different color. Save the program as **JBorders2.java** in the Chapter.10 folder on your Student Disk.

7. Create a Swing applet and use dialog boxes to prompt the user to enter a name and weight in pounds. Once the name and weight are entered, use Graphics2D methods to display the user's name and weight, with weight displayed in pounds, ounces, kilograms, and metric tons on separate lines. Use the following conversion factors:

   1 pound = 16 ounces

   1 kilogram = 1 pound / 2.204623

   1 metric ton = 1pound / 2204.623

   Save the program as **JCalculateWeight.java** in the Chapter.10 folder on your Student Disk.

8. Write a Swing applet that uses the Graphics2D environment to create a GeneralPath object. Use the General path object to create the outline of your favorite state. Display the state name at the approximate center of the state boundaries. Save the program as **JFavoriteState.java** in the Chapter.10 folder on your Student Disk.

9. Write a Swing applet that draws a realistic-looking Stop sign. Save the program as **JStopSign.java** in the Chapter.10 folder on your Student Disk.

10. Write a Swing applet that uses the ImageIcon class to place image icon objects on four JButtons. Download your favorite GIF files from the Internet. If necessary, reduce the size of the GIF images to approximately 30 by 30 pixels, or you can use the four GIF files in your Student Disk if you wish. Each time a JButton is clicked, display a different message below the JButtons. Save the program as **JButtonIcons.java** in the Chapter.10 folder on your Student Disk.

11. Each of the following files in the Chapter.10 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugTen1.java will become FixDebugTen1.java. You can test each applet with the TestDebugTen.html files on your Student Disk. Remember to change the Java class file referenced in the HTML document so it matches the DebugTen applet on which you are working.

    a. DebugTen1.java

    b. DebugTen2.java

    c. DebugTen3.java

    d. DebugTen4.java

Case Project

The Party Planners organization in your local town is sponsoring a contest to see who can program the best Java Swing applet to be used as an advertisement for their party events. You can download sound clips and graphics images from the Internet to use in your program. Create a Swing applet named JPartyPlanner and an HTML test file to run the Swing applet. Good luck! I hope you win!

CHAPTER

11

INTRODUCTION TO INHERITANCE

In this chapter, you will:

- Learn about the concept of inheritance
- Extend classes
- Override superclass methods
- Work with superclasses that have constructors
- Use superclass constructors that require arguments
- Access superclass methods
- Learn about information hiding
- Use methods you cannot override

"You look exhausted," Lynn Greenbrier says to you late one Friday afternoon.

"I am," you reply."Now that I know some Java, I am writing program after program for several departments in the company. It's fun, but it's a lot of work, and the worst thing is that I seem to be doing a lot of the same work over and over."

"What do you mean?" Lynn asks.

"Well, the Event Planning Department asked me to develop several classes that will hold information for every event type handled by Event Handlers. There are weekday and weekend events, events with or without dinners, and events with or without guest speakers. Sure, these various types of events have differences, but all events have many things in common, such as an event number and a number of guests."

"I see," Lynn says."You'd like to create a class based on an existing class, just by adding the specific new components the new class needs. You want to avoid rewriting the components that you already created."

"Exactly!" you say."But, since I can't do that, I guess I'll just have to get back to work."

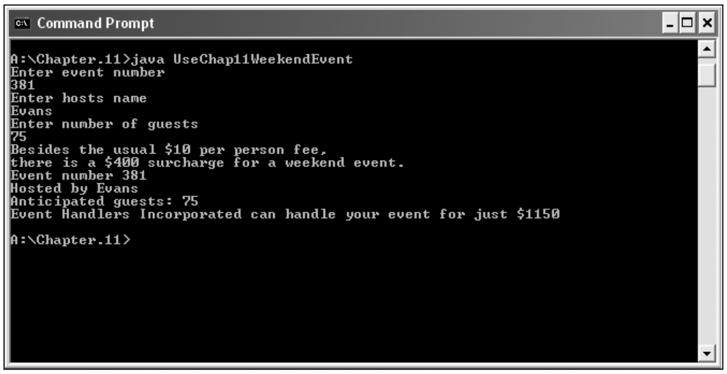
"Go home and relax!" Lynn says."On Monday morning, I'll teach you how to use inheritance to solve these problems."

PREVIEWING AN EXAMPLE OF INHERITANCE

Weekend events hosted by Event Handlers Incorporated cost more than weekday events because staff members who work at the events are paid overtime rates. Weekend events have all the components of all other events, but they also possess this surcharge. You can use compiled versions of the Chap11Event, Chap11WeekendEvent, and UseChap11WeekendEvent class files that are saved in the Chapter.11 folder on your Student Disk to run the UseChap11WeekendEvent program.

To use the Chap11WeekendEvent class:

1. Go to the command prompt for the Chapter.11 folder on your Student Disk, type **`java UseChap11WeekendEvent`**, and then press **[Enter]**. This program allows you to supply data for a weekend event. The prompts will ask you for an event number, host name, and number of guests. You can supply any answers you want to these questions. The data you enter will echo to the screen, and you will see the charge for the event. The charge is calculated at $10 per person, plus a $400 surcharge for the weekend. Figure 11-1 shows a typical program run.



```
A:\Chapter.11>java UseChap11WeekendEvent
Enter event number
381
Enter hosts name
Evans
Enter number of guests
75
Besides the usual $10 per person fee,
there is a $400 surcharge for a weekend event.
Event number 381
Hosted by Evans
Anticipated guests: 75
Event Handlers Incorporated can handle your event for just $1150

A:\Chapter.11>
```

Figure 11-1 Output of the UseChap11WeekendEvent program

2. Open your text editor, and then open the **UseChap11WeekendEvent.java** file, or examine the code shown in Figure 11-2. The program creates a Chap11WeekendEvent object named anEvent, and then calls five methods.

3. Open the **Chap11WeekendEvent.java** file in your text editor, or examine the code shown in Figure 11-3. The class Chap11WeekendEvent does not contain any data fields, and it contains only one method, named computePrice(). However, when you ran the program UseChap11WeekendEvent, you provided input for several data fields. The program called several methods, and displayed several lines of output. The additional fields and methods were available because the WeekendEvent class inherited its additional components. You will create similar classes in this chapter.

```
public class UseChapter11WeekendEvent
{
    public static void main(String args[]) throws Exception
    {
      Chap11WeekendEvent anEvent = new Chapter11WeekendEvent();
      anEvent.setEventNum();
      anEvent.setEventHost();
      anEvent.setnumGuests();
      anEvent.computePrice();
      anEvent.printDetails();
    }
}
```

Figure 11-2 UseChap11WeekendEvent.java program

```
public class Chapter11WeekendEvent extends Chap11Event
{
    public void computePrice()
    {
      super.computePrice();
      quotedPrice += 400;
      System.out.println
        ("Besides the usual $10 per person fee,");
      System.out.println
        ("there is a $400 surcharge for a weekend event.");
    }
}
```

Figure 11-3 Chap11WeekendEvent class

Learning About the Concept of Inheritance

Inheritance is the principle that allows you to apply your knowledge of a general category to more-specific objects. In Java, **inheritance** is a mechanism that enables one class to inherit both the behavior and the attributes of another class. You are familiar with the concept of inheritance from all sorts of nonprogramming situations.

Tip

In Chapter 3, you first learned about inheritance, where a class object can inherit all the attributes of an existing class. A functional new class can be created simply by indicating how it is different from the class from which it is derived.

When you use the term inheritance, you might think of genetic inheritance. You know from biology that your blood type and eye color are the product of inherited genes; you can say that many facts about you—or your data fields—are inherited. Similarly, you often can attribute your behaviors to inheritance. For example, your attitude toward saving money might be the same as your grandma's, and the odd way that you pull on your ear when you are tired might also be what your Uncle Steve does—thus your methods are inherited, too.

You also might choose plants and animals based on inheritance. You plant impatiens next to your house because of your shady street location; you adopt a Doberman pinscher because you need a watchdog. Every individual plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, you are provided with data fields and methods automatically.

From the first chapter of this book, you have been creating classes and instantiating objects that are members of those classes. For example, consider the simple Employee class shown in Figure 11-4. The class contains two data fields, empNum and empSal, and four methods, a get and set method for each field.

```
public class Employee
{
    private int empNum;
    private double empSal;
    public int getEmpNum()
    {
      return empNum;
    }
    public double getEmpSal()
    {
      return empSal;
    }
    public void setEmpNum(int num)
    {
      empNum = num;
    }
    public void setEmpSal(double sal)
    {
      empSal = sal;
    }
}
```

Figure 11-4 Employee class

After you create the Employee class, you can create specific Employee objects, such as `Employee receptionist = new Employee();` and `Employee deliveryPerson = new Employee();`. These Employee objects can eventually possess different numbers and salaries, but because they are Employee objects, you know that each Employee has *some* number and salary.

Suppose you hire a new Employee named serviceRep. A serviceRep object requires an employee number and a salary, but a serviceRep object also requires a data field to indicate territory served. You can create a new class with a name such as EmployeeWithTerritory, and provide the class three fields (empNum, empSal, and empTerritory) and six methods (get and set methods for each of the three fields).

However, when you do this, you are duplicating much of the work that you have already done for the Employee class. The wise, efficient alternative is to create the class EmployeeWithTerritory so it inherits all the attributes and methods of Employee. Then, you can add just the one field and two methods that are additions within EmployeeWithTerritory objects. Figure 11-5 shows a diagram of this relationship.

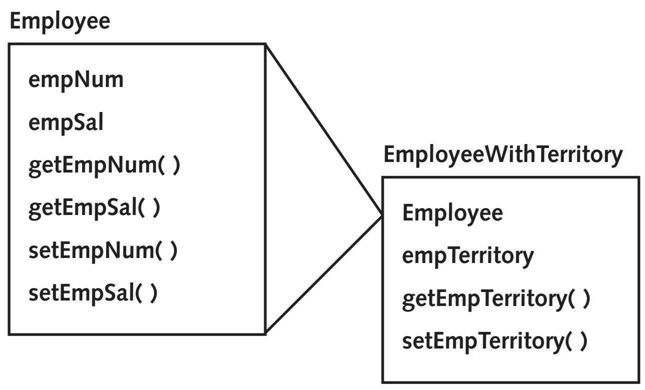


Figure 11-5 EmployeeWithTerritory class inherits from Employee class

When you use inheritance to create the EmployeeWithTerritory class, you:

- Save time, because the Employee fields and methods already exist
- Reduce errors, because the Employee methods already have been used and tested
- Ease understanding, because you have used the Employee methods on simpler objects and already understand how they work

The ability to use inheritance in the Java programming language makes programs easier to write, less error prone, and easier to understand. Imagine that besides creating EmployeeWithTerritory, you also want to create several other specific Employee classes (perhaps EmployeeEarningCommission including a commission rate, or DismissedEmployee including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly.

Tip: The concept of a class inheritance is useful because it makes class code reusable.

A class that is used as a basis for inheritance, such as Employee, is called a **base class**. When you create a class that inherits from a base class (such as EmployeeWithTerritory), it is a **derived class**. When confronted with two classes that inherit from each other, you can tell which class is the base class and which class is the derived class by using the two classes in a sentence with the phrase "is a". A derived class always "is a" case or instance of the more general base class. For example, a Tree class may be a base class to an Evergreen class. An Evergreen "is a" Tree, so Tree is the base class; however, it is not true for all Trees that "a Tree is an Evergreen". Similarly, an EmployeeWithTerritory "is an" Employee—but not the other way around—so Employee is the base class.

You can use the terms **superclass** and **subclass** as synonyms for base class and derived class. Thus, Evergreen can be called a subclass of the Tree superclass. You can also use the terms **parent class** and **child class**. An EmployeeWithTerritory is a child to the Employee parent. Use the pair of terms with which you are most comfortable; all of these terms will be used interchangeably throughout the book.

As an alternate way to discover which of two classes is the base class or subclass, you can try saying the two class names together. When people say their names together, they state the more-specific name before the all-encompassing family name, as in "Ginny Kroening". Similarly, with classes, the order that "makes more sense" is the child-parent order. "Evergreen Tree" makes more sense than "Tree Evergreen", thus Evergreen is the child class.

Finally, you usually can distinguish superclasses from their subclasses by size. Although it is not required, in general a subclass is larger than a superclass because it usually has additional fields and methods. A subclass description might look small, but any subclass contains all the fields and methods of its superclass, as well as the new, more-specific fields and methods you add to that subclass.

Extending Classes

You use the keyword `extends` to achieve inheritance within the Java programming language. For example, the class header `public class EmployeeWithTerritory extends Employee` creates a superclass-subclass relationship between Employee and EmployeeWithTerritory. Each EmployeeWithTerritory automatically receives the data fields and methods of the superclass Employee; you then add new fields and methods to the newly created subclass. Figure 11-6 shows an EmployeeWithTerritory class.

Tip

You used the phrase `extends JApplet` throughout Chapters 9 and 10. Every Swing applet that you write is a child of the JApplet class.

When you write a program that instantiates an object using the statement `EmployeeWithTerritory northernRep = new EmployeeWithTerritory();`, then you can use any of the following statements to get field values for northernRep:

- `northernRep.getEmpNum();`
- `northernRep.getEmpSal();`
- `northernRep.getTerritoryNum();`

```
public class EmployeeWithTerritory extends Employee
{
  private int territoryNum;
  public int getTerritoryNum()
  {
    return territoryNum;
  }
  public void setTerritoryNum(int num)
  {
    territoryNum = num;
  }
}
```

Figure 11-6 EmployeeWithTerritory class

The northernRep object has access to all three methods—two methods that it inherits from Employee and one method that belongs to EmployeeWithTerritory.

Similarly, any of the following statements are legal:

- `northernRep.setEmpNum(915);`
- `northernRep.setEmpSal(210.00);`
- `northernRep.setTerritoryNum(5);`

Inheritance is a one-way proposition; a child inherits from a parent, not the other way around. If a program instantiates an Employee object, as in **`Employee aClerk = new Employee;`**, then the Employee object does not have access to the EmployeeWithTerritory methods. Employee is the parent class, and aClerk is an object of the parent class. It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you will not know how many future subclasses there might be, or what their data or methods might look like. In addition, subclasses are more specific. An Orthodontist class and Periodontist class are children of the Dentist class. You do not expect all members of the general parent class Dentist to have the Orthodontist's braces() method or the Periodontist's deepClean() method. However, Orthodontist objects and Periodontist objects have access to the more general Dentist methods takeXRays() and billPatients().

Next you will create a working example of inheritance. You will create this example in four stages:

1. First, you will create a general Event class for Event Handlers Incorporated. This Event class will be small—it will hold just one data field and two methods.
2. After you create the general Event class, you will write a program to demonstrate its use.
3. Then you will create a more-specific DinnerEvent subclass that inherits the attributes of the Event class.

4. Finally, you will modify the demonstration program to add an example using the DinnerEvent class.

To create the general Event class:

1. Open a new file in your text editor, and then enter the following first few lines for a simple Event class. The class will host one integer data field—the number of guests expected at the event.

```
public class Event
{
 private int eventGuests;
```

2. To the Event class, add the following method that displays the number of eventGuests:

```
public void printEventGuests()
{
 System.out.println("Event guests: " + eventGuests);
}
```

3. Add a second method that prompts the user for the number of guests and stores the response in the eventGuests field. Begin by typing the following method header, which includes the `throws Exception` clause that handles data entry:

```
public void setEventGuests() throws Exception
{
```

Tip: You first learned about `throws Exception` in Chapter 5.

4. Enter the following code to add an integer variable to hold each character as it is read from the keyboard and a String variable to which you will add each numeric character the user enters:

```
char inChar;
String guestsString = new String("");
```

5. Prompt the user for the guest number and read the response using the following code:

```
System.out.print
 ("Enter the number of guests at your event ");
inChar = (char)System.in.read();
```

6. Enter the following `while` loop. While the user continues to enter digits, add each to a String.

```
while(inChar >= '0' && inChar <= '9')
{
  guestsString = guestsString + inChar;
  inChar = (char)System.in.read();
 }
```

7. When the user finishes entering digits, use the parseInt() method to assign the String value to the eventGuests data field. In addition, add one more read() statement to absorb the extra byte from [Enter].

```
  eventGuests = Integer.parseInt(guestsString);
  System.in.read();
  }
 }
```

8. Save the file as **Event.java** in the Chapter.11 folder on your Student Disk. At the command prompt, compile the class using the **`javac Event.java`** command. If necessary, correct any errors and compile again.

Now that you have created a class, you can use it in an application or Swing applet. A very simple application creates an Event object, sets a value for the data field, and then displays the results.

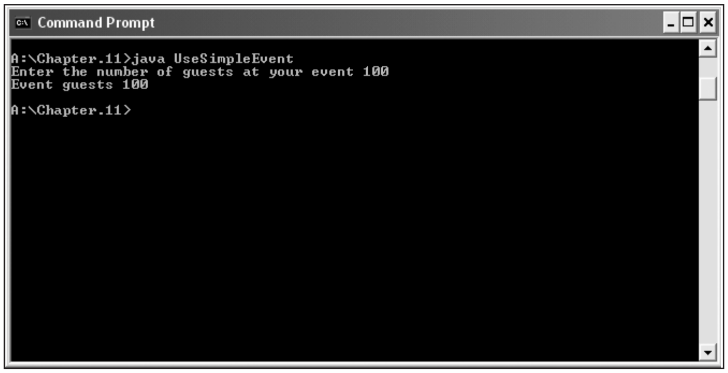
To write a simple application that uses the Event class:

1. Open a new file in your text editor.
2. Write a UseSimpleEvent program that has one method—a main() method. Enter the following main() method, which declares an Event object, supplies it with a value, and then prints the value. Add a closing curly brace for the class:

```
public class UseSimpleEvent
{
 public static void main(String[] args) throws Exception
 {
  Event anEvent = new Event();
  anEvent.setEventGuests();
  anEvent.printEventGuests();
 }
}
```

3. Save the file as **UseSimpleEvent.java** in the Chapter.11 folder on your Student Disk. Compile the program using the **`javac UseSimpleEvent.java`** command. After the program compiles without errors, run the program by typing **`java UseSimpleEvent`**, type **100** and press **[Enter]**, type **Evans** and press **[Enter]**, then type **75** and press **[Enter]**. The program's output appears in Figure 11-7.

Next you will create a new class named DinnerEvent. A DinnerEvent "is a" type of Event at which dinner is served, so DinnerEvent will be a child class of Event.


Command Prompt

```
A:\Chapter.11>java UseSimpleEvent
Enter the number of guests at your event 100
Event guests 100

A:\Chapter.11>
```

Figure 11-7 Output of the UseSimpleEvent program

To create a DinnerEvent class that extends Event:

1. Open a new file in your text editor, and type the following header for the DinnerEvent class:

   ```
   public class DinnerEvent extends Event
   {
   ```

2. A DinnerEvent contains a number of guests, but you do not have to define the variable here. The variable is already defined in Event, which is the superclass of this class. You only need to add any variables that are particular to a DinnerEvent. Enter the following code to add a character to hold the dinner menu choice, which will be b or c (for beef or chicken) for each DinnerEvent object: **`char dinnerChoice;`**.

3. The Event class already contains methods to set and print the number of guests, so DinnerEvent only needs methods to print and set the dinnerChoice variable. To keep this example simple, you will not validate the input character to ensure that it is b or c; you can add this improvement to the method later. The printDinnerChoice() method will assume that if the choice is not beef, it must be chicken. Type the printDinnerChoice() method as follows:

   ```
   public void printDinnerChoice()
   {
    if(dinnerChoice == 'b')
     System.out.println("Dinner choice is beef");
    else
     System.out.println("Dinner choice is chicken");
   }
   ```

4. Enter the following setDinnerChoice() method, which prompts the user for the choice of entrées at the event, and then add a closing curly brace for the class:

   ```
    public void setDinnerChoice() throws Exception
   {
     System.out.println("Enter dinner choice");
   ```

```
    System.out.print("b for beef, c for chicken ");
    dinnerChoice = (char)System.in.read();
    System.in.read(); System.in.read();
   }
  }
```

5. Save the file as **DinnerEvent.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Now you can modify the UseSimpleEvent program so that it creates a DinnerEvent as well as a plain Event.

To modify the UseSimpleEvent program:

1. Open the **UseSimpleEvent.java** file in your text editor.
2. Change the class name from UseSimpleEvent to **UseDinnerEvent**.
3. Position your insertion point at the end of the line that constructs anEvent, and then press **[Enter]** to start a new line. Type the following println() statement so that when you run the program you will know that you are using the Event class to create the event:

   ```
   System.out.println("A plain event");
   ```

4. Position your insertion point at the end of the line that prints the event guests (just before the closing curly braces), and then press **[Enter]** to start a new line. Add the following two new statements—one constructs a Dinner event, and the other prints a line so that when you run the program you will understand you are creating a DinnerEvent:

   ```
   DinnerEvent aDinnerEvent = new DinnerEvent();
   System.out.println("An event with dinner");
   ```

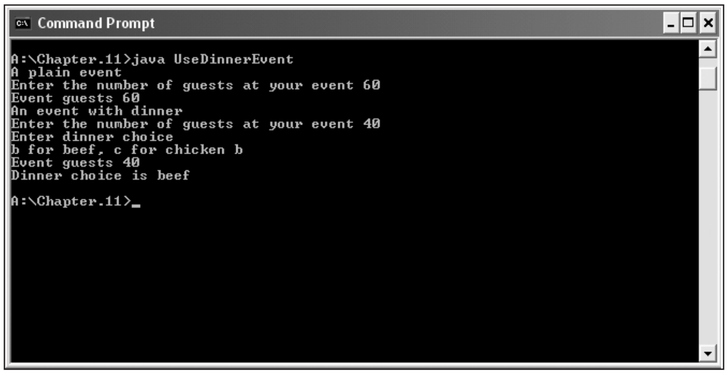
5. Add the following method calls to set the number of guests and dinner choice for the DinnerEvent object. Even though the DinnerEvent class does not contain a setEventGuests() method, its parent class (Event) does, so aDinnerEvent can use the setEventGuests() method.

   ```
   aDinnerEvent.setEventGuests();
   aDinnerEvent.setDinnerChoice();
   ```

6. Enter the following code to call the methods that print the entered data:

   ```
   aDinnerEvent.printEventGuests();
   aDinnerEvent.printDinnerChoice();
   ```

7. Save the file as **UseDinnerEvent.java** in the Chapter.11 folder on your Student Disk. Compile the program and run it using the values shown in Figure 11-8. The DinnerEvent object successfully uses the data field and methods of its superclass, as well as its own data field and methods.



```
A:\Chapter.11>java UseDinnerEvent
A plain event
Enter the number of guests at your event 60
Event guests 60
An event with dinner
Enter the number of guests at your event 40
Enter dinner choice
b for beef, c for chicken b
Event guests 40
Dinner choice is beef

A:\Chapter.11>_
```

Figure 11-8 Output of the UseDinnerEvent program

Overriding Superclass Methods

When you create a new subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. Sometimes those superclass data fields and methods are not entirely appropriate for the subclass objects.

When you use the English language, you often use the same method name to indicate diverse meanings. For example, if you think of MusicalInstrument as a class, you can think of play() as a method of that class. If you think of various subclasses such as Guitar and Drum, you know that you carry out the play() method quite differently for each subclass. Using the same method name to indicate different implementations is called **polymorphism**. The word polymorphism means "many forms"; many forms of action take place, even though you use the same word to describe the action. In other words, there are many forms of the same word depending on the object associated with the word.

For another example, you can create an Employee superclass containing data fields such as firstName, lastName, socialSecurityNumber, dateOfHire, rateOfPay, and so on. The methods contained in the Employee class include the usual set and get methods. If your usual time period for payment to each Employee object is weekly, then your getRateOfPay() method might include a statement such as `System.out.println("Pay is " + rateOfPay + " per week");`.

Imagine your company has a few Employees who are not paid weekly. Maybe some are paid by the hour, and others are Employees whose work is contracted on a job-to-job basis. Because each Employee type requires different paycheck-calculating procedures, you might want to create subclasses of Employee, such as HourlyEmployee and ContractEmployee.

When you call the getRateOfPay() method for an HourlyEmployee object, you want the display to include the phrase "per hour", as in "Pay is $8.75 per hour". When you call the getRateOfPay() method for a ContractEmployee, include "per contract" as in "Pay is $2000 per contract". Each class—the Employee superclass and the two subclasses—requires its own getRateOfPay() method. Fortunately, if you create separate getRateOfPay() methods for each class, then each class's objects will use the appropriate method for that class.

Tip: It is important to note that each subclass method overrides any method in the parent class that has the same name and argument list.

If you could not override superclass methods, you could always create unique names for each subclass method, such as getRateOfPayForHourly() and getRateOfPayForContractual(), but the classes you create are easier to write and understand if you use one reasonable name for methods that do essentially the same thing. Because you are attempting to get the rate of pay for each object, getRateOfPay() is an excellent method name for all three object types.

Tip: You already have overridden methods in your Swing applets. When you write your own init() or start() method within a Swing applet, you are overriding the automatically supplied superclass version you get when you use the phrase `extends JApplet`.

Tip: If a superclass and its subclass have methods with the same name but different argument lists, you are overloading methods, and not overriding them. You learned about overloading methods in Chapter 4.

Next you will create two methods with the same name, printHeader(), with one version in the Event superclass and another in the DinnerEvent subclass. When you call the printHeader() method, the correct version will execute based on the object you use.

To demonstrate that subclass methods override superclass methods with the same name:

1. In your text editor, open the **Event.java** file in the Chapter.11 folder on your Student Disk. Change the class name from Event to **EventWithHeader** because this new class will contain a method that allows you to print a header line of explanatory text with each class object. Save the file as **EventWithHeader.java** in the Chapter.11 folder on your Student Disk. In addition to providing a descriptive name, changing the class name serves another purpose. By giving the class a new name, you retain the original class on your disk so you can study the differences later.
2. Position the insertion point at the end of the line that contains the closing curly brace for the printEventGuests() method, and then press **[Enter]** to start a new line.
3. Enter the following printHeader() method:

```
public void printHeader()
{
 System.out.println("Simple event: ");
}
```

4. Save the file, and then compile it.
5. Open the **DinnerEvent.java** file in the Chapter.11 folder. Change the class name from DinnerEvent to **DinnerEventWithHeader**, and change the class from which it extends—`Event`—to `EventWithHeader`. Save the file as **DinnerEventWithHeader.java** in the Chapter.11 folder on your Student Disk.
6. Position your insertion point at the end of the line that contains the closing curly brace for the printDinnerChoice() method, and then press **[Enter]** to start a new line of text. Add the following printHeader() method to this class:

```
public void printHeader()
{
 System.out.println("Dinner event: ");
}
```

7. Save the file, and then compile it.

You just created a DinnerEventWithHeader class that contains a printHeader() method. Then you extended the class by creating a DinnerEvent subclass containing a method with the same name. Now you will write a program that demonstrates that the correct method executes depending on the object.

To create a program that demonstrates that the correct printHeader() method executes depending on the object:

1. Open a new file in your text editor, and then enter the first few lines of a UseEventWithHeader class:

```
public class UseEventWithHeader
{
 public static void main(String[] args) throws Exception
 {
```

2. Enter the following code to create two objects, EventWithHeader and DinnerEventWithHeader:

```
EventWithHeader anEvent = new EventWithHeader();
DinnerEventWithHeader aDinnerEvent =
new DinnerEventWithHeader();
```

3. Enter the following code to call the three EventWithHeader methods:

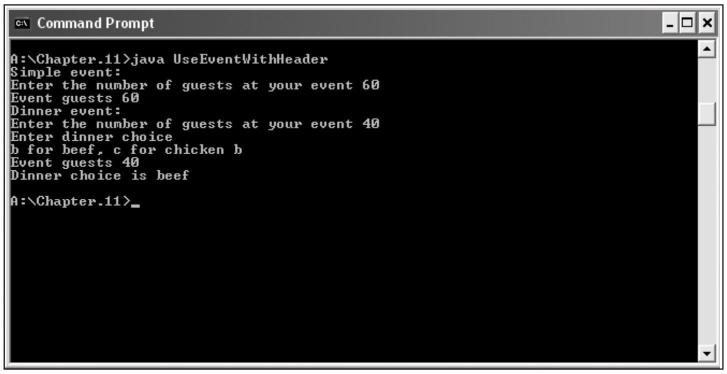
```
anEvent.printHeader();
anEvent.setEventGuests();
anEvent.printEventGuests();
```

4. Enter the following code to call the five DinnerEventWithHeader methods, and then add the closing curly brace for the class:

```
aDinnerEvent.printHeader();
aDinnerEvent.setEventGuests();
aDinnerEvent.setDinnerChoice();
aDinnerEvent.printEventGuests();
```

```
        aDinnerEvent.printDinnerChoice();
      }
    }
```

5. Save the file as **UseEventWithHeader.java** in the Chapter.11 folder on your Student Disk. Compile and run the program. Input the values shown in Figure 11-9. Be sure to notice that for each type of object, the correct method executes.



```
A:\Chapter.11>java UseEventWithHeader
Simple event:
Enter the number of guests at your event 60
Event guests 60
Dinner event:
Enter the number of guests at your event 40
Enter dinner choice
b for beef, c for chicken b
Event guests 40
Dinner choice is beef

A:\Chapter.11>_
```

Figure 11-9 Output of the UseEventWithHeader program

Working with Superclasses that Have Constructors

When you create any object, as in `SomeClass anObject = new SomeClass();`, you are calling a class constructor method that has the same name as the class itself. When you instantiate an object that is a member of a subclass, you are actually calling at least two constructors: the constructor for the base class and the constructor for the extended, derived class. When you create any subclass object, the superclass constructor must execute first, and *then* the subclass constructor executes.

In the examples of inheritance so far in this chapter, each class contained default constructors, so their execution was transparent. However, you should realize that when you create an object using `HourlyEmployee clerk = new HourlyEmployee();` (where HourlyEmployee is a subclass of Employee), *both* the Employee() and HourlyEmployee() constructors execute.

You can create a class whose constructor does nothing but print a message. Then, when you extend the class, you can create a subclass constructor that prints a different message. When the program is run, both classes will print messages confirming that the constructor in each class is called.

To demonstrate that a subclass constructor calls the superclass constructor first:

1. Open a new file in your text editor.

2. Create the following superclass, whose constructor prints a message on the screen:

```
public class ASuperClass
{
 public ASuperClass()
 {
  System.out.println("In superclass constructor");
 }
}
```

3. Save the file as **ASuperClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
4. Open a new file in your text editor, and then enter the following program to create a derived subclass that extends the superclass. The constructor of the subclass also prints a message.

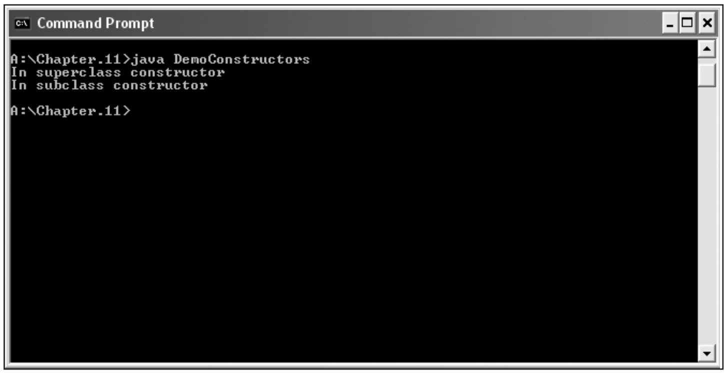
```
public class ASubClass extends ASuperClass
{
 public ASubClass()
 {
  System.out.println("In subclass constructor");
 }
}
```

5. Save the file as **ASubClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
6. Open a new text file and enter the following program that does just one thing—it creates a child class object:

```
public class DemoConstructors
{
 public static void main(String[] args)
 {
  ASubClass child = new ASubClass();
 }
}
```

7. Save this file as **DemoConstructors.java** in the Chapter.11 folder on your Student Disk. Compile the file, and then execute the program. The output appears as shown in Figure 11-10. Even though you create only one subclass object, two separate messages print—one from the superclass constructor and one from the subclass constructor.

Of course, most constructors perform many more tasks than printing a message to inform you that they exist. When constructors initialize variables, you usually want the superclass constructor to take care of initializing the data fields that originate in the superclass. The subclass constructor only needs to initialize the data fields that are specific to the subclass.



```
A:\Chapter.11>java DemoConstructors
In superclass constructor
In subclass constructor

A:\Chapter.11>
```

Figure 11-10 Output of the DemoConstructors program

Next you will add a constructor to the Event class you created for Event Handlers Incorporated. When you instantiate a subclass DinnerEvent object, the superclass Event constructor will execute.

To add a constructor to the Event class:

1. Open the **EventWithHeader.java** file in your text editor. Use your text editor's Save As command to save the file as **EventWithConstructor.java** in the Chapter.11 folder on your Student Disk. Be sure to change the class name from EventWithHeader to **EventWithConstructor**.

2. Position your insertion point to the right of the statement that declares the eventGuests data field, and then press **[Enter]** to start a new line. Type the following constructor that initializes the number of guests to zero:

   ```
   public EventWithConstructor()
   {
    eventGuests = 0;
   }
   ```

3. Save the file and compile it.
4. In your text editor, open the **DinnerEventWithHeader.java** file from the Chapter.11 folder. Change the class header so that both the class name and the parent class name read as follows:

   ```
   public class DinnerEventWithConstructor
    extends EventWithConstructor
   ```

5. Save the file as **DinnerEventWithConstructor.java** in the Chapter.11 folder on your Student Disk, and then compile it.
6. In your text editor, open a new file so you can write a program to demonstrate the use of the base class constructor with both a base class object and an extended class object. To begin the class, you create a class header, a main()

method header, and definitions of two objects—one is a member of the base class, and the other is a member of the extended class:

```
public class UseEventsWithConstructors
{
 public static void main(String[] args)
 {
  EventWithConstructor anEvent =
   new EventWithConstructor();
  DinnerEventWithConstructor aDinnerEvent =
   new DinnerEventWithConstructor();
```

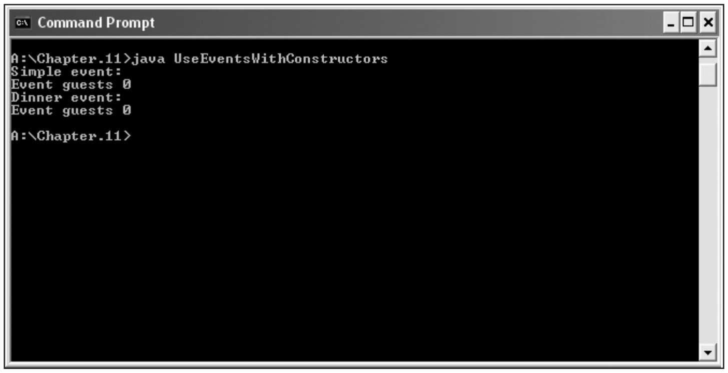
7. Add the following statements that print a header and the number of guests for the parent class member:

```
  anEvent.printHeader();
  anEvent.printEventGuests();
```

8. Add statements that print a header and the number of guests for the child class member, and then add a closing curly brace for the class:

```
  aDinnerEvent.printHeader();
  aDinnerEvent.printEventGuests();
  }
}
```

9. Save the program as **UseEventsWithConstructors.java** in the Chapter.11 folder on your Student Disk. Then compile and run the program. The output is shown in Figure 11-11. The guest number is initialized correctly for objects of both classes.



```
A:\Chapter.11>java UseEventsWithConstructors
Simple event:
Event guests 0
Dinner event:
Event guests 0

A:\Chapter.11>
```

Figure 11-11 Output of the UseEventsWithConstructors program

USING SUPERCLASS CONSTRUCTORS THAT REQUIRE ARGUMENTS

When you create a class and do not provide a constructor, Java automatically supplies you with one that never requires arguments. When you write your own constructor, you replace the automatically supplied version. Depending on your needs, the constructor you create for a class might require arguments. When you use a class as a superclass, and the class has a constructor that requires arguments, then you must make sure that any subclasses provide the superclass constructor with what it needs.

Tip

Don't forget that a class can have many constructors. As soon as you create at least one constructor for a class, you no longer can use an automatic version.

When a superclass constructor requires arguments, you must include a constructor for each subclass you create. Your subclass constructor can contain any number of statements, but the first statement within the constructor must call the superclass constructor. Even if you have no other reason to create a subclass constructor, you must write the subclass constructor so it can call its superclass' constructor.

The format of the statement that calls a superclass constructor is **`super(list of arguments);`**. The keyword **`super`** always refers to the superclass of the class in which you use it. Suppose that you create an Employee class with a constructor that requires three arguments—a character, a double, and an integer—and you create an HourlyEmployee class that is a subclass of Employee. The following code shows a valid constructor for HourlyEmployee:

```
public HourlyEmployee()
{
 super('P', 12.35, 40);
 // Other statements can go here
}
```

The HourlyEmployee constructor requires no arguments, but it passes three arguments to its superclass constructor. A different HourlyEmployee constructor can require arguments. It could then pass the appropriate arguments to the superclass constructor. For example:

```
public HourlyEmployee(char dept, double rate, int hours)
{
 super(dept, rate, hours);
 // Other statements can go here
}
```

Tip

Except for any comments, the super() statement must be the first statement in the subclass constructor. Not even data field definitions can precede it.

Tip

Although it seems that you should be able to use the superclass constructor name to call the superclass constructor, Java does not allow this. You must use the keyword `super`.

Next you will modify the Event class so that its constructor requires arguments. Then, to show how the superclass and subclass constructors work, you will create a subclass object that calls its superclass constructor.

To demonstrate how inheritance works when class constructors require arguments:

1. Open the **EventWithConstructor.java** file in your text editor, and then change the class name to **EventWithConstructorArg**.
2. Change the current constructor name so it matches the class name, and then change the constructor argument list so it requires an integer argument. In other words, change `public EventWithConstructor()` to **`public EventWithConstructorArg(int guests)`**.
3. Change the constructor statement that sets eventGuests to zero as follows so that it sets the event guests field to the constructor's argument value: **`eventGuests = guests;`**.
4. Save the file as **EventWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Next you will add a constructor to the DinnerEvent class so it can call its parent's constructor. The child class constructor requires an integer argument, which it then passes to the parent class constructor.

To create the child class:

1. Open the **DinnerEventWithConstructor.java** file in your text editor.
2. Change the class header as follows so that the name of this class is **DinnerEventWithConstructorArg**, and thus inherits from EventWithConstructorArg:

```
public class DinnerEventWithConstructorArg
 extends EventWithConstructorArg
```

3. Position your insertion point after the declaration of the dinnerChoice field, and then press **[Enter]** to start a new line. Create the following constructor that requires an integer argument and passes it to the superclass constructor:

```
public DinnerEventWithConstructorArg(int guests)
{
 super(guests);
}
```

4. Save the file as **DinnerEventWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Now you can create a program to demonstrate creating parent and child class objects when the parent constructor needs an argument.

To create the program:

1. Open a new file in your text editor, and then enter the following first few lines of a program that demonstrates using super():

```
public class UseEventsWithConstructorArg
{
 public static void main(String[] args)
 {
```

2. Enter the following code to create an EventWithConstructorArg object and give it a value for the number of guests:

```
EventWithConstructorArg anEvent =
  new EventWithConstructorArg(45);
```

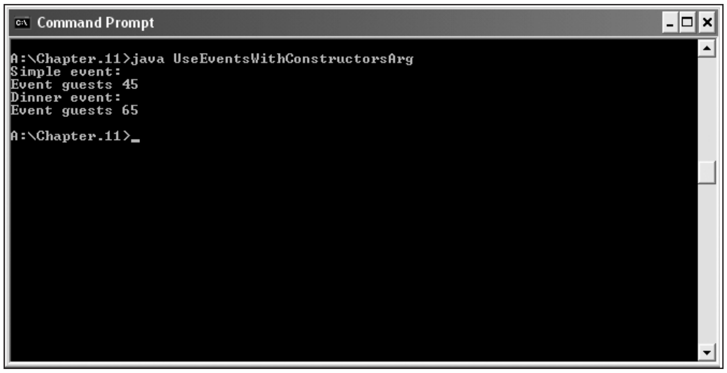
3. Add the following code to create a DinnerEventWithConstructorArg object. This constructor also requires an integer argument.

```
DinnerEventWithConstructorArg aDinnerEvent =
  new DinnerEventWithConstructorArg(65);
```

4. Add the following statements to print explanations and guest fields for each object, and then add the closing curly brace for the class:

```
 anEvent.printHeader();
 anEvent.printEventGuests();
 aDinnerEvent.printHeader();
 aDinnerEvent.printEventGuests();
 }
}
```

5. Save the file as **UseEventsWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, then compile and execute the program. The output appears in Figure 11-12. Each object is correctly initialized because the superclass constructor was correctly called in each case.



```
A:\Chapter.11>java UseEventsWithConstructorsArg
Simple event:
Event guests 45
Dinner event:
Event guests 65

A:\Chapter.11>_
```

Figure 11-12 Output of the UseEventsWithConstructorsArg program

Accessing Superclass Methods

Earlier in this chapter, you learned that a subclass could contain a method with the same name and arguments as a method in its parent class. When this happens, using the subclass method overrides the superclass method. However, you might want to use the superclass method within a subclass. If so, you can use the keyword `super` to access the parent class method. To demonstrate, you will create a simple subclass that has a method with the same name as a method that is part of its superclass.

To access a superclass method from within a subclass:

1. Open a new file in your text editor, then create the following parent class with a single method:

```
public class AParentClass
{
 private int aVal;
 public void printClassName()
 {
  System.out.println("AParentClass");
 }
}
```

2. Save the file as **AParentClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.

3. Open a new text file, then create the following child class that inherits from the parent. The child has one method. The method has the same name as the parent's method, but the child can call the parent's method without conflict by using the keyword `super`.

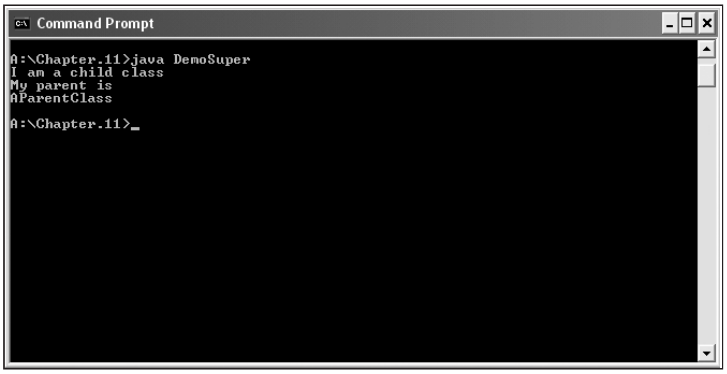
```
public class AChildClass extends AParentClass
{
 public void printClassName()
 {
  System.out.println("I am AChildClass");
  System.out.println("My parent is ");
  super.printClassName();
 }
}
```

4. Save the file as **AChildClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.

5. Finally, open a new text file and enter the following demonstration program to show that the child class can call its parent's method:

```
public class DemoSuper
{
  public static void main(String[] args)
  {
   AChildClass child = new AChildClass();
   child.printClassName();
  }
 }
```

6. Save the file as **DemoSuper.java** in the Chapter.11 folder on your Student Disk, then compile and execute the program. As the output in Figure 11-13 shows, even though the child and parent classes have methods with the same name, the child class can use the parent class method correctly by employing the keyword `super`.



```
A:\Chapter.11>java DemoSuper
I am a child class
My parent is
AParentClass

A:\Chapter.11>_
```

Figure 11-13 Output of the DemoSuper program

Tip: You can use the keyword `this` as the opposite of `super`. For example, if a superclass and its subclass each have a method named someMethod(), then within the subclass, super.someMethod() refers to the superclass version of the method. Both someMethod() and this.someMethod() refer to the subclass version.

Learning About Information Hiding

The AStudent class shown in Figure 11-14 is a typical construction for Java classes. The keyword `private` precedes each data field, and the keyword `public` precedes each method. As a matter of fact, the four get and set methods are necessary within the AStudent class specifically because the data fields are `private`. Without the `public` get and set methods, there would be no way to access these private data fields.

```
public class AStudent
{
  private int idNum;
  private double semesterTuition;
  public int getIdNum()
  {
    return idNum;
  }
  public double getTuition()
  {
    return semesterTuition;
  }
  public void setIdNum(int num)
  {
    idNum = num;
  }
  public void setTuition(double amt)
  {
    semesterTuition = amt;
  }
}
```

Figure 11-14 AStudent class

When a program is a class user of AStudent (that is, it instantiates the AStudent object), then the user cannot directly alter the data in any `private` field. For example, when you write a main() method that creates a AStudent as `AStudent someStudent = new AStudent();`, you cannot change the AStudent's idNum with a statement such as `someStudent.idNum = 812;`. The idNum of the someStudent object is not accessible in the main() program that uses the AStudent object because idNum is `private`. Only methods that are part of the AStudent class itself are allowed to alter AStudent data. To alter an AStudent's idNum, you must use the `public` method setIdNum(), as in `someStudent.setIdNum(812);`.

The concept of keeping data private is known as **information hiding**. When you employ information hiding, your data can be altered only by the methods you choose and only in ways that you can control. For example, you might want the setIdNum() method to check to make sure the idNum is within a specific range of values. If a class other than the AStudent class itself could alter idNum, then idNum could be assigned a value that the AStudent class couldn't control.

Tip: You first learned about information hiding and using the `public` and `private` keywords in Chapter 3. You may want to review these concepts.

When a class serves as a superclass to other classes you create, your subclasses inherit all the data and methods of the superclass, with one exception: `private` members of the parent class are not inherited. If you could use `private` data outside its class, you would use the advantages of information hiding. If you intend the AStudent class data field idNum to be `private`, then you don't want any outside classes using the field. If a new class could simply extend your AStudent class and get to its data fields without going through the proper channels, then information hiding would not be operating.

There are occasions when you want to access parent class data from within a subclass. For example, suppose you create two child classes that extend the AStudent class: PartTimeStudent and FullTimeStudent. If you want the subclass methods to be able to access idNum and semesterTuition, then those data fields cannot be `private`. However, if you don't want other, nonchild classes to access those data fields, then they cannot be `public`. To solve this problem, you can create the fields using the modifier `protected`. Using the keyword **`protected`** provides you with an intermediate level of security between public and private access. If you create a protected data field or method, it can be used within its own class or in any classes extended from that class, but it cannot be used by "outside" classes. In other words, protected members are those that can be used by a class and its descendents.

Next you will create a superclass with a protected field using the `protected` access modifier with a superclass data field so that you can access the field within a subclass method. Then you will create a method to set the number of event guests, and create a simple program to test this class.

To create a superclass with a protected field:

1. In your text editor, open the **EventWithHeader.java** file. For simplicity, you will use the file that you created before adding constructors. Change the class name to **EventWithProtectedData**.
2. Change the modifier on the eventGuests field from `private` to **`protected`**.
3. Save the file as **EventWithProtectedData.java** in the Chapter.11 folder on your Student Disk, and then compile it.
4. Open the **DinnerEventWithHeader.java** file. Change its name and its parent's name so the class header reads as follows:

   ```
   public class DinnerEventWithProtectedData
    extends EventWithProtectedData
   ```

   Assume that Event Handlers Incorporated requires at least 10 guests for an event with dinner, but there is no minimum guest number for other event types. To ensure that DinnerEvents (unlike plain Events) have at least 10 guests, the subclass setEventGuests() method will override the setEventGuests() method in the superclass. The subclass version of the method will call the superclass method, but if the user does not enter a guest number of at least 10, the subclass method will call the superclass method again.

5. To create the subclass setEventGuests() method, position your insertion point at the end of the closing curly brace of the setDinnerChoice() method, press **[Enter]** to start a new line, and then type the following header for the method and the method's opening curly brace:

```
public void setEventGuests() throws Exception
{
```

6. Call the superclass method with the same name:

```
super.setEventGuests();
```

7. Check the value of the eventGuests data field using the following `while` loop. Because the field is `protected` in the base class, it can be accessed here in the derived class. If the guest number continues to be too low, issue an error message and call the superclass method.

```
while(eventGuests < 10)
{
 System.out.print("Minimum required for dinner: ");
 System.out.println("10 guests!");
 super.setEventGuests();
}
```

Tip

If eventGuests had not been inherited (that is, if it was still `private`), you would need to use `public getEventGuests()` to access its value.

8. Add the closing curly brace for the method, save the file as **DinnerEventWithProtectedData.java** in the Chapter.11 folder on your Student Disk, and then compile it.

9. Next, create a simple program to test this class. Open a new file in your text editor and then enter the following first few lines of a demonstration program:

```
public class UseProtected
{
 public static void main(String[] args)
  throws Exception
 {
```

10. Create the aDinnerEvent object by entering the following code:

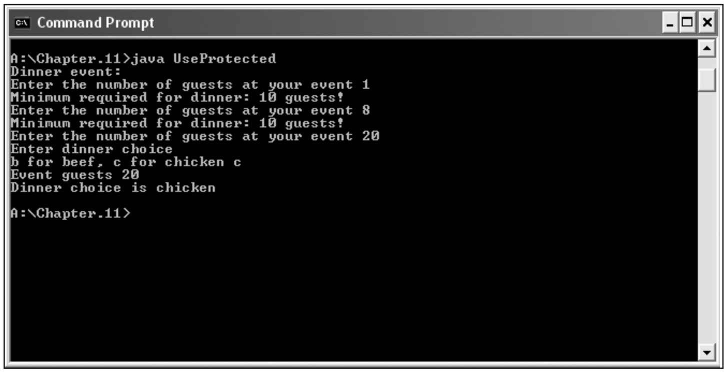
```
DinnerEventWithProtectedData aDinnerEvent =
 new DinnerEventWithProtectedData();
```

11. Using the newly created object, print an explanation, set the field values, then print the field values by entering the following code:

```
aDinnerEvent.printHeader();
aDinnerEvent.setEventGuests();
aDinnerEvent.setDinnerChoice();
aDinnerEvent.printEventGuests();
aDinnerEvent.printDinnerChoice();
```

12. Add the closing curly brace for the main() method and the closing curly brace for the class, and then save the file as **UseProtected.java** in the Chapter.11 folder on your Student Disk.

13. Compile and execute the program. When you run the program, make several attempts to set the number of dinner guests to values below 10. The program will continue to prompt you until your guest number meets the required minimum. A sample program output appears in Figure 11-15.



```
A:\Chapter.11>java UseProtected
Dinner event:
Enter the number of guests at your event 1
Minimum required for dinner: 10 guests!
Enter the number of guests at your event 8
Minimum required for dinner: 10 guests!
Enter the number of guests at your event 20
Enter dinner choice
b for beef, c for chicken c
Event guests 20
Dinner choice is chicken

A:\Chapter.11>
```

Figure 11-15 Sample output from the UseProtected program

Using Methods You Cannot Override

There are four types of methods that you cannot override in a subclass:

- `private` methods
- `static` methods
- `final` methods
- Methods within `final` classes

You already know that when you create a `private` variable in a superclass, the variable is not available for use in a subclass. Similarly, if you create a `private` method in a superclass, the method is not available for use in any class extended from the superclass. You also know that a subclass can access `private` variables in the superclass through the use of nonprivate methods. Again, similarly, if a superclass has a nonprivate method that accesses a `private` method, then a child class can use the inherited nonprivate method to access the noninherited `private` method.

For example, Figure 11-16 shows a superclass named Super that contains two methods: one method is `public` and the other method is `private`.

```
public class Super
{
  public void printPublic()
  {
    System.out.println("This method is public");
    printPrivate();
  }
  private void printPrivate()
  {
    System.out.println("This method is private");
  }
}
```

Figure 11-16 Super class

Figure 11-17 shows a subclass that attempts to extend Super. If you compile the Sub class, you will receive the error message, "No method matching printPrivate() found in class Sub". The statement that calls printPublic() works correctly because as a `public` method, printPublic() is inherited by the Sub class. The printPublic() method calls printPrivate(), which is perfectly legal because printPublic() and printPrivate() are methods within the same class. However, within the Sub class, because the printPrivate() method is "invisible", the compiler tells you that it doesn't exist. As a `private` method, printPrivate() is not inherited.

```
public class Sub extends Super
{
  public void printMessages()
  {
    printPublic();
    printPrivate();
  }
}
```

Figure 11-17 Incorrect Sub class with a printPrivate() method

Additionally, you cannot override a method that is `private` in a parent class. For example, examine the class in Figure 11-18. The Sub class extends the Super class shown in Figure 11-16 and adds its own printPrivate() method. When a main() program creates a Sub object, as in `Sub aSubObject = new Sub();`, and calls printPrivate() with `aSubObject.printMessages();`, then the screen will show "This method is private" from the Super class, and not "This will never print" from the Sub class method. This makes sense when you consider that printPublic() is a method that is part of the Super class. The printPublic() method calls the printPrivate() method from its own class. The subclass shown in Figure 11-18 compiles without error, but does so as if the printPrivate() method within the subclass does not exist.

```
public class Sub extends Super
{
  public void printMessages()
  {
    printPublic();
  }
  private void printPrivate()
  {
    System.out.println("This will never print");
  }
}
```

Figure 11-18 Sub class with a invisible method

In addition to `private` methods, you also cannot override `static` methods from a parent class. You learned in Chapter 4 that `static` methods are also called class methods and that they have no objects associated with them. You call a `static` method using the class name, not an object name. Recall that in Java the keyword `static` implies uniqueness; a `static` method is unique to the base class and all its descendants.

Tip: The main() methods in Java applications are always `static`.

Methods carrying the access modifier `final` cannot be overridden by subclass methods. For example, the statement `public final void getEmpNum()` is declared `final` by placing the modifier in the class declaration. If getEmpNum() is an automatically available method in a Java program, any attempt to create a method with the same name will result in an error. In Chapter 4, you learned that you can use the keyword `final` when you want to create a constant, as in `final double TAXRATE = .065;`. You can also use the `final` modifier with methods when you don't want the method to be overridden. You use `static` as a method access modifier when you create class methods for which you want to prevent overriding; you use `final` as a method access modifier when you create instance methods for which you want to prevent overriding.

Tip: You can think of `private` and `static` methods as being implicitly `final`.

Because a `final` method's definition can never change, the compiler optimizes the program by removing the calls to `final` methods and replacing them with the expanded code of their definitions at each method call location. This process is called **inlining** the code. You are never aware that inlining is taking place; the compiler chooses to use this procedure to save the overhead of calling a method.

Tip

The compiler chooses to inline a `final` method only if it is a small method containing just one or two lines of code.

Finally, you can declare a class to be `final`. When you do so, all of its methods are `final`, regardless of which access modifier precedes the method name. A `final` class cannot be a parent.

Tip

Java's Math class, which you learned about in Chapter 4, is an example of a `final` class.

CHAPTER SUMMARY

- Inheritance is the principle that asserts that you can apply knowledge of a general category to more-specific objects. The classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class through inheritance, you are automatically provided with data fields and methods.
- A class that is used as a basis for inheritance is called a base class, a superclass, or a parent class. When you create a class that inherits from a base class, it is called a derived class, a subclass, or a child class. In general, a subclass is larger than a superclass because you add new, more-specific fields and methods to a subclass, as well as inherit fields and methods from the superclass.
- If you create separate methods for each subclass of a superclass, then each class's objects will use the appropriate method for that class. Each child class method overrides the method that has the same name in the parent class. Using the same method name to indicate different implementations is called polymorphism.
- When you instantiate an object that is a member of a subclass, you are actually calling two constructors: the constructor for the superclass and the constructor for the subclass. The base class constructor executes first, and then the subclass constructor executes.
- When constructors initialize variables, you usually want the base class constructor to take care of initializing the data fields that originate in the base class. The derived class constructor must initialize only the data fields that are specific to the derived class.
- When a superclass constructor requires arguments, you must create a constructor for each subclass you create. The first statement within the constructor must call the superclass constructor. The format of the statement that calls a superclass constructor is `super(list of arguments);`. The keyword `super` always refers to the superclass of the class in which you use it.

- When a program is a class user, it cannot directly alter the data in any `private` field. The concept of keeping data private is known as information hiding. Information hiding lets you control how data is used and altered.
- If you create a data field or method that uses the `protected` access modifier, then the field or method can be used within its own class or in any classes extended from that class, but cannot be used by "outside" classes.
- There are four types of methods that you cannot override in a subclass: `private`, `static`, `final`, and methods within `final` classes.
- If you create a `private` method in a superclass, the method is not available for use in any class extended from the superclass. If a superclass has a nonprivate method that accesses a `private` method, then a child class can use the inherited nonprivate method to access the noninherited private method.
- When you declare a class to be `final`, all of its methods are `final`, regardless of which access modifier precedes the method name. A `final` class cannot be a parent.

Review Questions

1. ________________ as an alternate way(s) to discover which of two classes is the base class or subclass.
   a. Look at the class size
   b. Try saying the two class names together
   c. Use polymorphism
   d. Both a and b are correct.
2. Employing inheritance reduces errors because ________________.
   a. the new classes have access to fewer data fields
   b. the new classes have access to fewer methods
   c. you can copy methods that you already created
   d. many of the methods you need have already been used and tested
3. A base class can also be called a ________________.
   a. child class
   b. subclass
   c. derived class
   d. superclass

4. Which of the following choices most closely describes a parent class/child class relationship?
   a. Rose/Flower
   b. Present/Gift
   c. Dog/Poodle
   d. Sparrow/Bird
5. The Java keyword that creates inheritance is ________________.
   a. `static`
   b. `enlarge`
   c. `extends`
   d. `inherits`
6. A class named Building has a method named getFloors(). If School is a child class of Building, and ModelHigh is an object of type School, then which of the following statements is valid?
   a. `Building.getFloors();`
   b. `School.getFloors();`
   c. `ModelHigh.getFloors();`
   d. All of the above statements are valid.
7. Which of the following statements is false?
   a. A child class inherits from a parent class.
   b. A parent class inherits from a child class.
   c. Both of the above statements are false.
   d. Neither of the above statements is false.
8. When a subclass method has the same name and argument types as a superclass method, the subclass method can ________________ the superclass method.
   a. override
   b. overuse
   c. overload
   d. overcompensate
9. When you instantiate an object that is a member of a subclass, the ________________ constructor executes first.
   a. subclass
   b. child class
   c. extended class
   d. parent class

10. The keyword `super` always refers to the ________________ of the class in which you use it.
    a. child class
    b. derived class
    c. sub class
    d. parent class
11. If a superclass constructor requires arguments, then its subclass ________________.
    a. must contain a constructor
    b. must not contain a constructor
    c. must contain a constructor that requires arguments
    d. must not contain a constructor that requires arguments
12. If a superclass constructor requires arguments, any constructor of its subclasses must call the superclass constructor ________________.
    a. as the first statement
    b. as the last statement
    c. at some time
    d. multiple times if multiple arguments are involved
13. A child class Motorcycle extends a parent class Vehicle. Each class constructor requires one String argument. The Motorcycle class constructor can call the Vehicle class constructor with the statement ________________.
    a. `Vehicle("Honda");`
    b. `Motorcycle("Harley");`
    c. `super("Suzuki");`
    d. none of the above
14. In the Java programming language, the concept of keeping data private is known as ________________.
    a. polymorphism
    b. information hiding
    c. data deception
    d. concealing fields
15. If you create a data field or method that is ________________, it can be used within its own class or in any classes extended from that class.
    a. `public`
    b. `protected`
    c. `private`
    d. none of the above

16. Within a subclass, you cannot override ________________ methods.
    a. `public`
    b. `private`
    c. `protected`
    d. constructor
17. You call a `static` method using a(n) ________________ name.
    a. class
    b. superclass
    c. object
    d. none of the above
18. You use `final` as a method access modifier when you create ________________ methods for which you want to prevent overriding.
    a. class
    b. superclass
    c. subclass
    d. instance
19. A compiler can decide to ________________ a `final` method.
    a. duplicate
    b. inline
    c. redline
    d. beeline
20. You use ________________ as a method access modifier when you create class methods for which you want to prevent overriding.
    a. `final`
    b. `static`
    c. `private`
    d. `public`

EXERCISES

1. Create a class named Book that contains data fields for title and number of pages. Include get and set methods for these fields. Next create a subclass named Textbook, which contains an additional field that holds a grade level for the Textbook, and additional methods to get and set the grade level field. Write a program that demonstrates using objects of each class. Save the programs as **Book.java**, **Textbook.java**, and **DemoBook.java** in the Chapter.11 folder on your Student Disk.

2. Create a class named Square that contains data fields for height, width, and surfaceArea, and a method named computeSurfaceArea(). Create a child class named Cube. Cube contains an additional data field named depth, and a computeSurfaceArea() method that overrides the parent method. Write a program that instantiates a Square object and a Cube object and displays the surface areas of the objects. Save the programs as **Cube.java**, **Square.java**, and **Demo Square.java** in the Chapter.11 folder on your Student Disk.
3. Create a class named Order that performs order processing of a single item. The superclass has four fields: customer name, customer number, quantity ordered, and unit price. Include set and get methods for each field. This class also needs methods to compute the total price (quantity times unit price) and to display the fields. Create a subclass that overrides computePrice() by adding a shipping and handling charge of $4.00. Write a program that uses these classes. Save the programs as **Order.java**, **HandlingShipping.java**, and **UseHandlingShipping.java** in the Chapter.11 folder on your Student Disk.
4. Create a class named Vacation that computes the amount of vacation time an employee gets. If an employee has worked for more than five years, the total vacation time is three weeks annually; otherwise the employee gets two weeks annually. Use a superclass that contains an integer field that holds the number of vacation weeks, and get and set methods for the field. Use a superclass object for employees who have earned two weeks of vacation. Use a subclass for three-weeks vacation. Obtain the employee information from text fields in an applet. Use text boxes for employee number, employee name, and number of years. Display the number of weeks of vacation after computing the result. Save the programs as **Vacation.java**, **ExtraVacation.java**, **VacationHome.java**, **UseVacation.java**, and **TestUseVacation.html** in the Chapter.11 folder on your Student Disk.
5. a. Create a class named Year that contains a data field that holds the number of days in the year. Include a get method that displays the number of days and a constructor that sets the number of days to 365. Create a subclass named LeapYear. LeapYear's constructor overrides Year's constructor and sets the day field to 366. Write a program that instantiates one object of each class and displays their data. Save the programs as **Year.java**, **LeapYear.java**, and **UseYear.java** in the Chapter.11 folder on your Student Disk.

   b. Add a method named daysElapsed() to the Year class you created in Exercise 5a. The daysElapsed() method accepts two arguments representing a month and a day; it returns an integer indicating the number of days that have elapsed since January 1 of the year. Create a daysElapsed() method for the LeapYear class that overrides the method in the Year class. Write a program that calculates the days elapsed on March 1 for a Year and for a LeapYear. Save the programs as **Year2.java**, **LeapYear2.java**, and **UseYear2.java** in the Chapter.11 folder on your Student Disk.

6. Create a class named Computer that contains data fields for processor model (for example, Pentium III) and clock speed in gigahertz (for example, 1.6). Include a get method for each field and a constructor that requires a parameter for each field. Create a subclass named MultimediaComputer that contains an additional integer field for the CD-ROM speed. The MultiMedia class also contains a get method for the new data field and a constructor that requires arguments for each of the three data fields. Write a program to demonstrate creating and using an object of each class. Save the programs as **Computer.java**, **MultimediaComputer.java**, and **UseComputer.java** in the Chapter.11 folder on your Student Disk.
7. Create a class named HotelRoom that includes an integer field for the room number and a double field for the nightly rental rate. Include get methods for these fields, and a constructor that requires an integer argument representing the room number. The constructor sets the room rate based on the room number; rooms numbered 299 and below are $69.95 per night, others are $89.95 per night. Create an extended class named Suite whose constructor requires a room number and adds a $40.00 surcharge to the regular hotel room rate based on the room number. Write a program to demonstrate creating and using an object of each class. Save the programs as **HotelRoom.java**, **Suite.java**, and **UseHotelRoom.java** in the Chapter.11 folder on your Student Disk.
8. Create a class named Package with data fields for weight in ounces, shipping method, and shipping cost. The shipping method is a character: *A* for air, *T* for truck, or *M* for mail. The Package class contains a constructor that requires arguments for weight and shipping method. The constructor calls a calculateCost() method that determines the shipping cost based on the following:

Shipping Method ($)

Weight (lb)	Air	Truck	Mail
1 to 8	2.00	1.50	.50
9 to 16	3.00	2.35	1.50
17 and over	4.50	3.25	2.15

The Package class also contains a display() method that displays the values in all four fields. Create a subclass named **InsuredPackage** that adds an insurance cost to the shipping cost based on the following:

Shipping Cost Before Insurance ($)	Additional Cost ($)
0 to 1.00	2.45
1.01 to 3.00	3.95
3.01 and over	5.55

Write a program that instantiates at least three objects of each type (Package and InsuredPackage) using a variety of weights and shipping method codes. Display the results for each Package and InsuredPackage. Save the programs as **Package**, **InsuredPackage**, and **UsePackage** in the Chapter.11 folder on your Student Disk.

9. Write a program named CarRental that computes the cost of renting a car for a day, based on the size of the car: economy, medium, or full size. Include a constructor that requires the car size. Add a subclass to add the option of a car phone. Write a program to use these classes. Save the programs as **CarRental**, **CarPhone**, and **UseCarRentalAndPhone** in the Chapter.11 folder on your Student Disk.
10. Write a program named CollegeCourse that computes the cost of taking a college course. Include a constructor that requires a course ID number. Add a subclass to compute a lab fee for a course that uses a lab. Write a program to use these classes. Save the programs as **CollegeCourse**, **Lab**, and **UseCourse** in the Chapter.11 folder on your Student Disk.
11. Write a program named Discount that computes the price of an item. Include a constructor that requires the quantity, item name, and item number. Add a subclass to provide a discount based on the quantity ordered. Write a program to use these classes. Save the programs as **Discount**, **ComputeDiscount**, and **UseDiscount** in the Chapter.11 folder on your Student Disk.
12. Write a program named Vehicle that acts as a superclass for vehicle types. The Vehicle class must contain private variables for the number of wheels and the average number of miles per gallon. The Vehicle class must also contain a constructor with integer arguments for the number of wheels and range in miles, and a toString() method to return the number of wheels and range in miles when called. Write subclass programs Car and MotorCycle that extend the Vehicle class. Each subclass should contain a `private static final` integer variable that sets the number of wheels for the subclass and a `private` variable to set the number of passengers. Each subclass must have a toString() method for returning the number of wheels, range, and passengers for the vehicle type. Write a UseVehicle class to instantiate the two vehicle objects and print the object's values. Save the programs as **Vehicle**, **Car**, **MotorCycle**, and **UseVehicle** in the Chapter.11 folder on your Student Disk.
13. Create a class named Course that contains data fields for a course title and a boolean variable that indicates whether the class is offered online. Include get and set methods for these fields. Next create a subclass named OnLine, which contains an additional field that holds a grade level for students eligible to sign up for a course, and additional methods to get and set the grade level field. Write a program that demonstrates using objects of each class. Save the programs as **Course**, **OnLine**, and **DemoCourse** in the Chapter.11 folder on your Student Disk.

14. Each of the following files in the Chapter.11 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugEleven1.java will become FixDebugEleven1.java.
    a. DebugEleven1.java
    b. DebugEleven2.java
    c. DebugEleven3.java
    d. DebugEleven4.java

CASE PROJECT

Wei's Tea Shop asks you to write an Inventory control program to keep track of the boxes of tea they serve to customers. The program will allow the user to enter the name of the tea, the number of boxes on hand, and the number of boxes used during a one-day period. The output will display the input data and the number of boxes remaining. Use dialog boxes for program input and output. The program names are **InventoryControl**, **InventorySold**, and **UseInventorySold**. The InventorySold class `extends InventoryControl` and prompts for the quantity of tea boxes sold.

CHAPTER

12

ADVANCED INHERITANCE CONCEPTS

In this chapter, you will:

- Create and use abstract classes
- Use dynamic method binding
- Create arrays of subclass objects
- Use the Object class and its methods
- Use inheritance to achieve good software design
- Create and use interfaces
- Create and use packages

Inheritance sure makes my programming job easier, you tell Lynn Greenbrier over a frosty lemonade at the Event Handlers Incorporated company picnic.

"So everything is going well, I take it?" says Lynn, smiling.

"It is," you say as you smile back, "but I'm ready to learn more. What else can you tell me about inheritance?"

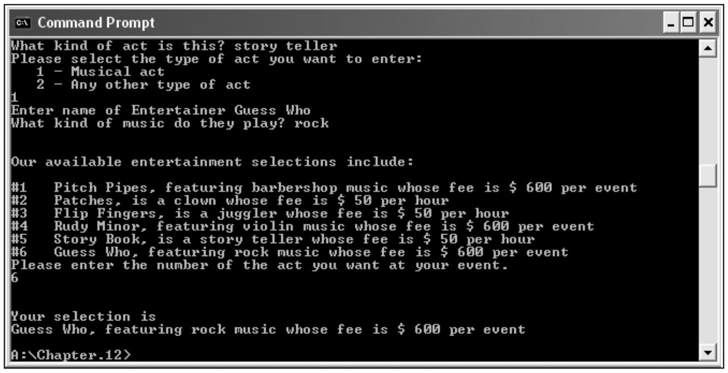
"Enjoy the picnic for today," Lynn says. "On Monday morning I'll teach you about superclass arrays that can use subclass methods, interfaces, and packages. Then you will be an inheritance pro."

PREVIEWING AN EXAMPLE OF USING AN ABSTRACT CLASS

Clients at Event Handlers Incorporated can choose many types of entertainment to feature at their events. Even though Event Handlers uses different class types to store different entertainment types, the different entertainment acts must be stored in a single entertainment database; this can be accomplished using methods introduced in this chapter. You can now use a completed version of the EntertainmentSelector program that is saved in the Chapter.12 folder on your Student Disk.

To use the Chap12EntertainmentSelector class:

1. Go to the command prompt for the Chapter.12 folder on your Student Disk. Determine that the following four classes are in the folder: Chap12EntertainmentSelector, Entertainment, MusicalEntertainment, and OtherEntertainment. At the command prompt type **`java Chap12EntertainmentSelector`**, and then press **[Enter]**. This program allows you to supply data for six entertainment acts that are under contract to Event Handlers Incorporated. When you see the prompt, type **1** or **2** for a musical or non-musical weekend event, respectively.
2. If you indicated that this act is a musical act, the prompts will ask you for an act name and a style of music; if you indicated that this act is a non-musical act, the prompts will ask you for an act type. Supply any answers you want to these questions. After you enter information for six acts, the data you entered will echo to the screen and you will see the charge for each act.
3. Musical acts are paid by the event; non-musical acts are paid by the hour. Select the act you want to perform at your event. The last lines of a typical program run appear in Figure 12-1. (Yours may differ.) You will create a similar program in this chapter.



```
What kind of act is this? story teller
Please select the type of act you want to enter:
   1 - Musical act
   2 - Any other type of act
1
Enter name of Entertainer Guess Who
What kind of music do they play? rock


Our available entertainment selections include:

#1    Pitch Pipes, featuring barbershop music whose fee is $ 600 per event
#2    Patches, is a clown whose fee is $ 50 per hour
#3    Flip Fingers, is a juggler whose fee is $ 50 per hour
#4    Rudy Minor, featuring violin music whose fee is $ 600 per event
#5    Story Book, is a story teller whose fee is $ 50 per hour
#6    Guess Who, featuring rock music whose fee is $ 600 per event
Please enter the number of the act you want at your event.
6


Your selection is
Guess Who, featuring rock music whose fee is $ 600 per event

A:\Chapter.12>
```

Figure 12-1 Output of the Chap12EntertainmentSelector program

Creating and Using Abstract Classes

Creating new classes is easier after you understand the concept of inheritance. When you use a class as a basis from which to create extended child classes, the child classes are more specific than their parent. When you create a child class, it inherits all the general attributes you need; thus you must create only the new, more-specific attributes. For example, a SalariedEmployee and an HourlyEmployee are more specific than an Employee. They can inherit general attributes, such as an employee number, but they add specific attributes, such as pay-calculating methods.

Notice that a superclass contains the features that are shared by its subclasses. The subclasses are more-specific examples of the superclass type; they add additional features to the shared, general features. Conversely, when you examine a subclass, you see that its parent is more general and less specific. Sometimes a parent class is so general that you never intend to create any specific instances of the class. For example, you might never create "just" an Employee; each Employee is more specifically a SalariedEmployee, HourlyEmployee, or ContractEmployee. A class, such as Employee, that you create only to extend from, but not to instantiate from, is an abstract class. An **abstract class** is one from which you cannot create any concrete objects, but from which you can inherit. Abstract classes usually have one or more empty abstract methods. You use the keyword `abstract` when you declare an abstract class.

Tip

Nonabstract classes from which objects can be instantiated are called concrete classes.

Tip

In Chapter 11 you learned that you can create `final` classes if you do not want other classes to be able to extend them. Classes that you declare to be `abstract` are the opposite; your only purpose in creating them is to enable other classes to extend them.

Tip

In other programming languages, such as C++, abstract classes are known as virtual classes.

Abstract classes are like regular classes because they have data and methods but usually contain at least one abstract method. The difference is that you cannot create instances of abstract classes by using the `new` operator. You create abstract classes simply to provide a superclass from which other objects may inherit. Usually, abstract classes contain abstract methods. An **abstract method** is a method with no method statements. When you create an abstract method, you provide the keyword `abstract` and the intended method type, name, and arguments, but you do not provide any statements within the method. When you create a subclass that inherits an abstract method from a parent, you must provide the actions, or implementation, for the inherited method. It's important to understand that you are required to code a subclass method to override the empty superclass method that is inherited. Programmers of abstract classes can include two method types: those that are implemented in the abstract class and simply inherited by its children, and those that are abstract and must be implemented by its children.

Tip

If you attempt to instantiate an object from an abstract class, you will receive an error message that you have committed an InstantiationError.

Tip

If you provide an empty method within an abstract class, the method is an abstract method even if you do not explicitly use the keyword `abstract` when defining the method.

Suppose you want to create classes to represent different animals, such as Dog and Cow.You can create a generic abstract class named Animal so you can provide generic data fields, such as the animal's name, only once.An Animal is generic, but all specific Animals make a sound. The actual sound differs from Animal to Animal. If you code an empty speak() method in the abstract Animal class, then you require all future Animal subclasses to code a speak() method that is specific to the subclass. Figure 12-2 shows an abstract Animal class containing a data field for the name, a constructor, a getName() method, and an abstract speak() method.

```
public abstract class Animal
{
  private String name;

  public Animal(String nm)
  {
    name = nm;
  }

  public String getName()
  {
    return(name);
  }

  public abstract void speak();
}
```

Figure 12-2 Animal class

The Animal class in Figure 12-2 is declared as `abstract`.You cannot place a statement such as `Animal myPet = new Animal("Murphy");` within a program because the program will not execute. Animal is an abstract class, so no Animal objects can exist.

Tip

If you declare any method to be an abstract method, then you must also declare its class to be abstract.

You would create an abstract class such as Animal only so that you can extend it. For example, because a dog is an animal, you can create a Dog class as a child class of Animal. Figure 12-3 shows a Dog class; notice that it extends Animal. The Animal parent class in Figure 12-2 contains a constructor that requires a String holding the Animal's name, so the child Dog class must also contain a constructor that passes a String to its superclass constructor.

Tip

You learned how child class and parent class constructors operate in Chapter 11.

```
public class Dog extends Animal
{
  public Dog(String nm)
  {
    super(nm);
  }
  public void speak()
  {
    System.out.println("Woof");
  }
}
```

Figure 12-3 Dog class

The speak() method within the Dog class is required because the abstract, parent Animal class contains an abstract speak() method. You can code any statements you like within the Dog speak() method, but the speak() method must exist. Remember, you cannot instantiate an Animal object; however, instantiating a Dog object is perfectly legal because Dog is not an abstract class. When you code `Dog myPet = new Dog("Murphy");` you create a Dog object. Then when you code `myPet.speak();`, the correct Dog speak() method executes.

Tip

If you do not provide a subclass method to override a superclass abstract method, then you cannot instantiate any subclass objects. In this case, you also must declare the subclass itself to be abstract. Then you can extend the subclass into sub-subclasses where you write code for the method.

The classes in Figures 12-4 and 12-5 also inherit from the Animal class. When you create a Cow or a Snake object, each Animal will be able to use speak() appropriately.

Tip In Chapter 11, you learned that using the same method name to indicate different implementations is called polymorphism. Using polymorphism, one method name causes different actions for different types of objects.

```
public class Cow extends Animal
{
  public Cow(String nm)
  {
    super(nm);
  }
  public void speak()
  {
    System.out.println("Moo");
  }
}
```

Figure 12-4 Cow class

```
public class Snake extends Animal
{
  public Snake(String nm)
  {
    super(nm);
  }
  public void speak()
  {
    System.out.println("Sss");
  }
}
```

Figure 12-5 Snake class

Next you will create an abstract Entertainment class for Event Handlers Incorporated. The Entertainment class holds data about entertainment acts that customers can hire for their events. The class includes fields for the name of the act and for the fee charged for providing the act. Entertainment is an abstract class. You will create two subclasses, MusicalEntertainment and OtherEntertainment. The more-specific classes include different methods for calculating the entertainment act's fee (musical acts are paid by the performance; other acts are paid by the hour), as well as different methods for displaying data.

To create an abstract Entertainment class:

1. Open a new file in your text editor and enter the following first two lines of an abstract Entertainment class:

   ```
   public abstract class Entertainment
   {
   ```

2. Define the two data fields that hold the entertainer's name and fee as `protected` rather than `private` because you want child classes to be able to access the fields when the fee is set and when the fields are shown on the screen. Define the fields as follows:

```
protected String entertainer;
protected int fee;
```

3. The Entertainment constructor calls two methods. The first method accepts the entertainer's name from the keyboard. The second method sets the entertainer's fee. Because the first method accepts keyboard data entry, you must include the phrase `throws Exception` in the following constructor method header:

```
public Entertainment() throws Exception
{
 setEntertainerName();
 setEntertainmentFee();
}
```

4. Include the following two get methods that return the values for the entertainer's name and the act's fee:

```
public String getEntertainerName()
{
 return entertainer;
}
public double getEntertainmentFee()
{
 return fee;
}
```

5. Enter the following setEntertainerName() method, which is similar to other data-entry methods you have coded in previous chapters. It prompts the user for the name of an entertainment act and assigns the characters to the entertainer field.

```
public void setEntertainerName() throws Exception
{
 String inputString = new String();
 char newChar;
 System.out.print("Enter name of entertainer ");
 newChar = (char)System.in.read();
 while(newChar >= 'A' && newChar<='z'||newChar==' '||
    newChar>='0' && newChar<='9')
 {
    inputString = inputString + newChar;
    newChar = (char)System.in.read();
 }
 System.in.read();
 entertainer = inputString;
}
```

6. The setEntertainmentFee() method is an abstract method. Each subclass you eventually create that represents different entertainment types will have a different fee schedule. Type the abstract method definition and the closing curly brace for the class:

```
public abstract void setEntertainmentFee();
}
```

7. Save the file as **Entertainment.java** in the Chapter.12 folder on your Student Disk. At the command prompt, compile the file using the `javac` command.

You just created an abstract class, but you cannot instantiate any objects from this class. Rather, you must extend this class to be able to create any Entertainment-related objects. Next you will create a MusicalEntertainment class that extends the Entertainment class. This new class will be concrete; that is, you can create actual MusicalEntertainment class objects.

To create the MusicalEntertainment class:

1. Open a new file in your text editor, and then type the following header and opening brace for a MusicalEntertainment class that is a child of the Entertainment class:

```
public class MusicalEntertainment extends Entertainment
{
```

2. Add the definition of a music type field that is specific to musical entertainment by typing: `private String typeOfMusic;`.

3. The MusicalEntertainment constructor must call its parent's constructor. It must also use the following method that sets the music type in which the entertainer specializes:

```
public MusicalEntertainment() throws Exception
{
 super();
 setTypeOfMusic();
}
```

4. Enter the following setTypeOfMusic method, which asks for user input:

```
public void setTypeOfMusic() throws Exception
{
 String inputString = new String();
 char newChar;
 System.out.print("What kind of music do they play? ");
 newChar = (char)System.in.read();
 while(newChar >= 'A' && newChar <= 'z'|| newChar == ' ')
 {
  inputString = inputString + newChar;
  newChar = (char)System.in.read();
 }
 System.in.read();
 typeOfMusic = inputString;
}
```

5. Event Handlers Incorporated charges a flat rate of $600 per event for musical entertainment. Add the following setEntertainmentFee() method to your program:

```
public void setEntertainmentFee()
{
 fee = 600;
}
```

6. Add the following toString() method that you can use when you want to convert the details of a MusicalEntertainment object into a String so you can easily and efficiently display the contents of the object. Add the closing curly brace for the class.

Tip In Chapter 7, you first used the automatically included toString() method that converts objects to Strings. Now you are overriding that method for this class by writing your own version. You will learn more about the toString() method later in this chapter.

```
 public String toString()
 {
  return(entertainer + ", featuring " + typeOfMusic
  + " music whose fee is $ " + fee + " per event!");
 }
}
```

7. Save the file as **MusicalEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file.

Event Handlers Incorporated classifies all non-musical entertainment acts, such as clowns, jugglers, and stand-up comics, as OtherEntertainment. The OtherEntertainment class inherits from Entertainment, just as the MusicalEntertainment class does. Whereas the MusicalEntertainment class requires a data field to hold the type of music played by the act, the OtherEntertainment class requires a field for the type of act. Other differences lie in the content of the prompt within the setTypeOfAct() method, and in the handling of fees. Event Handlers Incorporated charges $50 per hour for non-musical acts, so both the setEntertainmentFee() and toString() methods differ from those in the MusicalEntertainment class.

Next you will create an OtherEntertainment class to implement the abstract method setEntertainmentFee().

To create the OtherEntertainment class file:

1. Open a new file in your text editor, and then type the following first lines of the OtherEntertainment class:

```
public class OtherEntertainment extends Entertainment
{
```

2. Create the following String variable to hold the type of entertainment act (such as comedian): `private String typeOfAct;`.
3. Enter the following code so the OtherEntertainment class constructor calls the parent constructor, then calls its own method to set the act type:

```
public OtherEntertainment() throws Exception
{
 super();
 setTypeOfAct();
}
```

4. Enter the following setTypeOfAct() method:

```
public void setTypeOfAct() throws Exception
{
 String inputString = new String();
 char newChar;
 System.out.print("What type of act is this? ");
 newChar = (char)System.in.read();
 while(newChar >='A' && newChar <= 'z' || newChar == ' ')
{
 inputString = inputString + newChar;
 newChar = (char)System.in.read();
}
System.in.read();
typeOfAct = inputString;
}
```

5. The fee for non-musical acts is $50 per hour, so add the following setEntertainmentFee() method:

```
public void setEntertainmentFee()
{
 fee = 50;
}
```

6. Enter the following toString() method and add the closing curly brace for the class:

```
 public String toString()
 {
  return(entertainer + ", is a " + typeOfAct +
  " whose fee is $ " + fee + " per hour.");
 }
}
```

7. Save the file as **OtherEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file.

Finally, you will create a program that instantiates concrete objects that belong to each of the two child classes.

To create a program that demonstrates using the MusicalEntertainment and OtherEntertainment classes:

1. Open a new file in your text editor, and then enter the DemoEntertainment class header, opening brace, main() method header, and its opening brace as follows:

   ```
   public class DemoEntertainment
   {
    public static void main(String[] args) throws Exception
    {
   ```

2. Enter the following statement that prompts the user to enter a musical act description. Then instantiate a MusicalEntertainment object.

   ```
   System.out.println("Create a musical act description:");
   MusicalEntertainment anAct = new MusicalEntertainment();
   ```

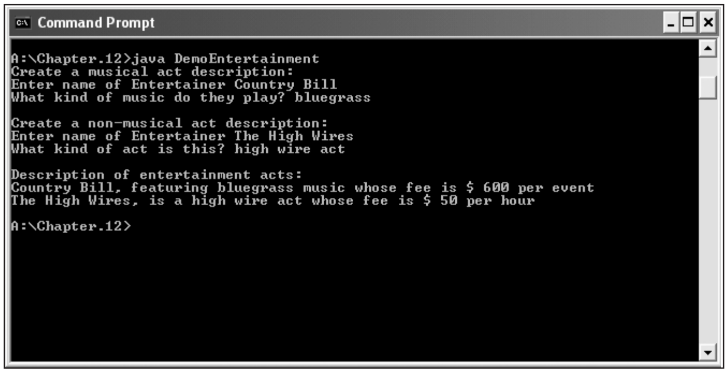
3. Enter the following similar statements for a non-musical act using the newline character `\n` to print on the next line:

   ```
   System.out.println
      ("\nCreate a non-musical act description:");
   OtherEntertainment anotherAct = new OtherEntertainment();
   ```

4. Enter the following lines to display the contents of the two objects and add the closing curly brace for the main() method and for the class:

   ```
    System.out.println
      ("\nDescription of entertainment acts:");
    System.out.println(anAct.toString());
    System.out.println(anotherAct.toString());
    }
   }
   ```

5. Save the file as **DemoEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file. After the file compiles with no errors, run this program using the `java DemoEntertainment` command. When the program prompts you to do so, enter the name of a musical act, a type of music, the name of a non-musical act, and the type of act. Figure 12-6 shows a sample program run.



```
A:\Chapter.12>java DemoEntertainment
Create a musical act description:
Enter name of Entertainer Country Bill
What kind of music do they play? bluegrass

Create a non-musical act description:
Enter name of Entertainer The High Wires
What kind of act is this? high wire act

Description of entertainment acts:
Country Bill, featuring bluegrass music whose fee is $ 600 per event
The High Wires, is a high wire act whose fee is $ 50 per hour

A:\Chapter.12>
```

Figure 12-6 Output of the DemoEntertainment program

Using Dynamic Method Binding

When you create a superclass and one or more subclasses, each object of the subclass "is a" superclass object. Every SalariedEmployee "is an" Employee; every Dog "is an" Animal. (The opposite is not true. Superclass objects are not members of any of their subclasses. An Employee is not a SalariedEmployee. An Animal is not a Dog.) Because every subclass object "is a" superclass member, you can convert subclass objects to superclass objects.

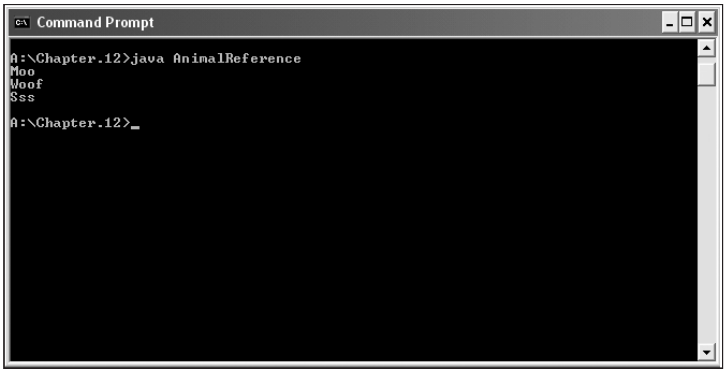
As you are aware, when a superclass is abstract, you cannot instantiate objects of the superclass; however, you can indirectly create a reference to a superclass abstract method. A reference is not an object, but points to a memory address. When you create a reference, you do not use the keyword `new` to create a concrete object; you create a variable name in a subclass in which you can hold the memory address of a subclass concrete object that "is a" superclass member.

Tip: You learned how to create a reference in Chapter 4. When you code `someClass someObject;`, you are creating a reference. If you later code `someObject = new someClass();`, then you actually set aside memory for someObject.

If you create an Animal class, as shown in Figure 12-2, and various subclasses, as shown in Figures 12-3 through 12-5, then you create a new public class with a generic Animal reference variable into which you can assign any of the concrete Animal child objects. Figure 12-7 shows a new AnimalReference class, and Figure 12-8 shows its output. The variable ref is a type of Animal. No superclass Animal object is created, but instead, Dog, Cow, and Snake objects are created using the `new` keyword from classes extended from the Animal class. When the Cow object is assigned to the Animal reference, the ref.speak() method call results in "Moo"; when the Dog object is assigned to the Animal reference, the method call results in "Woof".

```
public class AnimalReference
{
  public static void main(String[] args)
  {

    Animal ref;
    Cow aCow = new Cow("Mabel");
    Dog aDog = new Dog("Rover");
    Snake aSnake = new Snake("Siskal");

    ref = aCow;
    ref.speak();

    ref = aDog;
    ref.speak();

    ref = aSnake;
    ref.speak();

  }
}
```

Figure 12-7 AnimalReference class



```
A:\Chapter.12>java AnimalReference
Moo
Woof
Sss

A:\Chapter.12>_
```

Figure 12-8 Output of the AnimalReference program

Tip: Recall from Chapter 2 that when you assign a variable or constant of one type to a variable of another type, as in `doubleVar = intVar;`, the behavior is called casting.

The program in Figure 12-7 demonstrates polymorphic behavior. The same statement, `ref.speak();`, repeats after ref is set to each new animal type. Each call to the speak() method results in different output. Each reference "chooses" the correct speak() method based on the type of animal referenced. The program's ability to select the correct subclass method is known as **dynamic method binding**. When the program executes, the correct method is attached (or bound) to the program based on the current, changing context (dynamically).

Tip Dynamic method binding is also called late binding.

Creating Arrays of Subclass Objects

You might want to create a superclass reference and treat subclass objects as superclass objects so that you can create an array of different objects that share the same ancestry. For example, even though every Employee object is a SalariedEmployee or an HourlyEmployee subclass object, it can be convenient to create an array of generic Employee objects. Likewise, an Animal array might contain individual elements that are Dog, Cow, or Snake objects. As long as every Employee subclass has access to a calculatePay() method, or every Animal subclass has access to a speak() method, you can manipulate an array of superclass objects by invoking the appropriate method for each subclass member.

Tip In Chapter 8 you learned that all elements in a single array must be of the same type.

The statement `Animal[] ref = new Animal[3];` creates an array of three Animal references. The statement reserves enough computer memory for three Animal objects named ref[0], ref[1], and ref[2]. The statement does not actually instantiate Animals; Animals are abstract and cannot be instantiated. The statement simply reserves memory for three Animal object references. If you instantiate three Animal subclass objects, you can place references to those objects in the Animal array, as Figure 12-9 illustrates.

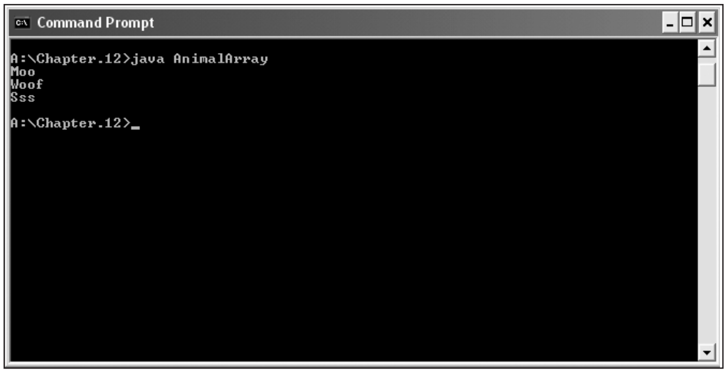
Tip Recall from Chapter 8 that when you create an array of any type of objects, concrete or abstract, you are not actually constructing those objects. Instead, you are creating space for references not yet instantiated.

Once the objects are in the array, you can manipulate them like any other array objects. For example, you can use a loop and a subscript to get each individual reference to speak(). Figure 12-10 shows the output of the AnimalArray program. The output is identical to that of Figure 12-8, except that an array of references is used, instead of a single reference.

```
public class AnimalArray
{

  public static void main(String[] args)
  {

    Animal[] ref = new Animal[3];

    Cow aCow = new Cow("Mabel");

    Dog aDog = new Dog("Rover");

    Snake aSnake = new Snake("Siskal");

    ref[0] = aCow;

    ref[1] = aDog;

    ref[2] = aSnake;

    for (int x = 0;x < 3; ++x)
      ref[x].speak();

  }

}
```

Figure 12-9 AnimalArray class

Next you will write an Event Handlers Incorporated program in which you create an array of Entertainment references. Within the program, you will assign MusicalEntertainment objects and OtherEntertainment objects to the same array. Then, because the different object types are stored in the same array, you can easily manipulate them by using a `for` loop.



```
A:\Chapter.12>java AnimalArray
Moo
Woof
Sss

A:\Chapter.12>_
```

Figure 12-10 Output of the AnimalArray program

To write a program that uses an Entertainment array:

1. Open a new file in your text editor, and then enter the following first few lines of the EntertainmentDataBase program:

```
public class EntertainmentDataBase
{
 public static void main(String[] args) throws Exception
 {
```

2. Create the following array of six Entertainment references and an integer subscript to use with the array:

```
Entertainment[] actArray = new Entertainment[6];
int x;
```

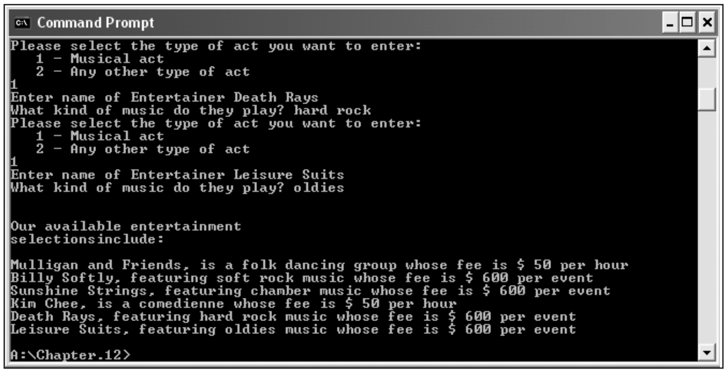
3. Enter the following `for` loop that prompts you to select whether you will enter a musical or non-musical entertainment act. Based on user input, instantiate either a MusicalEntertainment or an OtherEntertainment object.

```
for(x = 0; x < actArray.length; ++x)
{
 char selection;
 System.out.print("Please select the type of ");
 System.out.println("act you want to enter:");
 System.out.println("  1 - Musical act");
 System.out.println("  2 - Any other type of act");
 selection = (char)System.in.read();
 System.in.read(); System.in.read();
 if(selection == '1')
   actArray[x] = new MusicalEntertainment();
 else
   actArray[x] = new OtherEntertainment();
}
```

4. After entering the information for all the acts, display the array contents by typing the following code, and by typing the closing curly braces for the main() method and for the class:

```
 System.out.print("\n\nOur available entertainment ");
 System.out.println("selections include:\n");
 for(x = 0; x < actArray.length; ++x)
 System.out.println(actArray[x].toString());
 }
}
```

5. Save the file as **EntertainmentDataBase.java** in the Chapter.12 folder on your Student Disk, and then compile it. Run the program, enter some appropriate data (use Figure 12-11 as a guide, if you want), and then compare your results to the output shown in Figure 12-11.



```
Please select the type of act you want to enter:
   1 - Musical act
   2 - Any other type of act
1
Enter name of Entertainer Death Rays
What kind of music do they play? hard rock
Please select the type of act you want to enter:
   1 - Musical act
   2 - Any other type of act
1
Enter name of Entertainer Leisure Suits
What kind of music do they play? oldies


Our available entertainment
selectionsinclude:

Mulligan and Friends, is a folk dancing group whose fee is $ 50 per hour
Billy Softly, featuring soft rock music whose fee is $ 600 per event
Sunshine Strings, featuring chamber music whose fee is $ 600 per event
Kim Chee, is a comedienne whose fee is $ 50 per hour
Death Rays, featuring hard rock music whose fee is $ 600 per event
Leisure Suits, featuring oldies music whose fee is $ 600 per event

A:\Chapter.12>
```

Figure 12-11 Final lines of output of the EntertainmentDataBase program

Using the Object Class and Its Methods

Every class in Java is actually a subclass, except one. When you define a class, if you do not explicitly extend another class, then your class is an extension of the Object class. The Object class is defined in the java.lang package, which is imported automatically every time you write a program. It includes methods that you can use or override, as you see fit.

The toString() Method

12

You overrode the Object class toString() method in the steps you used to create the MusicalEntertainment and OtherEntertainment classes. If you do not create a toString() method for a class, then you can use the superclass version of the toString() method. For example, review the Dog class shown in Figure 12-3. Notice that it does not contain a toString() method and that it extends the Animal class. Examine the Animal class shown in Figure 12-2. Notice that it also does not define a toString() method. Yet, when you write the program that prints a Dog object, as shown in Figure 12-12, the program compiles correctly, converts the Dog object to a String, and produces the output shown in Figure 12-13. The output is not very useful, however. It consists of the class name of which the object is an instance, the at sign (@), and a hexadecimal (base 16) number that represents the object.

Tip

The hexadecimal number, which is expressed as a series of digits and letters, represents a computer memory address that can change every time you run the program, and basically is of no use to you.

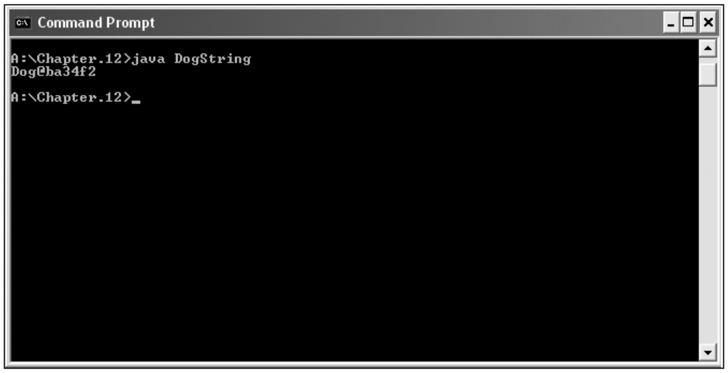
It is usually better to write your own toString() method that displays some or all of the data field values, instead of using the automatic toString() method with your classes.

Tip

A good toString() method can be very useful in debugging a program. If you do not understand why a class is behaving as it is, you can print the toString() value and examine its contents. You can also use a toString() method to return text to a method statement call. In Chapter 10, you used the toString() method to return a String representation of the screen resolution and screen size using a combination of text, method calls, and catenation.

```
public class DogString
{

  public static void main(String[] args)
  {

    Dog myDog = new Dog("Murphy");

    System.out.println(myDog);

  }
}
```

Figure 12-12 DogString program



```
A:\Chapter.12>java DogString
Dog@ba34f2

A:\Chapter.12>_
```

Figure 12-13 Output of the DogString program

The equals() Method

The Object class also contains an equals() method that takes a single argument, which must be the same type as the type of the invoking method, as in the following example:

```
someObject.equals(someOtherObjectOfTheSameType)
```

Tip

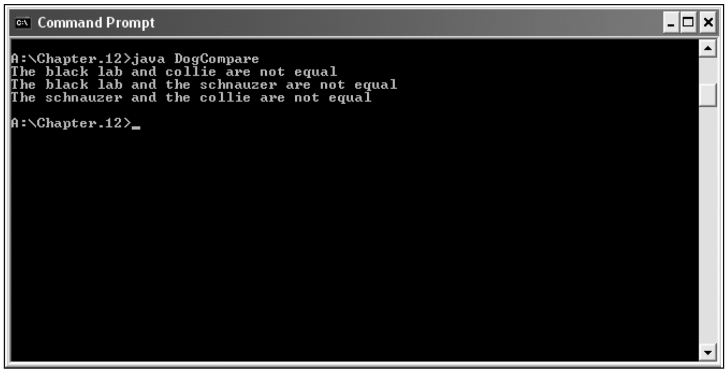
Other classes, such as the String class, also have their own equals() methods. You first used the equals() method to compare String objects in Chapter 7. Two String objects were considered equal only if their String contents were identical.

The Object class equals() method returns a Boolean value indicating whether the objects are equal. This equals() method considers two objects of the same class to be equal only if they have the same memory address; in other words, they are equal only if one is a reference to the other. If you want to consider two objects to be equal only when one is a reference to the other, you can use the Object class equals() method. However, if you want to consider objects to be equal based on their contents, then you must write your own equals() method for your classes.

The program shown in Figure 12-14 instantiates three Dog objects: aBlackLab named Murphy, aCollie named Colleen, and aSchnauzer named Murphy. The Dog class does not include its own equals() method, so it does not override the Object equals() method. Thus, the program in Figure 12-14 produces the output in Figure 12-15. Even though two of the Dog objects have the same name, none of the Dogs are equal because they do not have the same memory address.

```
public class DogCompare
{
  public static void main(String[] args)
  {
    Dog aBlackLab = new Dog("Murphy");
    Dog aCollie = new Dog("Colleen");
    Dog aSchnauzer = new Dog("Murphy");

    System.out.print("The black lab and collie are ");

    if (aBlackLab.equals(aCollie))
      System.out.println("equal");

    else
      System.out.println("not equal");

    System.out.print("The black lab and the schnauzer are ");

    if (aBlackLab.equals(aSchnauzer))
      System.out.println("equal");

    else
      System.out.println("not equal");

     System.out.print("The schnauzer and the collie are ");

       if (aSchnauzer.equals(aCollie))
         System.out.println("equal");

       else
         System.out.println("not equal");

  }

}
```

Figure 12-14 DogCompare program



```
A:\Chapter.12>java DogCompare
The black lab and collie are not equal
The black lab and the schnauzer are not equal
The schnauzer and the collie are not equal

A:\Chapter.12>_
```

Figure 12-15 Output of the DogCompare program when Dog does not override equals()

Next you will add an equals() method to the Dog class to override the equals() method in the DogCompare program.

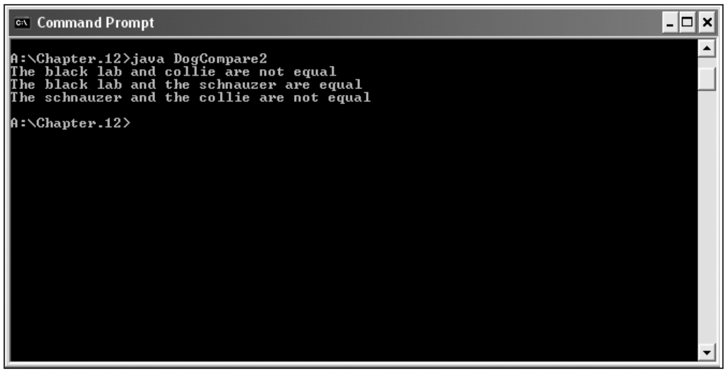
To override the equals() method in the DogCompare program:

1. Open the **Dog.java** program in your text editor, and change the class name to **`Dog2`**. Change the name of the Dog constructor to **`Dog2`**.
2. Add the equals() method shown in Figure 12-16 to the Dog2 class. You can add the equals() method to the Dog2 class file anywhere within the file as long as it is not within any other method. The best location is after the closing curly brace for the Dog2 class constructor and before the method header for speak(). This way, the equals() method appears in alphabetical order among the methods.
3. Save the file as **Dog2.java** in the Chapter.12 folder on your Student Disk, and then compile it.
4. Open the **DogCompare.java** program in your text editor, and change the class name to **`DogCompare2`**.
5. Change each occurrence of Dog that instantiates a new Dog object to **`Dog2`**.
6. Save the file as **DogCompare2.java** in the Chapter.12 folder on your Student Disk, and then compile it. Run the program and observe that the output (as shown in Figure 12-17) now indicates that two Dog objects with the same name are equal.

The equals() method in Figure 12-16 returns a Boolean value. When you call the method, you use the name of one Dog2 object, a dot, and the name of another Dog2 object as an argument within parentheses, as in `aBlackLab.equals(aCollie)`. Therefore, the equals() method header shows that it receives a Dog2 object, which has the local name anotherDog. When you compare the name of the calling Dog2 with the argument anotherDog by using the String equals() method, you determine whether the two Dog2s are to be considered equal.

```
boolean equals(Dog2 anotherDog)
{
  boolean result;
  if(getName().equals(anotherDog.getName()))
    result = true;
  else
    result = false;
  return result;
}
```

Figure 12-16 Dog2 equals() method



```
A:\Chapter.12>java DogCompare2
The black lab and collie are not equal
The black lab and the schnauzer are equal
The schnauzer and the collie are not equal

A:\Chapter.12>
```

Figure 12-17 Output of the DogCompare2 program when Dog2 overrides equals()

Tip: Recall from Chapter 4 that when you use an instance method for a class object, the method receives a `this` reference to the calling object. Therefore, the call to the getName() method retrieves the name for `this` Dog. You could replace the expression `getName().equals(anotherDog.getName())` with `this.getName().equals(anotherDog.getName())`.

Tip: If you change a class, such as adding the equals() method to the Dog class, not only must you recompile the Dog class, but to make use of the newly added method, you must also recompile a client program such as DogCompare.

If there were more fields in the Dog class, you could base equality on a large number of comparisons. Rather than simply comparing names, within the equals() method of the Dog class you could substitute a more-detailed comparison, such as the following:

```
if(getName().equals(anotherDog.getName()) &&
  getAge() == anotherDog.getAge() &&
  getGender() == anotherDog.getGender() &&
  getBreed().equals(anotherDog.getBreed()))
   result = true;
  else
   result = false;
```

Using this code, two Dog objects are considered equal only if they have the same name, age, gender, and breed.

Next you will add an equals() method to the Event Handlers Incorporated Entertainment class. Then you will use the equals() method in the EntertainmentDataBase program to compare each new Entertainment act to every act residing in the database. Your improved program will not allow two acts to have the same name.

To add an equals() method to the Entertainment class:

1. Open the **Entertainment.java** file in your text editor, and change the constructor and the class name to **`Entertainment2`**.
2. Position your insertion point after the closing curly brace of the Entertainment constructor, and then press **[Enter]** to start a new line.
3. Type the equals() method as follows:

```
public boolean equals(Entertainment2 act)
{
 boolean result;
 if(entertainer.equals(act.entertainer))
   result = true;
 else
   result = false;
 return result;
}
```

4. Save the file as **Entertainment2.java**, and then compile it using the **`javac`** command.

Next you will modify the EntertainmentDataBase program so the user cannot enter Entertainment objects with the same entertainer names.

To modify the EntertainmentDataBase class:

1. Open the **EntertainmentDataBase.java** file in your text editor, and then save it as **EntertainmentNoDuplicates.java**. Change the class name in the first line of the class file from EntertainmentDataBase to **`EntertainmentNoDuplicates`**.

2. Change the `Entertainment [] actArray = new Entertainment [6];` statement to **`Entertainment2[] actArray = new Entertainment2[6];`** because the Entertainment class used by the EntertainmentDataBase was updated in a previous example.

3. Change `actArray[x] = new MusicalEntertainment ();` and `actArray[x] = new OtherEntertainment ();` to **`actArray[x] = new MusicalEntertainment2();`** and **`actArray[x] = new OtherEntertainment2();`**, respectively, and then save the file.

4. Open the **MusicalEntertainment** file, and then save it as **MusicalEntertainment2.java**. Change the name of the class and the name of the constructor so each becomes **`MusicalEntertainment2`**. Change the class that this class extends to **`Entertainment2`**. Save the file, and then compile it.

5. Open the **OtherEntertainment** file, and then save it as **OtherEntertainment2.java**. Change the name of the class and the name of the constructor so each become **`OtherEntertainment2`**. Change the class that this class extends to **`Entertainment2`**. Save the file, and then compile it.

Tip

You must change the names of the MusicalEntertainment and OtherEntertainment files so that the new files MusicalEntertainment2 and OtherEntertainment2 now extend Entertainment2. When the equals() method in the Entertainment2 class receives an object to compare, it must be of the same type as Entertainment2.

6. Open the **EntertainmentNoDuplicates** file if necessary, and position your insertion point at the end of the line that reads **`actArray[x] = new OtherEntertainment2();`**. Press **[Enter]** to start a new line, and then add the following `for` loop that compares the most recently entered actArray element against all previously entered actArray elements. If the new element equals any previously entered Entertainment act, then issue an error message and reduce the subscript by one. Reducing the subscript ensures that the next act you enter overwrites the duplicate act.

```
for(int y = 0; y < x; ++y)
 if(actArray[x].equals(actArray[y]))
 {
  System.out.println
   ("Sorry, you entered a duplicate act");
  —x;
}
```

7. Save the file, compile it using the **`javac`** command, and then execute the program. When you see the prompts, enter any appropriate data. Make sure that you repeat an entertainer's name for several of the prompts. Each time you repeat a name, you will see an error message and get another opportunity to enter an act. The program will not end until you enter six acts with unique names.

Using Inheritance to Achieve Good Software Design

When an automobile company designs a new car model, the company does not build every component of the new car from scratch. The company might design a new feature completely from scratch; for example, at some point someone designed the first air bag. However, many of a new car's features are simply modifications of existing features. The manufacturer might create a larger gas tank or more comfortable seats, but even these new features still possess many properties of their predecessors in the older models. Most features of new car models are not even modified; instead, preexisting components, such as air filters and windshield wipers, are included on the new model without any changes.

Similarly, you can create powerful computer programs more easily if many of their components are used either "as-is" or with slight modifications. Inheritance does not give you the ability to write any programs that you could not write without it because you could create every part of a program from scratch. Inheritance simply makes your job easier. Professional programmers constantly create new class libraries for use with Java programs. Having these classes available makes programming large systems more manageable.

You have already used many "as-is" classes, such as String and JApplet. In these cases, your programs were easier to write than if you had to write these classes yourself. Now that you have learned about inheritance, you have gained the ability to modify existing classes. When you create a useful, extendable superclass, you and other future programmers gain several advantages:

- Subclass creators save development time because much of the code needed for the class has already been written.
- Subclass creators save testing time because the superclass code has already been tested and probably used in a variety of situations. In other words, the superclass code is reliable.
- Programmers who create or use new subclasses already understand how the superclass works, so the time it takes to learn the new class features is reduced.
- When you create a new subclass in Java, neither the superclass source code nor the superclass bytecode is changed. The superclass maintains its integrity.

When you consider classes, you must think about the commonalities between them, and then you can create superclasses from which to inherit. You might be rewarded professionally when you see your own superclasses extended by others in the future.

Creating and Using Interfaces

Many object-oriented programming languages, such as C++, allow a subclass to inherit from more than one parent class. For example, you might create an Employee class that contains data fields pertaining to each employee in your organization. You might also create a Product class that holds information about each product your organization produces.

When you create a Patent class for each product for which your company holds a patent, you might want to include product information, as well as information about your company's employee who was responsible for the invention. It would be convenient to inherit fields and methods from both the Product and the Employee classes. The capability to inherit from more than one class is called **multiple inheritance**.

Multiple inheritance is a difficult concept, and when programmers use it, they encounter many problems. Programmers have to deal with the possibility that variables and methods in the parent classes may have identical names, which creates conflict when the child class uses one of the names. Also, you have already learned that a child class constructor must call its parent class constructor. When there are two or more parents, this task becomes more complicated. To which class should super() refer when a child class has multiple parents?

For all of these reasons, multiple inheritance is prohibited in the Java programming language. Java, however, does provide an alternative to multiple inheritance—an interface. An **interface** looks much like a class, except all of its methods must be abstract and all of its data (if any) must be `static final`. When you create a class that uses an interface, you include the keyword `implements` and the interface name in the class header. This notation requires class objects to include code for all the methods in the interfaces that have been implemented. Whereas `extends` exposes elements of the superclass program to the user of subclasses, `implements` exposes elements of your programs to the user without exposing the program source code.

As an example, you can create a Working interface to use with the Animal subclasses. For simplicity, give the Working interface a single method named work(). Figure 12-18 shows the Working program.

```
public interface Working
{
  public void work();

}
```

Figure 12-18 Working interface

When any class implements Working, it must also include a work() method. The WorkingDog class in Figure 12-19 extends Dog and implements Working; its work() method calls the Dog speak() method, and then produces a line of output.

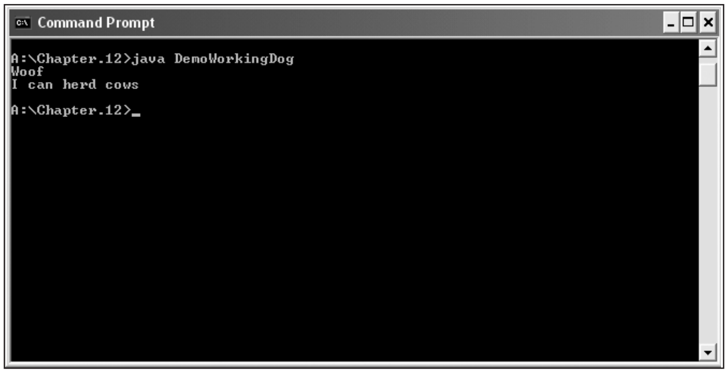
When you create a program that instantiates a WorkingDog object, as in Figure 12-20, you can use the work() method. The program output appears in Figure 12-21. You can also create WorkingCow and WorkingHorse classes that implement Working. In addition, if you decide to create a Playing interface, any class that implements Working can also implement Playing.

```
public class WorkingDog extends Dog implements Working
{
  public WorkingDog(String nm)
  {
    super(nm);
  }

  public void work()
  {
    speak();
    System.out.println("I can herd cows");
  }
}
```

Figure 12-19 WorkingDog class

```
public class DemoWorkingDog
{
  public static void main(String[] args)
  {
    WorkingDog mySheltie = new WorkingDog("Simon");

    mySheltie.work();
  }
}
```

Figure 12-20 DemoWorkingDog program



```
A:\Chapter.12>java DemoWorkingDog
Woof
I can herd cows

A:\Chapter.12>_
```

Figure 12-21 Output of the DemoWorkingDog program

Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one. Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one abstract superclass, but it can implement any number of interfaces.

Beginning programmers sometimes find it difficult to decide when to create an abstract superclass and when to create an interface. Remember, you create an abstract class when you want to provide data or methods that subclasses can inherit, but at the same time these subclasses maintain the ability to override the inherited methods.

Suppose you create a CardGame class to use as a base class for different card games. It contains four methods named shuffle(), deal(), listRules(), and keepScore(). The shuffle() method works the same way for every CardGame, so you write the statements for shuffle() within the superclass, and any CardGame objects you create later inherit shuffle(). The methods deal(), displayRules(), and keepScore() operate differently for every subclass, so you force CardGame children to contain instructions for those methods by leaving them empty in the superclass. When you write classes named Hearts, Solitaire, and Poker, you extend the CardGame parent class, inherit the shuffle() method, and implement deal(), displayRules(), and keepScore() methods for each specific child.

You create an interface when you know what actions you want to include, but you also want every user to separately define the behavior that must occur when the method executes. Suppose you create a MusicalInstrument class to use as a base for different musical instrument object classes such as Piano, Violin, and Drum. The parent MusicalInstrument class contains methods such as playNote() and outputSound() that apply to every instrument, but you want to implement these methods differently for each type of instrument. By making MusicalInstrument an interface, you require every subclass to code all the methods.

Tip

An interface specifies only the messages to which an object can respond; an abstract class can include methods that contain the actual behavior the object performs when those messages are received.

You also create an interface when you want a class to implement behavior from more than one parent. A NameThatInstrument card game that requires players to identify instrument sounds they hear by clicking cards, for example, could not extend from two classes, but it could extend from CardGame and implement MusicalInstrument.

Tip

You used prewritten interfaces earlier in this book. For example, you implemented the ActionListener interface in applets you wrote in Chapters 9 and 10. By implementing ActionListener, you provided your applet the means to respond to ActionEvents. You will use more interfaces in future chapters.

Creating and Using Packages

Throughout most of this book, you have imported packages into your programs. You learned in Chapter 4 that the java.lang package is automatically imported into every program you write. You have explicitly imported packages such as java.util and javax.Swing. When you create your own classes, you can place them in packages so that you or other programmers can easily import related classes into new programs. When you create a number of classes that inherit from each other, you will often find it convenient to place these classes in a package.

Tip

Creating packages encourages others to reuse software because it makes it convenient to import many related classes at once.

When you create classes for others to use, you most often do not want to provide the users with your source code in the files with .java extensions. You expend significant effort developing workable code for your programs, and you do not want other programmers to be able to copy your programs, make minor changes, and market the new product themselves. Rather, you want to provide users with the compiled files with the .class extensions. These are the files the user needs to run the program you have developed. Likewise, when other programmers use the classes you have developed, they need only the completed compiled code to import into their programs. The .class files are the files you place in a package so other programmers can import them.

You can include a package statement at the beginning of your class file to place the compiled code into the indicated folder. For example, when it appears at the beginning of a class file, the statement `package com.course.animals;` indicates that the compiled file should be placed in a folder named com.course.animals. That is, the compiled file will be stored in the animals subfolder inside the course subfolder inside the com subfolder (or com\course\animals). The pathname can contain as many levels as you want. The package statement must appear outside the class definition.

Tip

The package statement, import statements, and comments are the only statements that appear outside class definitions in Java program files.

When you compile a file that you want to place in a package, you must use a compiler option with the `javac` command. The `-d` option indicates that you want to place the generated .class file in a folder. For example, the command `javac -d c:\Animal.java` indicates that the compiled Animal.java file should be placed in the root directory of drive C. If the Animal class file contains the statement `package com.course.animals;`, then the Animal.class file will be placed in C:\com\course\animals. If any of these subfolders do not exist, Java will create them. If you similarly package the compiled files for Dog.java, Cow.java, and so on, future programs only need to use the statement `import com.course.animals.*` to be able to use all the related classes. Alternately, you can list each class separately, as in `import com.course.Dog;` and `import com.course.Cow;`. Usually, if you want to use only one or two classes in a package, you use separate import statements for each class. If you want to use many classes in a package, it is easier to import the entire package, even if there are some classes you will not use.

Tip

The *d* in the `-d` compiler option stands for directory, which is another name for folder.

Tip

You cannot import more than one package in one statement; for example, `import com.*` does not work.

Because the Java programming language is used extensively on the Internet, it is important to give every package a unique name. Sun Microsystems, the creator of the Java programming language, has defined a package-naming convention in which you use your Internet domain name in reverse order. For example, if your domain name is course.com, then you begin all of your package names with com.course. Subsequently, you organize your packages into reasonable subfolders. Using this convention ensures that your package names will not conflict with those of any other Java code providers.

Next you will place some of the Event Handlers Incorporated classes into a package. Because Event Handlers Incorporated sponsors a Web site at *eventhandlers.com*, you will use the com.eventhandlers package.

To place three of your classes for Event Handlers Incorporated into a package:

1. Open the **Entertainment.java** file in your text editor.
2. For the first line in the file, insert the following statement:

   ```
   package com.eventhandlers.entertainment;
   ```

3. Save the file as **a:\Entertainment.java**. Notice that you are saving this file in the root directory of your disk, and not in the Chapter.12 folder. Because Java uses the dot (period) to separate folder names for packages, you cannot use a dot within a folder name.

Tip

Note that your drive letter may vary.

4. At the command line for the root directory on your Student disk, compile the file using the command **`javac -d a:\ Entertainment.java`**. (Make sure that you type a space between the drive name and the class name Entertainment.java.) Java will create a folder named com\eventhandlers\entertainment on your Student Disk. The compiled Entertainment.class file will be placed within this folder.

Help

If you see a list of compile options when you try to compile the file, then you did not type a space between a:\ and Entertainment.java. Repeat Step 4 to compile again.

Tip

To change to the command prompt for the root directory on your Student Disk, type `cd\` at the command prompt.

5. Examine the folders on your Student Disk, using any operating system program with which you are familiar. For example, if you are compiling at the DOS command line, type **`dir a:\`** at the command-line prompt to view the folders stored in the root directory. You can see that Java created a folder named com. Within the com folder is an eventhandlers folder, and within eventhandlers is an entertainment folder. The Entertainment.class file is within the entertainment subfolder, and not in the same folder as the .java source file where it ordinarily would be placed.

? Help

If Java did not create a com folder on your Student Disk, then you probably did not compile the file at the command prompt for the root directory. Repeat Steps 4 and 5, but make sure that you first change to the command prompt for the root directory.

6. Delete the copy of the **Entertainment.java** file from the root directory of your Student Disk. There is no further need for this source file because the compiled .class file is stored in the com\eventhandlers\entertainment folder.

Tip

If you don't want to delete the Entertainment.java file, you can move it to the Chapter.12 folder and overwrite the existing file.

7. Open the **MusicalEntertainment.java** file in your text editor. For the first line in the file, insert the following statement: **`package com.eventhandlers.entertainment;`**.
8. Save the file in the root directory as **a:\MusicalEntertainment.java**. At the command line for the root directory on your Student Disk, compile the file using the command **`javac -d a:\ MusicalEntertainment.java`**. (Make sure that you type a space between the drive name and the MusicalEntertainment.java filename.) Then delete the **MusicalEntertainment.java** source file from the root directory of your Student Disk.
9. Open the **OtherEntertainment.java** file in your text editor. For the first line in the file, insert the following statement: **`package com.eventhandlers.entertainment;`**. Save the file in the root directory of your Student Disk as **a:\OtherEntertainment.java**. At the command line for the root directory of your Student Disk, compile the file using the command: **`javac -d a:\ OtherEntertainment.java`**. Then delete the **OtherEntertainment.java** source file from the root directory.

10. Open the **EntertainmentDataBase.java** file in your text editor. For the first line in the file, insert the following statement: **`import com.eventhandlers.entertainment.*;`**. Save the file as **a:\EntertainmentDataBase.java**. Compile the file at the A:\> prompt using the **`javac EntertainmentDataBase.java`** command, and then run the program at the A:\> prompt using the **`java EntertainmentDataBase`** command. The program's output should be the same as it was before you added the import statement.

Tip

Instead of using the wildcard `import com.eventhandlers.entertainment.*;`, you can use three separate import statements that name the classes individually, such as `import com.eventhandlers.entertainment.Entertainment;`.

11. Examine again the contents of your Student Disk. The only .class file in the root directory of your Student Disk is the EntertainmentDataBase.class file. Because this file imports the class files from the com.eventhandlers.entertainment package, your program recognizes the Entertainment, MusicalEntertainment, and OtherEntertainment classes, even though neither their .java files nor their .class files are in the same folder with the EntertainmentDataBase.

Placing the Entertainment-related class files in a folder is not required for the EntertainmentDataBase program to execute correctly; you ran it in exactly the same manner before you learned about creating packages. The first time you executed the EntertainmentDataBase, all the files you used (source files as well as .class compiled files) were in the Chapter.12 folder on your Student Disk. If you distribute that folder to clients, they have access to all the code you have written.

After placing the class files in a package, you could import the package into the EntertainmentDataBase program, and run the EntertainmentDataBase program from a separate folder. The folder with the three .class files is the only folder you would want to distribute to programmers who use your Entertainment classes to write programs similar to EntertainmentDataBase. Placing classes in packages gives you the ability to more easily isolate and distribute files.

Chapter Summary

- A class that you create only to extend from, but not to instantiate from, is an abstract class. An abstract class is one from which you cannot create any concrete objects, but from which you can inherit. You use the keyword `abstract` when you declare an abstract class.

- You cannot create instances of abstract classes using the `new` operator. Usually, abstract classes contain abstract methods. An abstract method is a method with no method statements. When you create an abstract method, you provide the keyword `abstract` and the intended method type, name, and arguments, but you do not provide any statements within the method. You must code a subclass method to override any inherited abstract superclass method.
- When you create a superclass and one or more subclasses, each object of the subclass "is a" superclass object. Because every subclass object "is a" superclass member, you can convert subclass objects to superclass objects.
- You can create a reference to a superclass. When you create a reference, you do not use the keyword `new` to create a concrete object. You create a variable name in which you can hold the memory address of a subclass concrete object that "is a" superclass member.
- The ability of a program to select the correct subclass method is known as dynamic method binding.
- You might want to create a superclass reference and treat subclass objects as superclass objects so you can create an array of different objects that share the same ancestry. You can manipulate an array of superclass objects by invoking the appropriate method for each subclass.
- When you create a useful, extendable superclass, you save development time because much of the code needed for the class has already been written. In addition, you save testing time and, because the superclass code is reliable, you reduce the time it takes to learn the new class features. You also maintain superclass integrity.
- An interface is similar to a class, but all of its methods must be abstract, and all of its data (if any) must be `static final`. When you create a class that uses an interface, you include the keyword `implements` and the interface name in the class header. This notation serves to require class objects to include code for all the methods in the interface.
- Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either. Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one abstract superclass, but it can implement any number of interfaces.
- You can place classes in packages so you or other programmers can easily import related classes into new programs. When you create a number of classes that inherit from each other, you often will find it convenient to place them in a package.
- You can include a package statement at the beginning of your class file to place compiled code in the indicated folder. The package statement must appear outside the class definition. When you compile a file that you want to place in a package, you must use the `-d` compiler option with the `javac` command.

- The convention for naming packages uses Internet domain names in reverse order to ensure that your package names will not conflict with those of any other Internet users.

Review Questions

1. Parent classes are ________________ than their child classes.
   a. smaller
   b. more specific
   c. easier to understand
   d. more cryptic
2. Abstract classes differ from regular classes in that you ________________.
   a. must not code any methods within them
   b. must instantiate objects from them
   c. cannot instantiate objects from them
   d. cannot have data fields within them
3. Abstract classes can contain ________________.
   a. abstract methods
   b. nonabstract methods
   c. both of the above
   d. none of the above
4. An abstract class Product has two subclasses, Perishable and NonPerishable. None of the constructors for these classes requires any arguments. Which of the following statements is legal?
   a. `Product myProduct = new Product();`
   b. `Perishable myProduct = new Product();`
   c. `NonPerishable myProduct = new NonPerishable();`
   d. none of the above
5. An abstract class Employee has two subclasses, Permanent and Temporary. The Employee class contains an abstract method named setType(). Before you can instantiate Permanent or Temporary objects, which of the following statements must be true?
   a. You must code statements for the setType() method within the Permanent class.
   b. You must code statements for the setType()method within both the Permanent and the Temporary classes.

c. You must not code statements for the setType() method within either the Permanent or Temporary classes.

d. You may code statements for the setType() method within the Permanent class or the Temporary class, but not both.

6. When you create a superclass and one or more subclasses, each object of the subclass ________________ superclass object.

   a. overrides the

   b. "is a"

   c. "is not a"

   d. is a new

7. Which of the following statements are false?

   a. Subclass objects are members of their superclass.

   b. Superclass objects can contain abstract methods.

   c. You can convert subclass objects to superclass objects.

   d. Two of the above statements are false.

8. When you create a ________________, you create a variable name in which you can hold the memory address of an object.

   a. class

   b. superclass

   c. subclass

   d. reference

9. The program's ability to select the correct subclass method to execute is known as ________________ method binding.

   a. polymorphic

   b. dynamic

   c. early

   d. intelligent

10. The statement ________________ creates an array of five reference objects of an abstract class named Currency.

   a. `Currency[] = new Currency[5];`

   b. `Currency[] currencyref= new Currency[5];`

   c. `Currency[5] currencyref = new Currency[5];`

   d. `Currency[5] = new Currency[5];`

11. You ________________ override the toString() method in any class you create.
    a. cannot
    b. can
    c. must
    d. must implement StringListener to
12. The Object class equals() method takes ________________.
    a. no arguments
    b. one argument
    c. two arguments
    d. as many arguments as you need
13. The following statement appears in a Java program:
    `if(thing.equals(anotherThing)) x = 1;`. You know that ________________.
    a. thing is an object of the Object class
    b. anotherThing is the same type as thing
    c. both of the above are correct
    d. none of the above are correct
14. The Object class equals() method considers two objects of the same class to be equal if they have the same ________________.
    a. value in all data fields
    b. value in any data field
    c. data type
    d. memory address
15. Java subclasses have the ability to inherit from ________________ parent class(es).
    a. one
    b. two
    c. multiple
    d. no
16. The alternative to multiple inheritance in Java is known as a(n) ________________.
    a. superobject
    b. abstract class
    c. interface
    d. none of the above

17. When you create a class that uses an interface, you include the keyword ________________ and the interface's name in the class header.
    a. `interface`
    b. `implements`
    c. `accouterments`
    d. `listener`
18. You cannot instantiate concrete objects from a(n) ________________.
    a. abstract class
    b. interface
    c. either a or b
    d. neither a nor b
19. In Java, a class can ________________.
    a. inherit from only one abstract superclass
    b. implement only one interface
    c. both a and b
    d. neither a nor b
20. When you want to provide some data or class that subclasses can inherit, but you want the subclasses to override some specific methods, you should write a(n) ________________.
    a. abstract class
    b. interface
    c. final superclass
    d. concrete object

EXERCISES

1. a. Create an abstract class named Book. Include a String field for the book's title and a double field for the book's price. Within the class, include a constructor that requires the book title and two get methods—one that returns the title and one that returns the price. Also include an abstract method named setPrice(). Create two child classes of Book: Fiction and NonFiction. Within the constructors for the Fiction and NonFiction classes, call setPrice so all Fiction Books cost $24.99 and all NonFiction Books cost $37.99. Finally, write a program that demonstrates that you can create both a Fiction and a NonFiction Book and display their fields. Save the **Book.java**, **Fiction.java**, **NonFiction.java**, and **UseBook.java** programs in the Chapter.12 folder on your Student Disk.

   b. Write a program named BookArray in which you create an array that holds 10 Books, some Fiction and some NonFiction. Using a `for` loop, display details about all 10 books. Save the **BookArray.java** program in the Chapter.12 folder on your Student Disk.

2. a. Create an abstract class named Account for a bank. Include an integer field for the account number and a double field for the account balance. Also include a constructor that requires an account number and sets the balance to 0.0. Include a set method for the balance. Also include two abstract get methods—one for each field. Create two child classes of Account: Checking and Savings. Within the Checking class, the get method displays the String "Checking Account Information", the account number, and the balance. Within the Savings class, add a field to hold the interest rate, and require the Savings constructor to accept an argument for the value of the interest rate. The Savings get method displays the String "Savings Account Information", the account number, the balance, and the interest rate. Save the **Account.java**, **Checking.java**, and **Savings.java** programs in the Chapter.12 folder on your Student Disk.

   b. Write a program named AccountArray in which you enter data for a mix of 10 Checking and Savings accounts. Use a `for` loop to display the data. Save the **AccountArray.java** program in the Chapter.12 folder on your Student Disk.

3. Create an abstract Auto class with fields for the car make and price. Include get and set methods for these fields; the setPrice() method is abstract. Create two subclasses for individual automobile makers (for example, Ford or Chevy) and include appropriate setPrice() methods in each subclass. Finally, write a program that uses the Auto class and subclasses to display information about different cars. Save the **Auto.java**, **Ford.java**, **Chevy.java**, and **UseAuto.java** programs in the Chapter.12 folder on your Student Disk.

4. Create an abstract class Division with fields for a company's division name and account number, and corresponding get and set methods. Use a constructor in the superclass. Create at least two subclasses for divisions such as Accounting or Human Resources. Write a program that uses the classes and displays information about them. Save the **Division.java**, **HumanResources.java**, **Accounting.java**, and **UseDivision.java** programs in the Chapter.12 folder on your Student Disk.

5. Write a program named UseChildren that uses an abstract Child class, and Male and Female subclasses, to display the name, gender, and age of two or more children. Use constructors with appropriate arguments in each of the classes. Include get and set methods, at least one of which is abstract. Save the **Child.java**, **Male.java**, **Female.java**, and **UseChildren.java** programs in the Chapter.12 folder on your Student Disk.

6. Create a class named NewsPaperSubscriber with fields for a subscriber's street address and the subscription rate. Include get and set methods for the subscriber's street address, and get and set methods for the subscription rate. The set method for the rate is abstract. Include an equals() method that indicates two Subscribers are equal if they have the same street address. Create child classes named SevenDaySubscriber, WeekdaySubscriber, and WeekendSubscriber. Each child class constructor sets the rate

as follows: SevenDaySubscribers pay $4.50 per week, WeekdaySubscribers pay $3.50 per week, and WeekendSubscribers pay $2.00 per week. Each child class should include a toString() method that returns the street address, rate, and service type. Write a program named Subscribers that prompts the user for the subscriber's street address and requested service, and creates the appropriate object based on the service type. Do not let the user enter more than one subscription type for any given street address. Save the **NewspaperSubscriber.java**, **WeekdaySubscriber.java**, **WeekEndSubscriber.java**, and **Subscriber.java** programs in the Chapter.12 folder on your Student Disk.

7. a. Create an interface named Turning, with a single method named turn(). Create a class named Leaf that implements turn() to print "Changing colors". Create a class named Page that implements turn() to print "Going to the next page". Create a class named Pancake that implements turn() to print "Flipping". Write a program named Turners that creates one object of each of these class types and demonstrates the turn() method for each class. Save the **Turning.java**, **Leaf.java**, **Pager.java**, **Pancake.java**, and **Turners.java** programs in the Chapter.12 folder on your Student Disk.

   b. Think of two more objects that use turn(), create classes for them, and then add objects to the Turners program. Save the programs, using the names of new objects that use turn(), in the Chapter.12 folder on your Student Disk.

8. Write a program that uses an abstract class named Drug, and subclasses for two specific drugs to display a drug, its purpose, and the number of times per day it should be taken. Use constructors in each class, with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the drug to be displayed, and then create the appropriate object. Save the programs, using the name of each drug for the program name, in the Chapter.12 folder on your Student Disk.

9. Write a program named UseInsurance that uses an abstract Insurance class and Health and Life subclasses to display different types of insurance policies and the cost per month. Use constructors in each class, with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the type to be displayed, and then create the appropriate object. Also create an interface for a print() method and use this interface with both subclasses. Save the **Life.java**, **Health.java**, **Insurance.java**, **Print.java**, and **UseInsurance.java** programs in the Chapter.12 folder on your Student Disk.

10. Write a program named UseLoan that uses an abstract class named Loans and subclasses to display different types of loans and the cost per month (home, car, and so on). Use constructors in each of the classes with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the type to be displayed, and then create the appropriate object. Also create an interface with at least one method that you use with your subclasses. Save the **Loans.java**, **Car.java**, **Home.java**, **Print.java**, and **UseLoan.java** programs in the Chapter.12 folder on your Student Disk.

11. Create an abstract class called GeometricFigure. Each figure includes a height, a width, a figure type, and an area. Include an abstract method to determine the area of the figure. Create two subclasses called Square and Triangle. Create a program that demonstrates the use of both subclasses, and create them using an array. Save the **GeometricFigure.java**, **Square.java**, **Triangle.java**, and **UseGeometric.java** programs in the Chapter.12 folder on your Student Disk.

CASE PROJECT

Sanchez Construction Loan Co. makes small loans for construction projects up to a maximum of $10,000.00. The current cost for a $10,000 loan is based on the following fee structure that has a maximum loan payoff time of 24 months:

Time	Fee
6 months	$800.00
12 months	$1,800.00
18 months	$3,000.00
24 months (max)	$4,000.00

You have been asked to write a program that will track a starting and ending date (due date) for all new construction loans. The program must also calculate the amount of the original loan and the total amount owed at the due date (original loan amount + loan fee). The program should include four classes, as shown in the table below:

Class	Type
Loan	public class
LoanInterface	public interface
AnnualLoan	`public class extends Loan implements LoanInterface`
DemoLoan	Test program

The Loan Interface requires a CalculateFee() method. This method must be implemented in the AnnualLoan class. The DemoLoan test program should instantiate at least two AnnualLoan objects that output the loan amount of $10,000, the loan fee for the period, the beginning date of the loan, the ending date of the loan, and the balance due at the loan due date. Save the programs as **Loan.java**, **LoanInterface.java**, **AnnualLoan.java**, and **DemoLoan.java** in the Chapter.12 folder on your Student Disk.

CHAPTER

13

UNDERSTANDING SWING COMPONENTS

In this chapter, you will:

- Use the JFrame class
- Use additional JFrame class methods
- Use Swing event listeners
- Use JPanel class methods
- Use the JCheckBox class
- Use the ButtonGroup classes
- Create a drop-down list and combo box using the JComboBox class
- Create JScrollPanes
- Create JToolBars

Learning about inheritance has been interesting," you say to Lynn Greenbrier, "and I certainly can see how using inheritance is going to make my programming life easier. But will understanding inheritance help me create fancier applets, such as ones with frames, user lists, and choice boxes?"

"You bet it will," Lynn replies. "One reason I gave you such a thorough grounding in inheritance concepts is so it will be easier for you to learn to use JFrame-type components. All the little gadgets such as the JComboBoxes that you want to put in your JFrames are relatives, and inheritance makes it possible to use all of them. What is more important, if you have a thorough knowledge of how inheritance and components work in general, then you can adapt your knowledge to other components."

"You won't show me every component?" you ask worriedly.

"I don't have time to show you every component now," Lynn says. "Besides, there are new components that Java developers around the world are shaping right this minute."

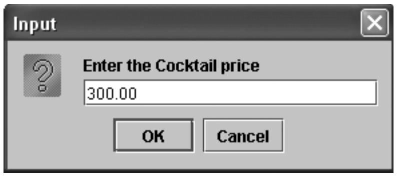
"In other words, I can use the knowledge you give me about components, and then I can extend that knowledge to future components. That's just like inheritance," you tell Lynn. "Please explain more."

Previewing the Swing Application for Chapter 13

Event Handlers Incorporated is developing a Swing application that lets a user determine the price of an event based on several event choices. For some options, such as whether cocktails or dinner will be served, a user can select options in any combination (serve only cocktails, serve only dinner, serve both cocktails and dinner, or serve nothing). For other options, such as the dinner entrée or the entertainment, only one choice is allowed. The Chap13JDemoButtonGroup class incorporates several such devices, which you can use now.

To use the Chap13JDemoButtonGroup class:

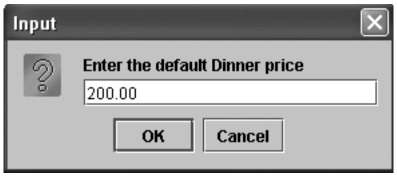
1. Go to the command prompt for the Chapter.13 folder on your Student Disk, type `java Chap13JDemoButtonGroup`, and then press **[Enter]**. In the input dialog box, type **300.00** for the cost of cocktails (see Figure 13-1), and then press **[Enter]**.


Enter the Cocktail price

300.00

Figure 13-1 Input dialog box showing the Cocktail price

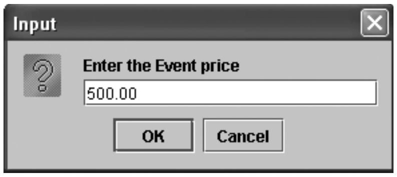
2. To enter the cost of the default dinner price, type **200.00** in the input dialog (see Figure 13-2), and then press **[Enter]**.


Enter the default Dinner price

200.00

Figure 13-2 Input dialog box showing the default Dinner price

3. To enter the event price, type **500.00** in the input dialog box (see Figure13-3), and then press **[Enter]**.


Enter the Event price

500.00

Figure 13-3 Input dialog box showing the Event price

4. The initial event cost of $500 is shown in Figure 13-4 before any boxes have been selected. Click the **Cocktails** box and **Beef** box and observe how the total price of the event changes in response to your selections. The total cost of the event, $1100 ($500 for the initial event cost, $300 for cocktails, and $300 for beef), is shown in Figure 13-5 after the cocktails and the beef are selected.
5. Close the application.

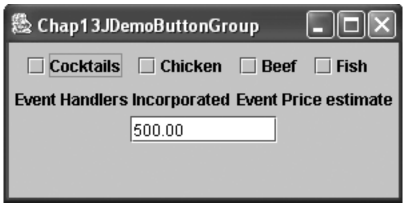


Figure 13-4 Output of Chap13JDemoButtonGroup before any boxes have been selected

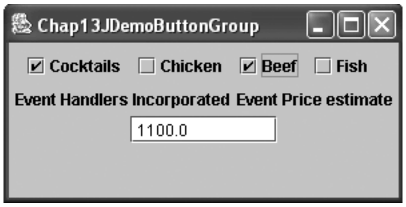


Figure 13-5 Output of Chap13JDemoButtonGroup with the Cocktails and the Beef boxes selected

USING THE JFRAME CLASS

Computer programs are usually more user friendly (and more fun to use) when they contain graphical user interface (GUI) components such as buttons, check boxes, and menus. In Chapter 9, you learned how to add a few GUI components to a Swing applet; in this chapter, you will learn how to add several more.

Tip: You can add GUI components to either applets or applications.

You already know that you do not need to create GUI components from scratch; Java's creators packaged the Swing components to inherit from the java.awt.Container class, so you can adapt them for your purposes. You insert the import statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing GUI components and their methods.

Components are also called widgets, which stands for windows gadgets.

When you use components in a Java Swing program, you usually place them in containers. A **container** is a type of component that holds other components so you can treat a group of several components as a single entity. Usually, a container takes the form of a window that you can drag, resize, minimize, restore, and close. Containers are defined in the Container class.

As you know, all Java classes are subclasses; they all descend from the Object class. The Component class is a child of the Object class, and the Container class is a child of the Component class. Therefore, every Container Object "is a" Component, and every Component Object (including every Container) "is an" Object.

The Container class is also a parent class. Its child is the Window class. Similarly, the Frame class is a subclass of the Window class, and the JFrame class is a subclass of the Frame class, as shown in Figure 13-6.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--javax.swing.JFrame
```

Figure 13-6 Relationship of the JFrame class to the Object, Component, Container, Window, and Frame superclass

Recall that the Object class is defined in the java.lang package, which is imported automatically every time you write a Java program.

The Component class is an abstract class. You learned in Chapter 12 that when you create an abstract class, you cannot create any concrete instances; instead, you create subclasses from which you create concrete instances. All GUI components, such as the buttons, text fields, and other objects with which the user interacts, are actually subclasses or extensions of the Component class. Likewise, the Container class, which descends from the Component class, is itself an abstract class. Therefore, there are no "plain" Containers; every concrete Container object is a member of a subclass of Container. The Window class, which inherits from Container, is not abstract; you can instantiate a Window object. However, Java programmers rarely use Window objects because the Window subclass Frame allows you to create more useful objects. Window objects do not have title bars or borders, but a Frame object does. As Figure 13-6 shows a JFrame"is a" Frame as well as a Window, a Container, a Component, and an Object, and therefore inherits all the methods of the parents.

You usually create a JFrame so that you can place other objects within it for display. The JFrame class has four constructors:

- JFrame() constructs a new frame that is initially invisible.
- JFrame(GraphicsConfiguration gc) creates a JFrame in the specified GraphicsConfiguration of a screen device and a blank title.
- JFrame(String title) creates a new, initially invisible JFrame with the specified title.
- JFrame(String title, GraphicsConfiguration gc) creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen.

JFrame objects constructed with the no-argument constructor are untitled. For example, the following two statements construct two JFrames—one with the title "Hello", and another JFrame with no title:

```
JFrame firstFrame = new JFrame("Hello");
JFrame secondFrame = new JFrame();
```

Next you will create a JFrame object that appears on the screen.

To create a JFrame object:

1. Open a new file in your text editor.
2. Type the following statement to import the java.swing classes: **`import javax.swing.*;`**.
3. On the next lines, type the following class header for the JDemoFrame class and its opening curly brace:

   ```
   public class JDemoFrame
   {
   ```
4. On the next lines, type the following main() method header and its opening curly brace:

   ```
   public static void main(String[] args)
   {
   ```
5. Within the body of the main() method, enter the following code to declare a JFrame with a title, set the JFrame's size, and make the JFrame visible. If you neglect to set a JFrame's size, you will see only the title bar of the JFrame. If you neglect to make the JFrame visible, you will not see anything at all. Add two closing curly braces—one for the main() method and one for the JDemoFrame class.

   ```
     JFrame aFrame = new JFrame("This is a frame");
     aFrame.setSize(200,100);
   ```

```
      aFrame.setVisible(true);
     }
    }
```

6. Save the file as **JDemoFrame.java** in the Chapter.13 folder on your Student Disk. Compile the class using the `javac` command, and then run the program using the `java` command. The output looks like Figure 13-7.

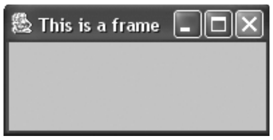


Figure 13-7 Output of the JDemoFrame program

The term frame means a generic GUI window that has a border and a title and may include buttons. You can create forms using many programming languages as well as by using Java.

The JFrame shown in Figure 13-7 resembles frames that you have seen when using different GUI programs. One reason to use similar frame objects in your programs is your program's user is already familiar with the frame environment. Users expect to see a title bar at the top of a frame that contains information (such as "This is a frame"). Users also expect to see Minimize, Maximize or Restore, and Close buttons in the frame's upper-right corner. Most users assume that they can change a frame's size by dragging its border, or reposition the frame on their screen by dragging the frame's title bar to a new location. Next you will confirm that the frame you just created has these capabilities.

To confirm that the JFrame you created has Minimize, Maximize, Restore, and dragging capabilities:

1. Run the **JDemoFrame** program again, if necessary. Click the JFrame's **Minimize** button. The JFrame minimizes to an icon on the Windows taskbar.
2. Click the JFrame's **icon** on the taskbar. The JFrame returns to its previous size.
3. Click the JFrame's **Maximize** button. The JFrame fills the screen.
4. Click the JFrame's **Restore** button to return the JFrame to its original size.
5. Position your mouse pointer on the JFrame's title bar, and then drag the JFrame to a new position on your screen.
6. Click the JFrame's **Close** button. The JFrame disappears because the default behavior is to simply hide the JFrame when the user closes the window. The cursor is left blinking in the command window until the application is closed. To close the application you must press **[Ctrl]+C**.

Tip

In Chapter 6, you learned to press [Ctrl]+C to stop a program that contains an infinite loop.

Using Additional JFrame Class Methods

When you extend the JFrame class, you inherit several useful methods. Table 13-1 lists the method header and purpose of several methods available to the JFrame class.

Table 13-1 Useful methods of the JFrame class

Method	Purpose
`void setTitle(String)`	Sets a JFrame's title
`void setSize(int, int)`	Sets a JFrame's size in pixels with the width and height as arguments
`void setSize(Dimension)`	Sets a JFrame's size using a Dimension class object by calling the Dimension(int, int) constructor that creates the object representing the specified width and height arguments
`String getTitle()`	Returns a JFrame's title
`void setResizable(boolean resizable)`	Sets the JFrame to be resizable by passing `true`, or sets the JFrame not to be resizable by passing `false` to the method
`boolean isResizable()`	Returns `true` or `false` to indicate whether the Frame is resizable
`void setVisible(boolean)`	Sets a JFrame visible using the Boolean argument `true` and invisible using the Boolean argument `false`
`void setBounds (int, int, int, int)`	Overrides the default behavior for the JFrame to be positioned in the upper-left corner of the computer's desktop. The first two arguments are the x and y position of the JFrame's upper-left corner on the desktop. The last two arguments set the width and height.

The syntax to use any of these methods is to use a JFrame object, a dot, and the method name. If you use any of these methods within a JFrame's class, then the method call to set the title is `this.setTitle("This is the title");`, or more simply, `setTitle("This is the title");`.

Earlier in this chapter you learned that when an application using a JFrame is closed, the normal behavior is for the application to keep running. To create a JFrame that ends the program when the user clicks the Close button, you can call the JFrame's setDefaultCloseOperation() method with the class variable EXIT_ON_CLOSE as an argument. There are four class variables that provide flexibility in handling the Close operation. These class variables and their ensuing actions that can be passed as an argument to the setDefaultCloseOperation() method include:

- EXIT_ON_CLOSE exits the program when the JFrame is closed.
- DISPOSE_ON_CLOSE closes the frame, disposes of the JFrame object, and keeps running the application.
- DO_NOTHING_ON_CLOSE keeps the JFrame and continues running.
- HIDE_ON_CLOSE closes the JFrame and continues running.

When a JFrame serves as a Swing application's main user interface, the normal behavior when a JFrame is closed is for the application to keep running. To exit a program when the JFrame is closed, add the following statement to the JFrame's constructor method: `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`. For example, to set a JFrame program named JDefault to close when the JFrame's Close button is clicked, you can write the following constructor code:

```
public JDefault()
{
  super("JDefault Example")
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//other statements
}
```

Using Swing Event Listeners

Classes that respond to user events must implement an interface that deals with the events. These interfaces are called **event listeners**. Each of these listeners can handle a specific event type, and a class can implement as many event listeners as it needs. Table 13-2 lists event listeners and the types of events for which they are used.

Table 13-2 Alphabetical listing of event listeners

Listener	Type of events	Example
ActionListener	Action events	Button clicks
AdjustmentListener	Adjustment events	Scrollbar moves
FocusListener	Keyboard focus events	Textfield gains or loses focus
ItemListener	Item events	Check box changes
KeyListener	Keyboard events	Text is entered from keyboard
MouseListener	Mouse events	Mouse clicks
MouseMotionListener	Mouse movement events	All mouse movements
WindowListener	Window events	Window closes
Change Listener	Slider events	Slider moves

In Table 13-3, each component that is created is associated with one of the methods to associate a listener with it:

Table 13-3 Swing components and their associated listeners

Components	Associated listeners
JButton, JCheckBox, JComboBox, JToolbar, JTextField, and JRadioButton	addActionListener()
JScrollBar	addAdjustmentListener()
all Swing Components	addFocusListener(), addKeyListener(), addMouseListener(), and addMouseMotionListener()
JButton, JCheckBox, JComboBox and JRadioButton	addItemListener()
all JWindow and JFrame Components	addWindowListener()
JSlider	addChangeEvent()

When a user event takes place, the appropriate method is called automatically by the system. For example, when an event occurs in a program that implements ActionListener, all classes that implement ActionListener must use a method with a structure similar to the following:

```
public void actionPerformed(ActionEvent event)
{
   //method to handle the event goes here
}
```

Failure to include the actionPerfomed() method will result in a compilation error. If more than one component has an action event listener, you must figure out which component was used and code accordingly in the program. In the example above, an ActionEvent object is sent as an argument when the method is called. ActionEvent is part of the java.awt.event package, and a subclass of the EventObject class.

This ActionEvent object that is an argument to actionPerformed() contains information about the object that caused the event—for example, perhaps a button click or a scrollbar movement generated the event. You can use several methods to discover the details about the event-generating object. For example, the getSource() method returns the identity of the component where the event occurred. Consider the following code example:

```
public void actionPerformed(ActionEvent event)
{
   Object source = event.getSource();
   if(source == answerButton)
   //take some action
   else
   //take some other action
}
```

In this example, the `if` statement tests if the source is `answerButton` and if `true` takes some action. An alternate method is to use the `instanceof` keyword to check what kind of object generated the event:

```
{
   Object source = event.getSource();
   if(source instanceof JButton)
   //take some action
   else
   //take some other action
}
```

In this example, if the source of the event is a JButton, the `if` statement evaluates to `true` and some action is taken.

Tip: You first learned the `instanceof` keyword in Chapter 9 when you read about action events of Swing applets.

Using JPanel Class Methods

Because it allows other components to be added directly to the container, the JPanel, found in the JPanel class, is the simplest Swing container. The structure of the JPanel class is shown in Figure 13-8.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
```

Figure 13-8 Structure of the JPanel class

To add a component to a JPanel, you call the Container's add() method, using the component as the argument. For example, the following code creates a JButton and adds it to a JPanel:

```
JButton exit = new JButton("Exit");
JPanel panel = new JPanel();
panel.add(exit);
```

Some Swing containers such as the JFrame and JApplet do not allow components to be added directly to the container. These Swing containers require that components added to containers must be broken down into panes.

Components are added to a container's content pane using the following steps:

1. Create a JPanel object.
2. Add components to the JPanel using the add() method.
3. Call the setContentPane() method with the panel object created to set the application's content pane.

For example, the following code creates a JPanel object named pane and a JButton object named button. Then the button object is added to the pane object, using the add() method with button as the argument. The pane object is set as the content pane, using the setContentPane() method with pane as the argument.

```
JPanel pane = new JPanel();
JButton button = JButton("Exit");
pane.add(button);
setContentPane(pane);
```

Swing components have a default size for the objects that are created. A component's size can be changed using the component's setPreferredSize() method. For example, a JButton's preferred size can be changed as follows:

```
Dimension buttonSize = new Dimension(200, 20);
JButton bigButton= new JButton();
bigButton.setPreferredSize(buttonSize);
```

In the example, the preferred size of the JButton named bigButton is changed to a width of 200 pixels and a height of 20 pixels using a Dimension object named buttonSize.

Next you will create a Swing application that displays a JFrame that uses a JPanel and some JButtons. This exercise will show you that you can add JPanels to a Swing application just as easily as you can add them to a Swing applet; it will also demonstrate some JComponent class methods and associated event methods. Within the Swing application, you create a JFrame and a JPanel containing three JButtons that change captions when the user clicks them.

To create a JFrame that displays three JButtons within a JPanel:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application. The import statements used before the class header definition make the Swing components, the AWT components, and the event listener available. Note that the JChangeMessage class `implements ActionListener`. Create a Dimension object to hold the width and height dimensions. Create three JButtons with captions of "North", "Center", and "South". Then create a BorderLayout object for the component layout.

   ```
   import javax.swing.*;
   import java.awt.*;
   import java.awt.event.*;
   public class JChangeMessage extends JFrame implements
      ActionListener
   ```

```
{
    Dimension size = new Dimension(250,20);
    JButton b1 = new JButton("North");
    JButton b2 = new JButton("Center");
    JButton b3 = new JButton("South");
    BorderLayout aBorder = new BorderLayout();
```

2. In the JChangeMessage constructor method, set the JFrame title to "Change Message" and the default close operation to `EXIT_ON_CLOSE`. Set the preferred size of each JButton by calling the setPreferredMethod() and using the Dimension object named size as the argument. Add an ActionListener for each JButton using the keyword `this` to represent the JFrame.

```
public JChangeMessage()
{
  super("Change Message");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  b1.setPreferredSize(size);
  b2.setPreferredSize(size);
  b3.setPreferredSize(size);
  b1.addActionListener(this);
  b2.addActionListener(this);
  b3.addActionListener(this);
```

3. Create a JPanel object named pane. Set the layout for pane to BorderLayout, with the setLayout() method using the aBorder object as an argument. Add three JButtons to the JPanel named pane using the layout locations "North", "Center", and "South". Then call the setContentPane() method with pane as the argument to make it the content pane.

```
    JPanel pane = new JPanel();
    pane.setLayout(aBorder);
    pane.add("North", b1);
    pane.add("Center", b2);
    pane.add("South", b3);
    setContentPane(pane);
}
```

4. Add the following main() method that creates a new JFrame named aFrame, sizes it using the setSize() method, and sets its visible property to `true`.

```
public static void main(String[] args)
{
    JFrame aFrame = new JChangeMessage();
    aFrame.setSize(275,100);
    aFrame.setVisible(true);
}
```

5. Enter the following actionPerformed() method which will execute when the user clicks a JButton. Notice that each JButton has its own listener. To determine

which JButton is clicked, an Object source is created by calling the ActionEvent's getSource() method. An **`if..else`** structure is used to determine which JButton is clicked. If the source of the JButton that sent the event is the b1 JButton, the caption of b1 is set to "Event Handlers Incorporated" using the setText() method. The same procedure is applied to each of the remaining JButtons. Add a closing curly brace to end the ChangeMessage class.

```
public void actionPerformed(ActionEvent e)
  {
    Object source = e.getSource();
    if (source == b1)
        b1.setText("Event Handlers Incorporated");
    else if (source == b2)
        b2.setText("Plan With Us");
    else if (source == b3)
        b3.setText("You just relax. We'll manage the
        fuss.");
  }
}
```

6. Save the file as **JChangeMessage.java** in the Chapter.13 folder on your Student Disk. Compile the file using the **`javac`** command. The output is shown in Figure 13-9 before any JButtons are clicked, and again in Figure 13-10 after all the JButtons are clicked.

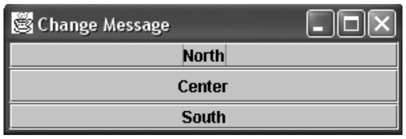


Figure 13-9 JChangeMessage program before any JButtons are clicked

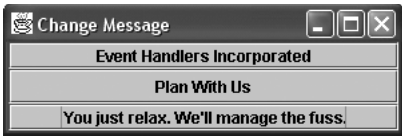

Change Message
Event Handlers Incorporated
Plan With Us
You just relax. We'll manage the fuss.

Figure 13-10 JChangeMessage program after all JButtons are clicked

USING THE JCHECKBOX CLASS

A JCheckBox consists of a JLabel positioned beside a square; you can click the square to display or remove a check mark. (Usually you use a JCheckBox to allow the user to turn an option on or off.) The structure of the JCheckBox class is shown in Figure 13-11.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.AbstractButton
                          |
                          +--javax.swing.JToggleButton
                                |
                                +--javax.swing.JCheckBox
```

Figure 13-11 Structure of the JCheckBox class

CheckBox methods are listed in Table 13-4.

Table 13-4 JCheckBox methods

Method	Purpose
`void setLabel (String label)`	Sets the label for the JCheckBox
`String getLabel()`	Returns the JCheckBox label
`void setState (boolean condition)`	Sets the JCheckBox state to `true` for checked or `false` for unchecked
`boolean getState()`	Gets the current state (checked or unchecked) of the JCheckBox

When you construct a JCheckBox, you can choose whether to assign it a label. The following statements create two JCheckBox objects—one with a label and one without a label:

```
JCheckBox boxOne = new JCheckBox();
JCheckBox boxTwo = new JCheckBox("Click here please");
```

If you do not initialize a JCheckBox with a label, or if you want to change the label later, you can use the setLabel() method, as in `boxOne.setLabel("Check this box now");`. You can set the state of a JCheckBox with the setState() method; for example, use `boxOne.setState(false);` to insure that boxOne is unchecked. The getState() method is most useful in Boolean expressions, as in `if(boxTwo.getState()) ++votes;`, which adds one to a votes variable if boxTwo is currently checked.

Using a JCheckBox object requires using a new interface, ItemListener. Whereas ActionListener provides for mouse clicks and requires that you write an actionPerformed() method, ItemListener provides for objects whose states change from `true` to `false` and requires that you write an itemStateChanged() method. When a check box's state is changed from checked to unchecked or from unchecked to checked, the code in the itemStateChanged() method executes.

You can call the getItem() method to determine the identity of the item that generated an event. To determine whether that item was selected or deselected you use the getChange() method. This method returns an integer that will be equal to either the class variable ItemEvent.SELECTED or ItemEvent.DESELECTED. For example, in the following itemStateChanged() method code, the getItem() method is called and returns the object named source. Then source is tested in an `if` statement to determine if it is equal to another JCheckBox object named checkBox. If the two objects are equal, then the getStateChange() method is called and returns an integer value that is assigned to the integer named select. The value of the select is compared to the class field ItemEvent.SELECTED, and if they are equal, a set of program statements executes. If they are not equal, the program statements following the `else` execute.

```
public void itemStateChanged(ItemEvent e)
  {
    Object source = e.getItem();
    if (source == checkBox)
    {
      int select = e.getStateChange();
      if(select == ItemEvent.SELECTED)
      //some statements
      else
      //other statements
    }
  }
```

Next you will create an interactive program that Event Handlers Incorporated clients can use to determine an event's price. The base price of an event is $500; serving cocktails adds $300, and serving dinner adds $200. The user can check and uncheck the cocktail and dinner check boxes to recalculate the event price.

To write a Swing application that includes two JCheckBox objects:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application that demonstrates the use of a JCheckBox. Note that the JDemoCheckBox class `implements ItemListener`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoCheckBox extends JFrame implements
   ItemListener
{
```

2. Create a FlowLayout object named flow to be used as an argument to set the layout of the program to FlowLayout. Create two JCheckBoxes named cocktailBox and dinnerBox, both unchecked, by setting their second argument to `false`. Create two JLabels to hold the headings "Event Handlers Incorporated" and "Event Price Estimate". Create a JTextField object named totPrice and a String variable named output. The JTextField is used to display the total price calculated for the event. The total price of the event is calculated from the prices of the individual event items selected and stored in the String variable named output.

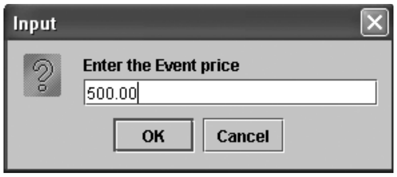
```
FlowLayout flow = new FlowLayout();
JCheckBox cocktailBox = new JCheckBox("Cocktails",
   false);
```

```
JCheckBox dinnerBox = new JCheckBox("Dinner", false);
JLabel aEvent = new JLabel
   ("Event Handlers Incorporated");
JLabel ePrice = new JLabel("Event Price estimate");
JTextArea totPrice = new JTextArea(1,10);
String output;
```

3. Using the JOptionPane showInputDialog()method, accept keyboard input for the price of cocktails, dinner, and the event. Use the parseDouble() method to change the input String objects to the doubles named cocktailPrice, dinnerPrice, and totalPrice.

```
   String firstPrice = JOptionPane.showInputDialog
      ("Enter the Cocktail price");
   String secondPrice = JOptionPane.showInputDialog
      ("Enter the Dinner price");
   String thirdPrice = JOptionPane.showInputDialog
      ("Enter the Event price");
   double cocktailPrice = Double.parseDouble(firstPrice);
   double dinnerPrice = Double.parseDouble(secondPrice);
   double totalPrice = Double.parseDouble(thirdPrice);
```

Figure 13-12 shows what the last dialog box will look like when the program is complete.



Enter the Event price

500.00

Figure 13-12 Input dialog box for the price of the event

4. Create the constructor method statement `public JDemoCheckBox()`. Add an opening curly brace, press **[Enter]**, and set the JFrame's title to "Check Box". Set the value of the JFrame's setDefaultCloseOperation to EXIT_ON_CLOSE so that the JFrame will close when the Close button is clicked. Add a JPanel named pane that will act as the content pane, and set the layout of the JPanel to FlowLayout. Add the cocktailBox and dinnerBox to the JPanel. Add the two JLabels and the JTextField to the JPanel. Note that you must add these components to the JPanel in the top-to-bottom and left-to-right order in which they are to appear. Use the setText() method to set the initial text of totPrice to the String variable thirdPrice which holds the event price captured from keyboard input. Register the cocktailBox and dinnerBox by adding an ItemListener for each, and then use the setContentPane() method to set the JPanel as the content pane for the program.

```
public JDemoCheckBox()
{
  super("Check Box");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel pane = new JPanel();
  pane.setLayout(flow);
  pane.add(cocktailBox);
  pane.add(dinnerBox);
  pane.add(aEvent);
  pane.add(ePrice);
  pane.add(totPrice);
  totPrice.setText(thirdPrice);
  cocktailBox.addItemListener(this);
  dinnerBox.addItemListener(this);
  setContentPane(pane);
}
```

5. Add the following main() method that creates a new JFrame named aFrame, sizes it using the setSize() method, and sets its visible property to `true`:

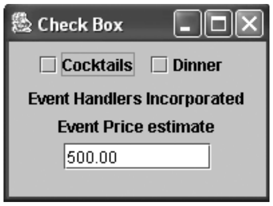
```
public static void main(String[] args)
{
 JFrame aFrame = new JDemoCheckBox();
 aFrame.setSize(200,150);
 aFrame.setVisible(true);
}
```

6. Enter the following itemStateChanged() method, which executes when the user changes the status of one of the two JCheckBoxes that are registered as ItemListeners. The base price of the event is set at $500. If the cocktail JCheckBox is checked, then the program adds the cocktail price ($300) to the event total price. If the dinner JCheckBox is checked, the program adds the dinner price ($200) to the event total price. If either of the JCheckBoxes is subsequently unchecked, the appropriate prices are subtracted from the total price. Note that the source of an ItemEvent is determined using the getItem() method. Note also that whether or not the ItemEvent object is checked is determined using the getStateChange() method. Finally, observe that the variable output holds the total price as a string. This is accomplished by assigning to output the result of concatenating an empty string " " + totalPrice which forces the result to yield a string representation.

```
public void itemStateChanged(ItemEvent check)
{
  Object source = check.getItem();
  if (source == cocktailBox)
  {
    int select = check.getStateChange();
    if(select == ItemEvent.SELECTED)
    {
```

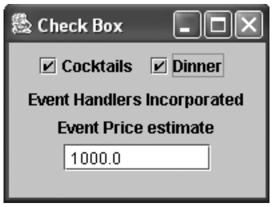
```
              totalPrice = totalPrice + cocktailPrice;
              output = " " + totalPrice;
              totPrice.setText(output);
            }
            else if(select == ItemEvent.DESELECTED)
            {
              totalPrice = totalPrice - cocktailPrice;
              output = " " + totalPrice;
              totPrice.setText(output);
            }
          }
          if(source == dinnerBox)
          {
            int select = check.getStateChange();
            if(select == ItemEvent.SELECTED)
            {
              totalPrice = totalPrice + dinnerPrice;
              output = " " + totalPrice;
              totPrice.setText(output);
            }
            else if(select == ItemEvent.DESELECTED)
            {
              totalPrice = totalPrice - dinnerPrice;
              output = " " + totalPrice;
              totPrice.setText(output);
            }
          }
        }
      }
```

7. Save the file as **JDemoCheckBox.java** in the Chapter.13 folder on your Student Disk. Compile the file using the `javac` command. Run the program with `java JDemoCheckBox`. The initial event cost of $500 is shown in Figure 13-13 before any boxes have been selected. Click the boxes of your choice and observe how the total price of the event changes in response to your selections. The total cost of the event, $1000, is shown in Figure 13-14, with cocktails and dinner selected.



```
Event Handlers Incorporated
Event Price estimate
500.00
```

Figure 13-13 Output of JDemoCheckBox before any boxes have been selected



Event Handlers Incorporated
Event Price estimate
1000.0

Figure 13-14 Output of JDemoCheckBox with both the Cocktails and the Dinner boxes selected

USING THE BUTTONGROUP CLASSES

You can group several JCheckBoxes so a user can select only one at a time. When you group JCheckBox objects, all other JCheckBoxes are automatically turned off when the user selects any one check box.

Tip: A group of JCheckBoxes is similar to a set of radio buttons that you can create using the JRadioButton class. However, when you place check boxes in a group, any number can be selected. A group of radio buttons can have, at most, one button selected at a time.

To organize check boxes into a group that allows only one at a time to be selected you must create a ButtonGroup class object. You can either create a ButtonGroup and then create the individual JCheckBoxes, or you can create the JCheckBoxes and then create the ButtonGroup. The structure of the ButtonGroup class is shown in Figure 13-15.

```
java.lang.Object
   |
   +--java.swing.ButtonGroup
```

Figure 13-15 Structure of the ButtonGroup class

To create a ButtonGroup, and then add a JCheckBox:

1. Create a ButtonGroup such as `ButtonGroup aGroup = new ButtonGroup();`.
2. Create a JCheckBox such as `JCheckBox aBox = new CheckBox();`.
3. Add `JCheckBox aBox` to `ButtonGroup aGroup` as `aGroup.add(aBox);`.

For example, if you define a ButtonGroup as `ButtonGroup favoriteStoogeGroup = new ButtonGroup();`, then you can assign an unselected JCheckBox to the group, and add the JCheckBox object to the ButtonGroup using the following code:

```
JCheckBox larryBox = new JCheckBox("Larry", false);
favoriteStoogeGroup.add(larryBox);
```

You can the use the following statement to assign `JCheckBox moeBox = new JCheckBox("Moe", true);` to the ButtonGroup favoriteStoogeGroup:

`favoriteStoogeGroup.add(moeBox);`.

Tip: If you assign the `true` state to multiple JCheckBoxes within a group, each new `true` assignment negates the previous one because only one box can be selected within a group.

You can set one of the JCheckBoxes within a group to "on" by clicking it with the mouse, or you can select a JCheckBox within a ButtonGroup with a statement such as `favoriteStoogeGroup.setSelected(larryBox);`. You can determine which, if any, of the JCheckBoxes in a ButtonGroup are selected with the isSelected() method. For example, the statement: `if(favoriteStoogeGroup.isSelected());` evaluates to `true` if the favoriteStoogeGroup is selected.

Tip: Each JCheckBox object has access to every JCheckBox class method regardless of whether the JCheckBox is part of a ButtonGroup.

Next you will add a ButtonGroup to the program that determines the price of an event for Event Handlers Incorporated. If the user wants to serve dinner at an event, the price varies based on the selected menu. Because the user can choose only one entrée (chicken, beef, or fish), it is appropriate to select the entrée using a ButtonGroup.

To add a ButtonGroup to the Event Handlers pricing program:

1. In your text editor, open the **JDemoCheckBox.java** file from the Chapter.13 folder on your Student Disk, and then save it as **JDemoButtonGroup.java**.
2. Delete the old class header and type the new class header: **`public class JDemoButtonGroup extends JFrame implements ItemListener`**.
3. Position your insertion point at the end of the statement `JCheckBox dinnerBox = new JCheckBox("Dinner", false);`, press **[Enter]** to open up a blank line. To add new JCheckBoxes for the remaining two entrées, beefBox and fishBox, initializing each entrée box to be unchecked add the following statements:

   ```
   JCheckBox beefBox = new JCheckBox("Beef", false);
   JCheckBox fishBox = new JCheckBox("Fish", false);
   ```

4. Position your insertion point at the end of the statement `double totalPrice = Double.parseDouble(thirdPrice);`, and then press **[Enter]**. Then add the following two new variables for the dinner price if the selected entrée is either beef or fish: **`int beefPrice = 300, fishPrice = 500;`**.

5. Delete the old JDemoCheckBox() constructor method, and then type the new constructor method as **`JDemoButtonGroup()`**. Change the title of the JFrame by changing the argument of the super() method to "Button Group": **`super("Button Group");`**

6. Position the insertion point in the JDemoButtonGroup() method at the end of the line `pane.setLayout(flow);`, and then press **[Enter]**. Type the following code to add a ButtonGroup to hold the three dinner options. Then add the dinnerBox (default dinner entrée), beefBox, and fishBox to the new dinnerGroup.

```
ButtonGroup dinnerGroup = new ButtonGroup();
dinnerGroup.add(dinnerBox);
dinnerGroup.add(beefBox);
dinnerGroup.add(fishBox);
```

7. Position your insertion point to the right of the statement `pane.add(dinnerBox);`, and then press **[Enter]**. Add the beefBox and fishBox to the pane as follows:

```
pane.add(beefBox);
pane.add(fishBox);
```

8. Position your insertion point to the right of the statement `dinnerBox.addItemListener(this);`, and then press **[Enter]** to start a new line. Enter the following code to register a listener for the beefBox and fishBox. It is necessary to register listeners for each of the JCheckBoxes, rather than a listener for the ButtonGroup.

```
beefBox.addItemListener(this);
fishBox.addItemListener(this);
```

9. Within the main() method, delete the statement `JFrame aFrame = new JDemoCheckBox();`, and then add **`JFrame aFrame = new JDemoButtonGroup();`**. Delete the statement **`aFrame.setSize(200,150);`**, and then add **`aFrame.setSize(300,150);`**.

10. Place your insertion point at the end of the closing brace of the `if(source == dinnerBox)` statement, and then press **[Enter]**. Add the following changes to the itemStateChanged() methods, which execute when the user changes the state of beefBox or fishBox:

```
if (source == beefBox)
{
     int select = check.getStateChange();
     if(select == ItemEvent.SELECTED)
     {
        totalPrice = totalPrice + beefPrice;
        output = " " + totalPrice;
        totPrice.setText(output);
     }
```

```
                    else if(select == ItemEvent.DESELECTED)
                    {
                        totalPrice = totalPrice - beefPrice;
                        output = " " + totalPrice;
                        totPrice.setText(output);
                    }
                 }
                 if (source == fishBox)
                 {
                     int select = check.getStateChange();
                     if(select == ItemEvent.SELECTED)
                     {
                          totalPrice = totalPrice + fishPrice;
                          output = " " + totalPrice;
                          totPrice.setText(output);
                     }
                     else if(select == ItemEvent.DESELECTED)
                     {
                        totalPrice = totalPrice - fishPrice;
                        output = " " + totalPrice;
                        totPrice.setText(output);
                     }
                 }
```

11. Save the file in the Chapter.13 folder on your Student Disk. Compile the program using the **`javac`** command, and then run it. The output should look similar to Figure 13-16 if you selected Cocktails and Beef. Note that selecting Beef added an additional $300 to the event price.

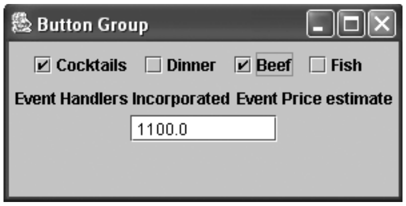

Button Group
☑ Cocktails ☐ Dinner ☑ Beef ☐ Fish
Event Handlers Incorporated Event Price estimate
1100.0

Figure 13-16 Output of the JDemoButtonGroup Swing application

Creating a Drop-Down List and Combo Box Using the JComboBox Class

The Swing package contains a single JComboBox class for picking items from a list or entering text into a field. Another option is a drop-down list, also called a choice list, which is a component that enables a single item to be chosen from a list of items. This list is usually configured to appear when the user clicks the component. A drop-down list can also be configured to be a combo box. If the setEditable() method of the component is called with the argument `true`, it becomes a combo box that the user can use to enter text into a field.

The default behavior of the JComboBox class is to display an option; by clicking the JComboBox object, a drop-down menu that contains a list of items appears. When the user selects an item from the drop-down list, the selected item replaces the original option in the display. The structure of the JComboBox class is shown in Figure 13-17.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JComboBox
```

Figure 13-17 Structure of the JComboBox class

You can build a JComboBox by using a constructor with no arguments, and then adding items to the list with the addItem() method. For example, the following statements create a JComboBox object with three options:

```
JComboBox majorChoice = new JComboBox();
majorChoice.addItem("English");
majorChoice.addItem("Math");
majorChoice.addItem("Sociology");
```

Table 13-5 lists the methods you can use with a JComboBox object. Using the select() method, you choose one of the items in a JComboBox to be the initially selected item, as in `majorChoice.select("Math");`. You can extract the text of a JComboBox object and assign it to a String variable, as in `String myMajor = majorChoice.getSelectedItem();` with the getSelectedItem() method.

Table 13-5 JComboBox class methods

Method	Purpose
`String getItemAt(int)`	Returns the text of the list item at the index position specified by the integer argument. The first item of a choice list is at index position zero.
`String getSelectedItem()`	Returns the text of the currently selected item
`int getItemCount()`	Returns the number of items in the list
`int getSelectedIndex()`	Returns the item in the list that matches the given item
`void setSelectedIndex(int)`	Selects the item at the position indicated (by the integer argument)
`void setSelectedItem(Object)`	Selects the specified object in the list
`void setMaximumRowCount(int)`	Sets the maximum number of combo box rows that are displayed at one time

You can also treat the items in a JComboBox object as a zero-based array. For example, you can use the getSelectedIndex() method to determine the list position of the currently selected item. Then you can use the index to access corresponding information. For example, if a JComboBox named historyChoice has been filled with a list of historical events, such as "Declaration of Independence," "Pearl Harbor," and "Man walks on moon," then after a user chooses one of the items, you can code `int pos = historyChoice.getSelectedIndex();`. Now the variable pos holds the position of the selected item, and you can use the pos variable to access an array of dates so you can display the appropriate one. For example, if `int[] dates = {1776, 1941, 1968};`, then `dates[pos]` holds the year for the selected historical event.

Next you will create a Swing application for Event Handlers Incorporated that allows the user to choose a party favor and displays the favor price. The party favor types are none ($0), Hats ($725), Streamers ($325), Noise Makers ($125), or Balloons ($135).

To write a drop-down list using a JComboBox:

1. Open a new file in your text editor, and then add the necessary import statements followed by the first few lines of the JDemoList program. The JDemoList program uses the JComboBox to create a drop-down list. Note that JDemoList `implements ItemListener`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoList extends JFrame implements
   ItemListener
{
```

2. Create a FlowLayout object named flow to be used as an argument to set the layout of the program to FlowLayout. Create a JComboBox object named favorBox, and a JLabel object named favorList to hold the list options. Add another JLabel to hold a heading, "Event Handlers Incorporated", and a JTextField for displaying the output generated from selecting a party favor from the list.

```
    FlowLayout flow = new FlowLayout();
    JComboBox favorBox = new JComboBox();
    JLabel favorList = new JLabel("Favor List");
    JLabel aEvent = new
       JLabel("Event Handlers     Incorporated");
    JTextField totPrice = new JTextField(10);
```

3. Enter the following code to create an array of integers to hold the five prices for the five party favor types. Add an integer variable to hold the price of the selected favor, a String variable to hold the output from the party favor selection, and an int variable to hold the index number of the party favor items in the party favor list.

```
int[] favorPrice = {0, 725, 325, 125, 135};
int totalPrice = 0;
```

```
String output;
int FavorNum;
```

4. Add the JDemoList() constructor, set the title of the JFrame to "JDemoList", and then set the value of the JFrame's setDefaultCloseOperation to EXIT_ON_CLOSE so that the JFrame will close when the Close button is clicked.

```
    public JDemoList()
    {
        super("JDemoList");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

5. Enter the text to create a JPanel named pane, and set the layout for the pane using the FlowLayout object named flow. Add the favorList object to the pane, register the ItemListener for the favorBox object, and add the names of five items to the JComboBox list:

```
        JPanel pane = new JPanel();
        pane.setLayout(flow);
        pane.add(favorList);
        favorBox.addItemListener(this);
        favorBox.addItem("None");
        favorBox.addItem("Hats");
        favorBox.addItem("Streamers");
        favorBox.addItem("Noise Makers");
        favorBox.addItem("Balloons");
```

6. Enter the following add() methods to add favorBox, aEvent, and totPrice to the pane object. Set the content pane for the application as the pane object, and then type the closing brace for the JDemoList() constructor.

```
        pane.add(favorBox);
        pane.add(aEvent);
        pane.add(totPrice);
        setContentPane(pane);
    }
```

7. Add the following main() method that creates a new JDemoList object named frame, sizes it using the setSize() method, and sets its visible property to `true`:

```
public static void main(String[] arguments)
    {
        JDemoList frame = new JDemoList();
        frame.setSize(200,150);
        frame.setVisible(true);
    }
```

8. Enter the following itemStateChanged() method, which determines the index of the selected party favor type and prints the correct price based on the index. When the user clicks the JComboBox option, a drop-down list appears. When an item is selected, and the getSource() method is used to

determine the item selected, the getSelectedIndex() method identifies the index position of the selected item. The index position is assigned to the integer variable favorNum, and the price of the item selected is assigned to the totPrice variable.

```
public void itemStateChanged(ItemEvent list)
{
    Object source = list.getSource();
    if (source == favorBox)
    {
     int favorNum = favorBox.getSelectedIndex();
     totalPrice = favorPrice[favorNum];
     output = "Favor Price $" + totalPrice;
     totPrice.setText(output);
    }
}
```

9. Add the closing curly brace for the class.
10. Save the file as **JDemoList.java** in the Chapter.13 folder on your Student Disk, compile it using the **`javac`** command, and run it with the command **`java JDemoList`**. If you selected the Streamers party favor from the drop-down list, your output should look like Figure 13-18. Make another choice from the list, and then observe the new favor price.

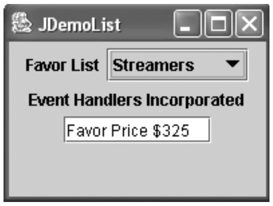


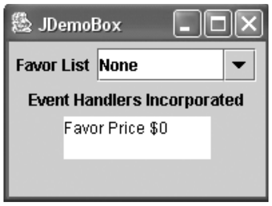
Figure 13-18 JDemoList output with Streamers selected

It is possible to create a combo box from a drop-down list. If the JComboBox component object's method setEditable() is called with `true` as an argument, the user can make a choice by entering the text of a list item in a JTextField and pressing [Enter], rather than using the drop-down list to make a selection. Next you will modify the JDemoList program so the user can make a choice by entering the text of a list item.

To allow the user to make a choice by entering a list item's name:

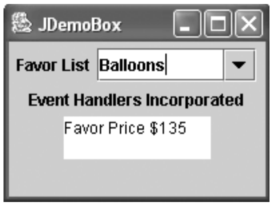
1. Open the **JDemoList** program if necessary, and then change the class name to **`JDemoBox`**.
2. Delete the JDemoList() constructor. Add the new constructor by typing: **`public JDemoBox()`**. Change the title of the JFrame by changing the argument of the super() method to **`super("JDemoBox");`**.

3. Position your insertion point at the end of the statement **`pane.add (favorBox);`**, and then press **[Enter]** to start a new line. Add the statement to set the favorBox object's setEditable() method to `true` by typing: **`favorBox.setEditable(true);`**. This statement causes the JComboBox to behave as a combo box and allow the user to enter text.
4. Delete the statement **`JDemoList frame = new JDemoList();`** in the main() method. Add the new statement that creates a JFrame named frame by typing: **`JDemoBox frame = new JDemoBox();`**.
5. Save the file as **JDemoBox.java** in the Chapter.13 folder on your Student Disk. Compile it using the **`javac`** command, and then run the class with the command **`java JDemoBox`**. Your output should look like Figure 13-19. Select the default selection ("None") and press **[Del]** to delete it. Type **Balloons** (be sure to type a capital B) in the text box. Press **[Enter]**. Press **[Enter]** and notice the Favor Price is now included, as shown in Figure 13-20.



Favor Price $0

Figure 13-19 Output of JDemoBox program prior to Favor selection



Favor Price $135

Figure 13-20 Output of JDemoBox program after [Enter] is pressed

CREATING JSCROLLPANES

When components in a Swing GUI are bigger than the area available to display them, you can add a scroll pane container. It is common to add a JTextArea component to a scroll pane container because it allows you to use both multiple rows and columns. The JScrollPane class is a container whose methods can be used to hold any component that can be scrolled. Figure 13-21 displays the structure of the JScrollPane class.

```
java.lang.Object
   |
   +--java.awt.Component
         |
         +--java.awt.Container
               |
               +--javax.swing.JComponent
                     |
                     +--javax.swing.JScrollPane
```

Figure 13-21 Structure of the JScrollPane class

The JScrollPane constructor takes one of four forms:

- JScrollPane() creates an empty JScrollPane where both horizontal and vertical scrollbars appear when needed.
- JScrollPane(Component) creates a JScrollPane that displays the contents of the specified component.
- JScrollPane(Component, int, int) creates a JScrollPane that displays the specified component, vertical scrollbar, and horizontal scrollbar.
- JScrollPane(int, int) creates a scroll pane with specified vertical and horizontal scrollbars.

A simple scroll pane can be created with the JScrollPane() constructor as follows:

```
JScrollPane scroll = JScrollPane();
```

The horizontal and vertical scrollbars will appear if they are needed. User control of the horizontal and vertical scrollbar configuration is achieved by using class variables of the ScrollPaneConstants class. Each of the following constants can be used for the horizontal and vertical scrollbar:

- HORIZONTAL_SCROLLBAR_AS_NEEDED
- HORIZONTAL_SCROLLBAR_ALWAYS
- HORIZONTAL_SCROLLBAR_NEVER
- VERTICAL_SCROLLBAR_AS_NEEDED
- VERTICAL_SCROLLBAR_ALWAYS
- VERTICAL_SCROLLBAR_NEVER

The following code creates a scroll pane with a vertical scrollbar and no horizontal scrollbar:

```
JScrollPane scroll = new JScrollPane(area,
  VERTICAL_SCROLLBAR_ALWAYS,
  HORIZONTAL_SCROLLBAR_NEVER);
```

A JTextArea can be created and added as a component to the scroll pane using the following code:

```
JTextArea area = new JTextArea(10,40);
JScrollPane scroll = new JScrollPane(area);
```

Notice that first a JTextArea object area is created. Then the JScrollPane(component) constructor is called with the JTextArea object named area as an argument, and the constructor creates a JScrollPane object named scroll. Text that is subsequently added to the JTextArea is viewable within the JScrollPane object created. When the added text fills more than the allotted 10 rows or 40 columns, a subsequent vertical (more than 10 rows) or horizontal (more than 40 columns) scrollbar appears automatically.

CREATING JTOOLBARS

A Swing GUI can be designed so that a user can move a toolbar from one section of a graphical user interface to another section. This type of a toolbar is called a **dockable** toolbar; the process that allows you to move and then attach the toolbar is called **docking**. A toolbar is created in Swing with the JToolBar class. The structure of the JToolBar class is shown in Figure 13-22.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JToolBar
```

Figure 13-22 Structure of the JToolBar class

The most common toolbar is a container that groups several components, usually JButtons, into a row or column. Constructor methods for the JToolBar class include the following:

- JToolBar() creates a new toolbar that will line up components in a horizontal direction.
- JToolBar(int) creates a new toolbar with a specified orientation of horizontal or vertical.

You can use the HORIZONTAL and VERTICAL constants from the SwingConstants class to explicitly set the orientation. For example, the following statement creates a toolbar that will line up components vertically:

```
JToolBar tbar = new JToolBar(SwingConstants.VERTICAL);
```

After you create a toolbar, you can add components to it using the toolbar's add() method. Continuing the previous example, the statements `JButton b1 = new JButton();`, followed by the statement `tbar.add(b1);` add the JButton named b1 to an existing JToolBar named tbar.

To make a JToolBar **dockable**, that is, attached to the edge of a screen much like a boat is tied to a dock, the JToolBar component must use the layout manager BorderLayout.

The container should use only two of the possible north, east, south, west, and center areas. One of the two areas must be the center area.

Tip

Chapter 14 covers the different layout managers in more detail. Recall that both the FlowLayout and the BorderLayout managers have been used in this chapter and in Chapter 9.

It is common practice to use toolbar components that are labels and buttons with images. The images are created using the ImageIcon class. The ImageIcon class was introduced in Chapter 9 in conjunction with the creation of JApplets. A JButton can have both an icon and a text label. For example the following statements demonstrate how to create an ImageIcon object, and then create a JButton object with both an icon and a text label:

```
ImageIcon picture = new ImageIcon("picture.gif");
JButton both = new JButton("My Text", picture);
```

The JButton will appear with the text "My Text" and the graphic named "picture.gif".

Next you will combine the scroll pane and text area objects (described in the previous section) with a toolbar to create a dockable toolbar Swing application.

Create a dockable toolbar application with a text area:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application that demonstrates the creation of a dockable toolbar. Note that the class `implements ActionListener` will be included.

   ```
   import javax.swing.*;
   import java.awt.*;
   import java.awt.event.*;
   public class JDemoToolBar extends JFrame implements
      ActionListener
   {
   ```

2. Create the necessary BorderLayout object required by a dockable toolbar. Create a JTextArea with space for eight lines of thirty characters each. Create a scroll pane object named scroll with three arguments—the first argument is the text area Component Object named edit, the second and third arguments are the scroll pane constants for including a vertical scrollbar and no horizontal scrollbar. The scroll pane will have a vertical scrollbar that allows the text area to hold rows of text larger than its row size.

   ```
     BorderLayout bord = new BorderLayout();
     JTextArea edit = new JTextArea(8,30);
     JScrollPane scroll = new JScrollPane(edit,
       ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
       ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
   ```

3. Create a JPanel object named pane to be used as the content pane. The JPanel will hold the completed toolbar and scroll pane. Next create three ImageIcon objects to hold the three images to be displayed on the buttons. Create three JButtons with constructors that take both a text object and an ImageIcon object as arguments. Then create a JToolBar() object named bar to hold the three JButton objects.

```
  JPanel pane = new JPanel();
  ImageIcon image1 = new ImageIcon("dining.gif");
  ImageIcon image2 = new ImageIcon("mail.gif");
  ImageIcon image3 = new ImageIcon("phone.gif");
  JButton b1 = new JButton("Dining", image1);
  JButton b2 = new JButton("Mail", image2);
  JButton b3 = new JButton("Phone", image3);
  JToolBar bar = new JToolBar();
```

4. Type the JDemoToolBar() constructor and add an opening curly brace. Use the super() method with the argument "Event Handlers Toolbar" to set the title of the JFrame. Use the setDefaultCloseOperation() method to cause the JFrame to close when the Close button is clicked. Add each of the three JButtons to the toolbar using the toolbar's add() method.

```
public JDemoToolBar()
  {
    super("Event Handlers Toolbar");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    bar.add(b1);
    bar.add(b2);
    bar.add(b3);
```

5. Register each JButton with an ActionListener and set the pane object as the content pane. Then add the closing curly brace to the JDemoToolBar() constructor.

```
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
    pane.setLayout(bord);
    pane.add(bar);
    pane.add(scroll);
    pane.add(bar,BorderLayout.NORTH);
    setContentPane(pane);
  }
```

6. Add the following main() method that creates a new JFrame named tFrame, sizes it using the setSize() method, and then sets its visible property to `true`:

```
  public static void main(String[] arguments)
  {
   JFrame tFrame = new JDemoToolBar();
   tFrame.setSize(400,200);
   tFrame.setVisible(true);
  }
```

7. Enter the following actionPerformed() method, which executes when the user clicks one of the three JButtons that are registered as ActionListeners. When the b1 JButton is clicked, a list of corporate event choices appears in the scroll pane text area. When the b2 JButton is clicked, the Event Handlers Incorporated name and address are added to the existing scroll pane text area. When the b3 JButton is clicked, the Event Handlers Incorporated name and phone numbers are appended to the existing scroll pane text area. Note also the use of the newline character "\n" to format a new line when it is contained in the string text. At the end of the actionPerformed() method add a closing curly brace for the class.

```
public void actionPerformed(ActionEvent event)
{
      Object source = event.getSource();
      if (source == b1)
      {
        edit.append("\nOur Event Choices are:\n"
        + "Corporate, Private, and Nonprofit\n"
        + "with cocktails, dinner, and party favors\n");
      }
      else if (source == b2)
      {
        edit.append("\nOur address is:\n"
        + "Event Handlers Incorporated\n"
        + "8900 U.S. Hwy 14\n"
        + "Crystal Lake, IL 60014\n");
      }
      else if (source == b3)
      {
        edit.append("\nOur Telephone numbers are:\n"
        + "1-800-656-4576\n"
        + "575-656-5879\n");
      }
    }
}
```

8. Save the file as **JDemoToolBar.java** in the Chapter.13 folder on your Student Disk. Compile the file using the `javac` command. Run the program with `java JDemoToolBar`. The output of the Swing application before any toolbar buttons are clicked is shown in Figure 13-23.

9. Click the **waiter** icon on the toolbar. A list of corporate event choices appears in the scroll pane text area, as shown in Figure 13-24.

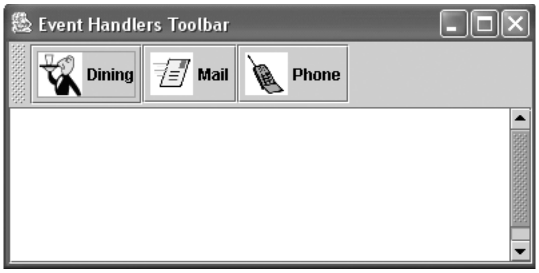
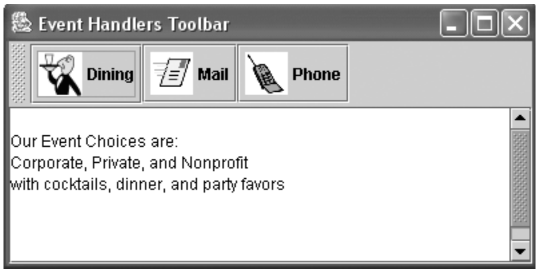


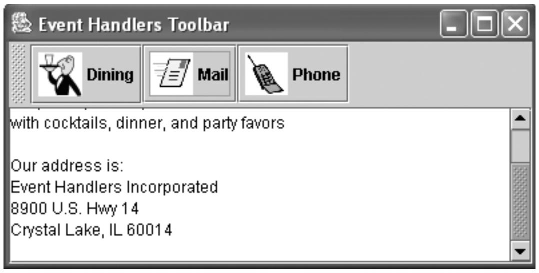
Figure 13-23 Output of the Swing application before any toolbar buttons are clicked



```
Our Event Choices are:
Corporate, Private, and Nonprofit
with cocktails, dinner, and party favors
```

Figure 13-24 Output of a list of corporate event choices

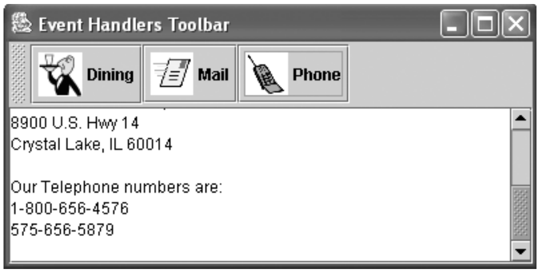
10. Click the **letter** icon on the toolbar. The Event Handlers Incorporated name and address are appended to the existing scroll pane text area. Your output should look like Figure 13-25.



```
with cocktails, dinner, and party favors

Our address is:
Event Handlers Incorporated
8900 U.S. Hwy 14
Crystal Lake, IL 60014
```

Figure 13-25 Output of Event Handlers Incorporated name and address

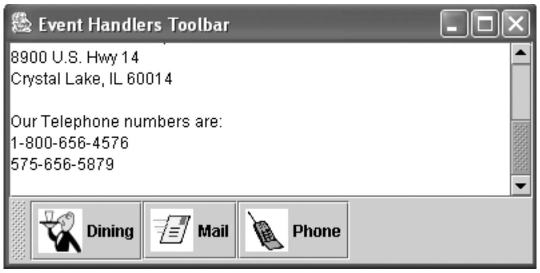
11. Click the **cell phone** icon on the toolbar. The Event Handlers Incorporated telephone numbers are added to the existing scroll pane text area, as shown in Figure 13-26.



```
8900 U.S. Hwy 14
Crystal Lake, IL 60014

Our Telephone numbers are:
1-800-656-4576
575-656-5879
```

Figure 13-26 Output of Event Handlers Incorporated telephone numbers

12. Grab the toolbar by its handle (the dotted area to the left of buttons), and drag the toolbar to the bottom of the application window. When you release the toolbar, the application is rearranged using the BorderLayout manager. The new position of the toolbar is shown in Figure 13-27. You can even drag the toolbar entirely outside the application (not shown).



```
8900 U.S. Hwy 14
Crystal Lake, IL 60014

Our Telephone numbers are:
1-800-656-4576
575-656-5879
```

Figure 13-27 Output of JToolBar at the new border layout position

Chapter Summary

- You insert the import statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing GUI components and their methods.
- Within the awt package, components are defined in the Component class. When you use components in a Java program, you usually place them in containers. A Container "is a" component that holds other components so you can treat a group of several components as a single entity. Containers are defined in the Container class.
- The JFrame class "is a" subclass of the awt Component class. The JFrame is the best container option for hosting Java applications. Used in conjunction with the Component class, the setSize() method allows you to set the physical size of a JFrame and the setVisible() method makes the JFrame component visible or invisible to the user. You usually create a JFrame so you can place other objects within it for display using a JPanel.

- Both the Container and Component classes are abstract classes. All of the GUI components, such as buttons, text fields, and other objects with which the user interacts, are subclasses or extensions of the Component class.
- A JFrame's action in response to a user clicking the Close button is set by passing an argument to the setDefaultCloseOperation() method placed inside the JFrame constructor method. The most common action is to close the application using the argument JFrame.EXIT_ON_CLOSE.
- Classes that respond to user events must implement an interface that deals with the events. These interfaces are called event listeners. Each of these listeners can handle a specific event type, and a class can implement as many event listeners as it needs.
- The simplest Swing container is the JPanel found in the JPanel class. To add a component to the JPanel you call the container's add() method, using the component as the argument.
- Using a JCheckBox Object requires using the interface, ItemListener. Whereas the interface ActionListener provides for mouse clicks and requires you to write an actionPerformed() method, ItemListener provides for objects whose states change from `true` to `false` and requires you to write an itemStateChanged() method.
- Since every event handling method is sent an event object of some type, the objects getSource() method can be used to determine the component that sent the event.
- A JCheckBox consists of a label positioned beside a square; you can click the square to display or remove a check mark. JCheckBox methods include those that set or get the JCheckBox's label, and set or get the JCheckBox's state of checked or unchecked.
- By using the ButtonGroup class, you can group several JCheckBoxes so that the user can select only one at a time. The methods you use with the ButtonGroup class include those that set a JCheckBox in the group to `true` and that return the currently selected JCheckBox.
- A JComboBox Object, created as a drop-down list, displays an option; clicking the option displays a menu that contains other options. When the user selects an item in the menu, the selected item replaces the original item in the display. The JComboBox methods include methods that add an item to the list, methods that select or return an item based on its name or position in the menu of the list, and a method that changes the JComboBox Object to a combo box that allows the user to enter text to select an item.
- When components in a Swing GUI are larger than the area available to display them, you can create a scroll pane container from the JScrollPane class. The JScrollPane class is a container whose methods can be used to hold any component that can be scrolled. It is common to add a JTextArea component to a scroll pane container because it allows you to use both multiple rows and columns.
- A Swing GUI can be designed so that a user can move a toolbar from one section of a graphical user interface to another section. This type of a toolbar is called a dockable toolbar, and the process is called docking. A toolbar is created in Swing with the JToolBar class and can display both text and images in the toolbar menu.

Review Questions

1. The generic name for the component type that holds other components so you can treat a group of several components as a single entity is ________________.
   a. frame
   b. window
   c. container
   d. receptacle
2. To cause a JFrame to close when the Close button is clicked, you can call a JFrame's setDefaultCloseOperation() method with ________________ as an argument:
   a. EXIT_ON_CLOSE
   b. DISPOSE_ON_CLOSE
   c. DO_NOTHING_ON_CLOSE
   d. HIDE_ON_CLOSE
3. A drop-down list can also be configured to be a combo box if the component's ________________ method is called with the argument `true`.
   a. setSize()
   b. setVisible()
   c. setEditable()
   d. setComponent()
4. If you use an argument with a JFrame constructor, the argument represents the JFrame's ________________.
   a. title
   b. size
   c. color
   d. position
5. Within an event-driven program, an object that is interested in an event is a ________________.
   a. source
   b. Component
   c. Container
   d. listener
6. Which of the following statement(s) is true concerning event listeners?
   a. Each listener can handle a specific event type.
   b. A class can implement as many event listeners as it needs.

   c. both a and b
   d. none of the above
7. The statement ________ adds a JCheckBox object aBox to the ButtonGroup object aGroup.
   a. `aGroup.add(aBox);`
   b. `aBox.add(aGroup);`
   c. `add(aBox);`
   d. You cannot add a JCheckBox object to a ButtonGroup object.
8. A JFrame is ________.
   a. a Container
   b. a Component
   c. a Frame
   d. all of the above
9. Component classes include all of the following classes, except ________.
   a. Object
   b. JLabel
   c. ButtonGroup
   d. JCheckBox
10. Component methods include ________.
   a. getName()
   b. setComponent()
   c. isVisible()
   d. deleteComponent()
11. The method that changes the text displayed on a JButton is ________.
   a. setText()
   b. setLabel()
   c. setButton()
   d. setName()
12. Which of the following constant(s) can be used to determine whether a JCheckBox is selected or deselected?
   a. SELECTED
   b. DESELECTED
   c. ItemEvent.SELECTED
   d. both a and b

13. Which of the following statements initializes a JCheckBox with the label “Choose Me”?
    a. `JCheckBox ChooseMe = new JCheckBox();`
    b. `JCheckBox aBox = new JChooseMe();`
    c. `JCheckBox aBox = new JCheckBox("Choose Me");`
    d. `JCheckBox aBox = "Choose Me";`
14. ItemListener is a(n) ________________.
    a. method
    b. Container
    c. Component
    d. interface
15. When you use ItemListener, you must write a method named ________________.
    a. itemMethod()
    b. itemStateChanged()
    c. actionPerformed()
    d. listenerActivated()
16. When you group JCheckBox objects within a ButtonGroup, all other JCheckBoxes are automatically ________________ when the user selects a JCheckBox.
    a. turned off
    b. turned on
    c. disabled
    d. enabled
17. Which of the following statements is true?
    a. You can create a ButtonGroup, and then create the individual JCheckBoxes.
    b. You can create JCheckBoxes, and then create their ButtonGroup.
    c. Both of the above are true.
    d. None of the above are true.
18. The getSelectedIndex() method returns ________________.
    a. void
    b. a Boolean value
    c. an `int`
    d. a ButtonGroup object

19. Clicking a JComboBox object results in ________________.
    a. a check mark appearing on the screen
    b. an item being dimmed
    c. the display of a menu
    d. a JButton becoming disabled
20. To make a JToolBar dockable, the JToolBar component must use a ________________ layout manager.
    a. FlowLayout
    b. BorderLayout
    c. GridLayout
    d. CardLayout

Exercises

1. Write a program that displays a JFrame that contains the words to any well-known nursery rhyme. Save the program as **JNurseryRhyme.java** in the Chapter.13 folder of your Student Disk.
2. Create a Swing application with a JFrame that holds five labels describing reasons a customer might not buy your product (for example, "Too expensive"). Every time the user clicks a JButton, remove one of the negative reasons. Save the program as **JDemoResistance.java** in the Chapter.13 folder of your Student Disk.
3. Write a Swing application for a construction company to handle a customer's order to build a new home. Use separate ButtonGroups to allow the customer to select one of four models (the Aspen, $100,000; the Brittany, $120,000; the Colonial, $180,000; or the Dartmoor, $250,000), the number of bedrooms (two, three, or four; each bedroom adds $10,500), and a garage (no, one-, two-, or three-car; each car adds $7,775). Save the program as **JMyNewHome.java** in the Chapter.13 folder of your Student Disk.
4. a. Write a Swing application for a video store. Place the names of 10 of your favorite movies in a drop-down list. Let the user select the movie that he or she wants to rent. Charge $2.00 for most movies, and $2.50 for your personal favorite movie. Display the total rental fee. Save the program as **JVideo.java** in the Chapter.13 folder on your Student Disk.
   b. Change the drop-down list in the JVideo class to a combo box. Type the name of the movie you wish to rent. Save the program as **JVideo2.java** in the Chapter.13 folder on your Student Disk.
5. Design an order form Swing application for a pizzeria. The user makes a choice from drop-down lists, and the application displays the price. The user can choose a pizza size of small ($7), medium ($9), large ($11), or extra-large ($14), and any number of toppings. There is no additional charge for cheese, but all other toppings

add $1 each to the base price. You must offer a choice of at least five different toppings. Save the program as **JPizzaria.java** in the Chapter.13 folder on your Student Disk.

6. Write a program that allows the user to choose basketball team names that represent the team the user wants to win the NCAA. Put at least five team names in a drop-down list, allow the user to select a team, and then display the selected team. Save the program as **JBasketball.java** in the Chapter.13 folder on your Student Disk.
7. Write a program that allows the user to choose insurance options in JCheckBoxes. Use a ButtonGroup group for HMO (health maintenance organization) and PPO (preferred provider organization) options; the user can only select one option. Use regular JCheckBoxes for dental and vision options; the user can select one option. Save the program as **JInsurance.java** in the Chapter.13 folder on your Student Disk.
8. Write a program that allows the user to select options for a dormitory room. Use JCheckBoxes for the options, such as private room, Internet connection, cable TV connection, microwave, and refrigerator. Display the names of the options checked in a common text area. Save the program as **JDormRoom.java** in the Chapter.13 folder on your Student Disk.
9. Create a Swing applet with a JPanel that holds a JLabel. The JLabel hosts an icon that is larger than the Swing Applet. Create a JScrollPane to hold the JPanel so when the Swing applet is displayed the user can use both horizontal and vertical scrollbars to view the entire picture. If you wish, you can use some appropriate images found in the Chapter.13 folder of your Student Disk. Save the program as **JScrollApplet.java** in the Chapter.13 folder of your Student Disk.
10. Create a JFrame with a JPanel that holds a JLabel. The JLabel hosts an icon that is larger than the JFrame. Create a JScrollPane to hold the JPanel so when the JFrame is displayed you can use both horizontal and vertical scrollbars to view the entire picture. If you wish, you can use some appropriate images found in the Chapter.13 folder of your Student Disk. Save the program as **JScrollPicture.java** in the Chapter.13 folder of your Student Disk.
11. Create a JToolBar with both text and icons on the menu bar. When the user clicks a menu button, display another icon that has the same theme as the menu button in the scroll pane. For example, if you use a patriotic icon on the menu bar you could display a flag in the scroll pane. If you wish, you can use some appropriate images found in theChapter.13 of your Student Disk. Use at least three menu bar items. Save the program as **JImageBar.java** in the Chapter.13 folder of your Student Disk.

12. Each of the following files in the Chapter.13 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugThirteen1.java will become FixDebugThirteen1.java.
    a. DebugThirteen1.java
    b. DebugThirteen2.java
    c. DebugThirteen3.java
    d. DebugThirteen4.java

CASE PROJECT

The WebBuy Company has asked you to write a Swing application that will allow a user to compose three parts of an e-mail message. The three parts of a complete e-mail message include the "To:", "Subject:", and "Message:" text. The "To:" and "Subject:" text areas should each allow for one line. The "Message:" area should allow scrolling, if necessary, to accommodate a long message. A button is required to send the e-mail message. When the message is completed and the Send button is clicked, the program appends "E-mail has been sent!" on a new line in the message area. Save the program as **JEmail.java** in the Chapter.13 folder on your Student Disk.

CHAPTER

14

USING LAYOUT MANAGERS AND THE EVENT MODEL

In this chapter, you will:

- ♦ Learn about layout managers
- ♦ Use JPanels
- ♦ Learn about advanced layout managers
- ♦ Understand events and event handling
- ♦ Use the AWTEvent class methods
- ♦ Use event methods from higher in the inheritance hierarchy
- ♦ Handle mouse events

You have been developing Java applets and applications at Event Handlers Incorporated for several months now. "I love this job!" you exclaim to Lynn Greenbrier one day. "I've learned so much, yet there's so much more I don't know. Sometimes I look at a Swing applet that I've created and think how far I have come; other times I realize I barely have a start in the Java programming language."

"Go on," Lynn urges, "what do you need to know more about right now?"

"Well, I wish it were easier to place components accurately within applets and frames," you say. "I'd like to be able to create more-complex applets and applications, and one thing I'm really confused about is handling events. You've taught me about 'registering objects as listeners,' 'listening,' and 'handling,' but I'd like to learn more about the big picture."

"Event handling is a complicated system," Lynn says. "Let's see if I can help you organize it in your mind. After all, we are the Event Handlers!"

PREVIEWING THE CHAP14 SWING APPLET

Event Handlers Incorporated is developing an applet that the user can manipulate to uncover an advertising slogan. The user passes the mouse over different regions of the applet surface, individually revealing three colored panels. The user can reveal one-third of the advertising slogan at a time by clicking one of the colored panels. The user can also reposition each slogan segment within its panel area by clicking the mouse in a new position. In this chapter, you will learn the techniques used to create this Swing applet. Next you will run the finished version of the Swing applet that is saved on your Student Disk.

To run the Chap14 Swing applet:

1. At the command line for the Chapter.14 folder on your Student Disk, type **`appletviewer TestChap14JPanelApplet.html`**. After the Swing applet appears in the Applet Viewer window, move the mouse pointer inside the boundaries of the Swing applet.
2. When you move the mouse pointer inside the Swing applet boundaries, three colored panels are revealed. When your pointer leaves the Swing applet, the background color of the panels changes to black. When your pointer reenters the Swing applet area, the panels' background colors change back to their original colors.
3. Click any one of the three colored panels. Depending on the exact position of your mouse, you will see all or part of a slogan segment. Click each of the remaining colored panels to reveal all three messages, which are shown in Figure 14-1.

Tip

You can examine the code used to create the applet by opening the Chap14JPanelApplet.java file in your text editor.

4. Close the Applet Viewer window.

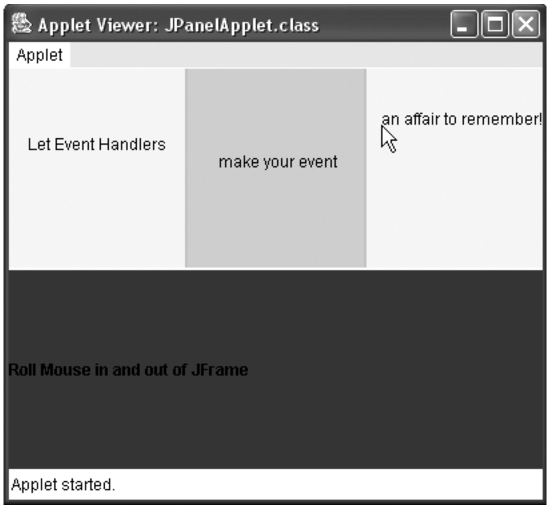


Figure 14-1 Chap14JPanelApplet applet with three messages

Learning About Layout Managers

When you add more than one or two components to a JFrame, Swing applet, or any other container, you can spend a lot of time computing exactly where to place each component so that the layout is attractive and no component obscures another one. Another alternative is to use a layout manager. A **layout manager** is an interface class that is part of the JDK. The layout manager aligns your components so they neither crowd each other nor overlap. For example, one layout manager arranges components in equally spaced columns and rows; another layout manager centers components within their container.

Each layout manager defines methods that arrange components within a container, and each component you place within a Container can also be a container itself, so you can assign layout managers within layout managers. The Java platform supplies layout managers that range from the very simple—FlowLayout and GridLayout, to the special purpose—BorderLayout and CardLayout, to the very flexible—GridBagLayout and BoxLayout. Table 14-1 shows each layout manager and typical situations where they are commonly used.

Table 14-1 Swing layout managers

Swing layout manager	When to use
BorderLayout	Use when you add components to a maximum of five sections arranged in North, South, East, West, and Center positions.
FlowLayout	Use when you need to add components from left to right; FlowLayout automatically moves to the next row when needed.
GridLayout	Use when you need to add components into a grid of rows and columns.
CardLayout	Use when you need to add components that are displayed one at a time.
BoxLayout	Use when you need to add components into a single row or a single column.
GridBagLayout	Use when you need to set size, placement, and alignment constraints for every component that you add.

BorderLayout

The **BorderLayout manager** is the default manager for all content panes. You can use the BorderLayout class with any container that has five or fewer components. However, you should be aware that any of the components could be a container that holds even more components. The components fill the screen in five regions named North, South, East, West, and Center. Figure 14-2 shows five JButton objects filling the five regions in an applet.

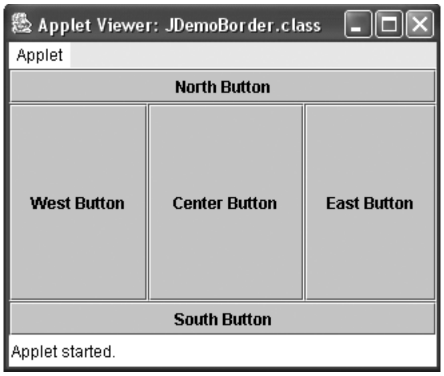


Figure 14-2 Positions using BorderLayout

When you place exactly five components in a container and use BorderLayout, each component fills one entire region, as illustrated in Figure 14-2. When the program runs, the compiler determines the exact size of each component based on the component's contents. When you resize a container that uses BorderLayout, the regions also change in size. If you drag the container's border to make the container wider, then the North, South, and Center regions become wider, but the East and West regions do not change. If you increase the container's height, then the East, West, and Center regions become taller, but the North and South regions do not change.

When you create a container, you can set its layout manager to BorderLayout with the statement `setLayout(new BorderLayout());`. For example, if you create a JFrame with the code `JFrame aFrame = new JFrame();`, then you can set the aFrame's layout manager with `aFrame.setLayout(new BorderLayout());`.

Tip: You may choose to create a BorderLayout manager, but it's not necessary to do so. The program automatically defaults to BorderLayout because it is the default layout manager for all content panes.

When you use the add() method to add a component to a container that uses BorderLayout, you use one of the five area names to specify the region of the container where the component should be placed. For example, when you place a JButton into a container, `add(someButton, "South");` places the someButton object in the South region of the current container, and when you place a JCheckBox into a container, `aNiceFrame.add(someCheckbox, "East");` adds someCheckbox to the East region of aNiceFrame.

When you use BorderLayout with a container, you are not required to add five components. If you add fewer components, any empty component regions disappear and the remaining components expand to fill the available space. Next, so you can observe the effect, you will remove a component from a container that uses BorderLayout.

To create a Container that uses BorderLayout with only four objects:

1. Open a new file in your text editor, and then type the following first few lines of a Swing applet that will demonstrate BorderLayout with only four objects:

```
import javax.swing.*;
import java.awt.*;
public class JDemoBorderNoNorth extends JApplet
{
```

2. Enter the following lines to create four buttons. Note that the North button is purposefully omitted:

```
    private JButton sb = new JButton("South Button");
    private JButton eb = new JButton("East Button");
    private JButton wb = new JButton("West Button");
    private JButton cb = new JButton("Center Button");
```

3. Enter the following code to create the init() method in which you will set the layout manager to BorderLayout:

```
    public void init()
    {
      Container con = getContentPane();
      con.setLayout(new BorderLayout());
```

4. Next enter the following code to add the four buttons to the four regions, along with the closing curly braces for the init() method and class:

```
            con.add(sb,"South");
            con.add(eb,"East");
            con.add(wb,"West");
            con.add(cb,"Center");
        }
    }
```

5. Save the program as **JDemoBorderNoNorth.java**, and then compile it using the **`javac`** command.
6. Open a new text file in your text editor, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JDemoBorderNoNorth.class" WIDTH = 375
   HEIGHT = 300>
</APPLET>
</HTML>
```

7. Save the file as **TestJBorder.html**. Run the Swing applet using the **`appletviewer`** command. The output looks like Figure 14-3. Notice that there are only four components in this Swing applet and that none has been assigned to the North region. The four components simply expand their sizes to fit the space.

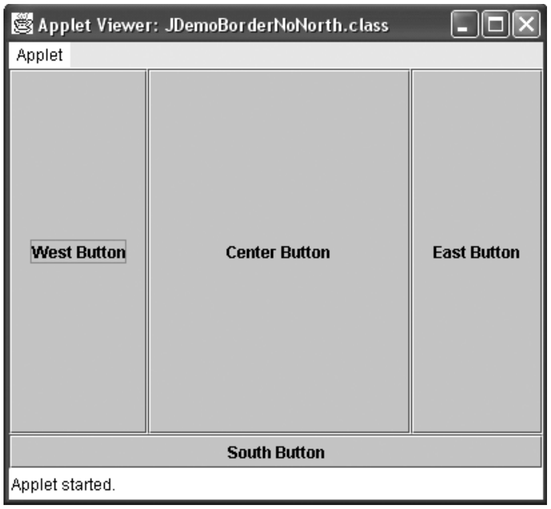


Figure 14-3 JDemoBorderNoNorth applet

8. Close the Applet Viewer window.

9. Open the **JDemoBorderNoNorth.java** file, and then experiment with removing different components. Run the applet by using the **`appletviewer`** command and observe the results.

FlowLayout

Remember from Chapter 9 that you can use the FlowLayout class to arrange components in rows across the width of a container. Each component that you add is placed to the right of previously added components. When you use BorderLayout, the components you add fill their regions. However, when you use FlowLayout, each component retains its default size; for example, a JButton will be large enough to hold its text. When you use BorderLayout, if you resize the window, the components change size accordingly. With FlowLayout, when you resize the window, each component retains its size, but it might become partially obscured or change position.

Next you will modify a BorderLayout Swing applet to demonstrate FlowLayout.

To demonstrate FlowLayout:

1. Open the **JDemoBorderNoNorth.java** file in your text editor, and immediately save it as **JDemoFlowRight.java**. Position the insertion point at the end of the opening curly brace for the JDemoBorderNoNorth.class and press **[Enter]**. Create a new North JButton with the statement **`private JButton nb = new JButton("North Button");`**.
2. Change the class name from JDemoBorderNoNorth to **`JDemoFlowRight`**.
3. Within the init() method, change the setLayout() statement to use FlowLayout and right align:

   ```
   con.setLayout(new FlowLayout(FlowLayout.RIGHT));
   ```

4. Add the North button with the statement **`con.add(nb);`**. Remove the "**South**", "**East**", "**West**", and "**Center**" locations from the remaining statements.
5. Save the file, and then compile it using the **`javac`** command.
6. Open a new text file, and then create the following HTML document to host the Swing applet:

   ```
   <HTML>
   <APPLET CODE = "JDemoFlowRight.class" WIDTH = 300
      HEIGHT = 150>
   </APPLET>
   </HTML>
   ```

7. Save the HTML document as **TestJDemoFlowRight.html** in the Chapter.14 folder on your Student Disk, and then run the applet using the **`appletviewer`** command. Your output should look like Figure 14-4.

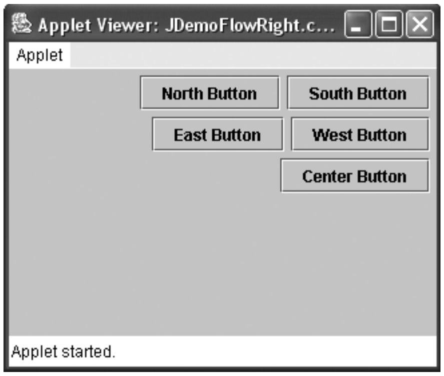


Figure 14-4 JDemoFlowRight Swing applet

8. Experiment with widening and narrowing the Applet Viewer window, and observe how the components realign themselves.
9. Close the Applet Viewer window.

GridLayout

If you want to arrange components into equal rows and columns, you can use the GridLayout class. When you create a GridLayout object, you indicate the numbers of rows and columns you want, and then the container surface is divided into a grid, much like the screen you see when using a spreadsheet program. For example, the statement `setLayout(new GridLayout(4,5));` establishes a GridLayout with four horizontal rows and five vertical columns.

Tip

You specify rows first, and then columns, which is the same technique you used when specifying two-dimensional arrays in Chapter 8.

As you add new Components to a GridLayout, they are positioned left-to-right across each row, in sequence. Unfortunately, you can't skip a position or specify an exact position for a component. You can specify zero for either the row or column figure, which will provide an unlimited number of rows or columns. You can also specify a vertical and horizontal gap measured in pixels using two additional arguments. For example, the statement `setLayout(new GridLayout(4,5,2,6));` establishes a GridLayout with four horizontal rows and five vertical columns, a horizontal gap of two pixels, and a vertical gap of six pixels.

Next you will modify a BorderLayout Swing applet to demonstrate GridLayout.

To demonstrate GridLayout:

1. Open the **JDemoFlowRight.java** file in your text editor, and then save the file as **JDemoGrid.java**.
2. Change the class name from JDemoFlowRight to **`JDemoGrid`**.
3. Within the init() method, type the following statement to change the setLayout() method call to establish a GridLayout with two rows, three columns, a horizontal space of two pixels, and a vertical space of four pixels:

   ```
   con.setLayout(new GridLayout(2,3,2,4));
   ```
4. Save the file, and then compile it using the **`javac`** command.
5. Open the **TestJBorder.html** file in your text editor, change the Swing applet reference to **`JDemoGrid.class`**, and then save the file as **TestJGrid.html** in the Chapter.14 folder on your Student Disk.
6. Use the **`appletviewer`** command to run the applet, and then compare your output to Figure 14-5. The components are arranged in two rows and three columns. Because there are only five components, one grid position still is available.
7. Close the Applet Viewer window.

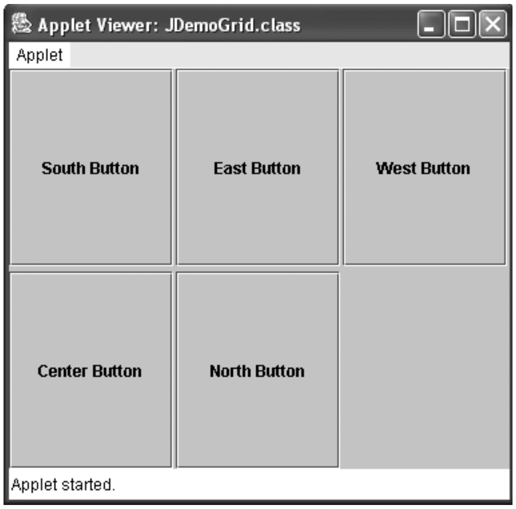


Figure 14-5 JDemoGrid Swing applet

CardLayout

The **card layout manager** generates a stack of containers or components, one on top of another much like a blackjack dealer reveals cards one at a time from the top of a deck of cards. Each container in the group is referred to as a **card**. A card layout is created from the CardLayout class using one of two constructors:

- CardLayout() creates a new card layout without a horizontal or vertical gap.
- CardLayout(int hgap, int vgap) creates a new card layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges. The vertical gaps are placed at the top and bottom edges.

For example, the statement `CardLayout cl = new CardLayout();` creates the card with no horizontal or vertical gaps, while the statement `CardLayout cl = new CardLayout(5,10);` creates the card with a horizontal gap of 5 pixels and a vertical gap of 10 pixels.

After you set the layout manager, as in the statement `con.setLayout(cl);`, you use a slightly different add() method to add to the layout. The method is add(String, container); Container in this example is the Container con. The following statements have five effects:

```
CardLayout cl = new CardLayout();
Container con = this.getContentPane();
con.setLayout(cl);
JButton button1 = new JButton();
con.add("Options", button1);
```

1. Create a CardLayout object named cl.
2. Create a Container object named con for the program.
3. Set the layout of the Container to CardLayout.
4. Create a JButton called button1.
5. Add the Button to the Container con.

The String "Options" represents the name of the card; you can use any name you want. If you have several cards, you might choose to name them "Option1", "Option2", and so on.

Usually in a program that has a card layout manager, a change of cards is triggered by a user's action. For example, a user could select a card by clicking a button that had been registered as an event listener: `option1.addActionListener(this);`. The statement `next(getContentPane())` flips to the next card of the specified container, and the statement `previous(getContentPane())` flips to the previous card of the specified container. For example, the statement `cl.next(getContentPane());` flips to the next card held in the parent content pane.

Next you will create a CardLayout with five cards, each representing a kind of entertainment provided by Event Handlers Incorporated.

To demonstrate CardLayout:

1. Open a new file in your text editor, and then type the following first few lines of a Swing applet that demonstrates CardLayout with five objects:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JCardLayout extends JApplet implements
   ActionListener
{
```

2. Create a new CardLayout manager cl with horizontal and vertical gaps, and declare five buttons as the Components to be displayed in the program:

```
  CardLayout cl = new CardLayout(5,5);
  JButton b1, b2, b3, b4, b5;
```

3. Within the init() method, type the following statement to change the setLayout() method call to establish a CardLayout:

```
  public void init()
  {
   Container con = this.getContentPane();
   con.setLayout(cl);
```

4. Create the JButton b1 with the caption "Clown". Add b1 to the Container with the name "opt1", then register b1 as an event listener. Repeat this process to create four additional buttons for "Singer", "Magician", "Poet", and "Lion Tamer".

```
    b1 = new JButton("Clown");
    con.add("opt1",b1);
    b1.addActionListener(this);
    b2 = new JButton("Singer");
    con.add("opt2",b2);
    b2.addActionListener(this);
    b3 = new JButton("Magician");
    con.add("opt3",b3);
    b3.addActionListener(this);
    b4 = new JButton("Poet");
    con.add("opt4",b4);
    b4.addActionListener(this);
    b5 = new JButton("Lion Tamer");
    con.add("opt5",b5);
    b5.addActionListener(this);
}
```

5. Add the following ActionPerformed method to flip from one card to the next. The cards will display in the order in which they were placed in the content pane.

```
        public void actionPerformed(ActionEvent e)
        {
          cl.next(getContentPane());
        }
      }
   }
```

6. Save the file as **JCardLayout.java**, and then compile it using the `javac` command.
7. Open the **TestJGrid.html** file in your text editor, change the Swing applet reference to `JCardLayout.class`, and then save the file as **TestJCardLayout.html** in the Chapter.14 folder on your Student Disk.
8. Use the `appletviewer` command to run the Swing applet, and then compare your output to Figure 14-6. Note that the first card that appears contains the name of an entertainment act. Click a card and the next card appears with a new name. When you click the last card, the first card appears again.
9. Close the Applet Viewer window.

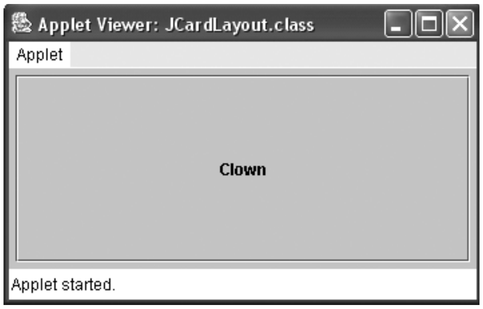


Figure 14-6 JCardLayout Swing applet

Using JPanels

Using the BorderLayout, FlowLayout, GridLayout, and CardLayout managers provides a limited number of screen arrangements. You can increase the number of possible component arrangements by using the JPanel class. A JPanel is similar to a JWindow in that a JPanel is a surface on which you can place components. But a JPanel is *not* a JWindow; it is a sibling of a JComponent, as shown in Figure 14-7. A JPanel is a Container, which means that it can contain other components. For example, you can create a Swing applet using BorderLayout and place a JPanel in any of the five regions. Then within the North JPanel, you can place four JButtons using GridLayout, and within the East JPanel, you can place three JLabels using FlowLayout. By using JPanels within JPanels, you can create an infinite variety of screen layouts.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
```

Figure 14-7 Structure of the JPanel class

When you create a JPanel object, you can use one of two constructors:

- JPanel() creates a new JPanel with a double buffer and a flow layout.
- JPanel(LayoutManager layout) creates a new buffered JPanel with the specified layout manager.

Next you will create a Swing applet that uses a layout manager and contains a JPanel that uses a different layout manager. To begin, you will create one JPanel named wp that holds JButtons indicating the states in which Event Handlers Incorporated does business. Using GridLayout, you will place three JButtons and a JLabel in this JPanel to represent three states. When the user clicks a JButton representing Wyoming, for example, the Swing applet displays the locations of Event Handlers offices in that state. For simplicity, you will activate only one of the three JButtons.

To create the JWesternPanel object:

1. Open a new file in your text editor, and then enter the following first few lines of the JWesternPanel class. The JWesternPanel class extends JApplet and implements ActionListener because the JPanel contains a clickable JButton. Both the JButton and the JLabel are placed at the beginning so that their scope extends throughout the JWesternPanel class.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JWesternPanel extends JApplet
  implements ActionListener
{
  JButton wyButton = null;
  JLabel infoLabel = null;
```

2. Create an init() method for the Swing applet, and enter the following code to create a Container for the Swing applet and set the Container's layout to BorderLayout:

```
    public void init()
    {
      Container container = this.getContentPane();
      container.setLayout(new BorderLayout());
```

3. Create a new JPanel with a GridLayout using two rows and two columns to hold the JLabel and JButtons. Create four Components: three JButtons for the three Western states in which Event Handlers operates, and a JLabel.

```
        JPanel wp = new JPanel(new GridLayout(2,2,2,2));
        wyButton = new JButton("Wyoming");
        JButton coButton = new JButton("Colorado");
        JButton nvButton = new JButton("Nevada");
        infoLabel = new JLabel(" Location Info ");
```

4. Add the Wyoming JButton to the grid layout of the JPanel. Register the JButton as an ActionListener so users can click the Wyoming Button. Add the other JButtons and the JLabel to the grid layout. For now, these components are not clickable, so don't use the addActionListener() method with them.

```
        wyButton = new JButton("Wyoming");
        wp.add(wyButton);
        wyButton.addActionListener(this);
        wp.add(coButton);
        wp.add(nvButton);
        wp.add(infoLabel);
```

5. Create four new JButtons representing the North, South, East, and Center regions of the Swing applet Container. Add these JButtons along with the JPanel, placing the JPanel in the West region.

```
        JButton nb = new JButton("North Button");
        JButton sb = new JButton("South Button");
        JButton eb = new JButton("East Button");
        JButton cb = new JButton("Center Button");
        container.add(nb,"North");
        container.add(sb,"South");
        container.add(eb,"East");
        container.add(wp,"West");
        container.add(cb,"Center");
    }
```

6. When the user clicks the Wyoming Button (which is the only active JButton), the following actionPerformed() method executes. This method displays the name of the Event Handlers Incorporated Wyoming location—Cody. After the method, add a closing curly brace for the class.

```
    public void actionPerformed(ActionEvent e)
    {
      Object source = e.getSource();
      if (source == wyButton)
      {
         infoLabel.setText("Cody");
      }
    }
  }
```

7. Save the file as **JWesternPanel.java** in the Chapter.14 folder on your Student Disk, and then compile the file using the **`javac`** command.
8. Open a new text file, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JWesternPanel.class" WIDTH = 400
   HEIGHT = 300>
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJWesternPanel.html** in the Chapter.14 folder on your Student Disk, and then run the Swing applet using the **`appletviewer`** command. Your output should look like Figure 14-8.

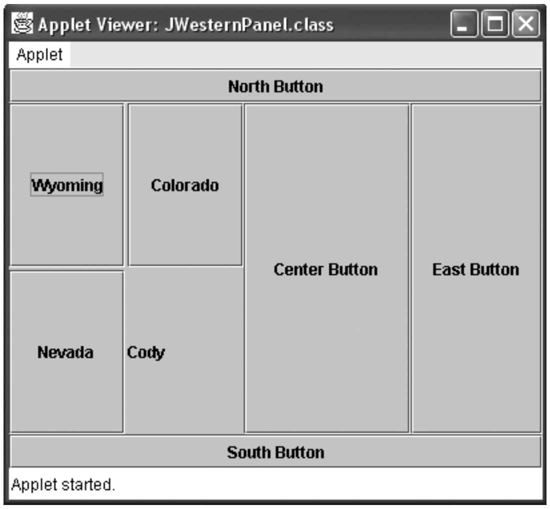


Figure 14-8 JWesternPanel Swing applet

10. Click the **North**, **South**, **Center**, and **East** JButtons. Nothing happens, because you have not activated these JButtons. Similarly, in the West region, click the **Colorado** and **Nevada** JButtons; again, no actions result. Now, click the **Wyoming** JButton; the JLabel within the JWestern Panel displays Cody, the city name of the Wyoming Event Handlers location.
11. Close the Applet Viewer window.

LEARNING ABOUT ADVANCED LAYOUT MANAGERS

Like Components, professional Java programmers are constantly creating new layout managers. You are certain to encounter new and interesting layout managers during your programming career; you might even create your own. Information about more-complicated layout managers is available in the JDK documentation, which you can access at the *java.sun.com* Web site.

When GridLayout is not sophisticated enough for your purposes, you can use GridBagLayout. The GridBagLayout class allows you to add components to precise locations within the grid, as well as to indicate that specific Components should span multiple rows or columns within the grid. For example, if you want to create a JPanel with six JButtons, in which two of the JButtons are twice as wide as the others, you can use GridBagLayout. This class is difficult to use because you must set the position and size for each component.

Another layout manager option is the BoxLayout manager of the BoxLayout class. When you use the BoxLayout manager, components are arranged in either a single row or a single column. When a row or column is filled, additional components do not spill over onto another row or column. Instead, the box layout manager tries to make all the components the same height(row) or width(column). When used in conjunction with a JPanel, a row or column of JCheckBoxes or JButtons can be placed in a BoxLayout and then added to the JPanel.

The BoxLayout constructor requires two arguments: the first refers to the container to which the layout manager applies, and the second is a constant value. The constant value can be either `BoxLayout.X_Axis` for a row arrangement, or `BoxLayout.Y_Axis` for a column arrangement.

If you use a `null` layout manager, as in `setLayout(null);`, then you must use the component class methods setBounds(), setSize(), and setLocation() to position your components. You would do this if you need a complicated layout design that you cannot achieve by using the existing layout managers, or if you want to create a new layout manager.

UNDERSTANDING EVENTS AND EVENT HANDLING

You have already worked with many events in the programs you have written. Beginning in Chapter 9, you learned how to create Swing applets which contain widgets that are controlled by user-initiated events. Now that you understand inheritance and abstract classes, you can take a deeper look at event handling.

Like all Java classes, events are Objects. Specifically, events are Objects that the user initiates, such as key presses and mouse clicks. Many events that occur have significance only for specific components within a program. For example, you have written programs, as well as used programs written by others, in which pressing [Enter] or double-clicking a specific component has no effect. Other events have meaning outside your program; for example, clicking the Close button in the AppletViewer window sends a message to your computer's operating system, which closes the window and stops the program.

The parent class for all event objects is named EventObject, which descends from the Object class. EventObject is the parent of AWTEvent, which in turn is the parent to specific event classes such as ActionEvent and ComponentEvent. Figure 14-9 illustrates the structure of these relationships.

```
java.lang.Object
   |
   +--java.util.EventObject
         |
         +--java.awt.AWTEvent
               |
               +--java.awt.event.ActionEvent
               +--java.awt.event.AdjustmentEvent
               +--java.awt.event.ItemEvent
               +--java.awt.event.TextEvent
               +--java.awt.event.ComponentEvent
                     |
                     +--java.awt.event.ContainerEvent
                     +--java.awt.event.FocusEvent
                     +--java.awt.event.PaintEvent
                     +--java.awt.event.WindowEvent
                     +--java.awt.event.InputEvent
                           |
                           +--java.awt.event.KeyEvent
                           +--java.awt.event.MouseEvent
```

Figure 14-9 Relationships among event classes

The abstract class AWTEvent is contained in the package java.awt.event.

You can see in Figure 14-9 that ComponentEvent is itself a parent to several event classes, including InputEvent, which is parent to KeyEvent and MouseEvent. The family tree for events has roots that go fairly deep, but the class names are straightforward and they share basic roles within your programs. For example, ActionEvents pertain to components that users can click, such as JButtons and JCheckboxes, and TextEvents pertain to components into which the user enters text, such as a JTextField. MouseEvents include determining the location of the mouse and distinguishing between a single- and double-click. Table 14-2 lists some common user actions and the events that are generated from them.

Because ActionEvents involve the mouse, it is easy to confuse ActionEvents and MouseEvents. If you are interested in ActionEvents, you are interested in changes in a component; if you are interested in MouseEvents, your focus is centered on what the user has done manually with the mouse equipment.

Table 14-2 Examples of user actions and their resulting event types

User Action	Resulting Event Type
Click a button	ActionEvent
Click a component	MouseEvent
Click an item in a choice	ItemEvent
Click an item in a check box	ItemEvent
Change text in a text field	TextEvent
Open a window	WindowEvent
Iconify a window	WindowEvent
Press a key	KeyEvent

When you write programs with GUI interfaces, you are always handling events that originate with the mouse or keys on specific Components or Containers. Just as your telephone notifies you when you have a call, the computer's operating system notifies you, the user, when an AWTEvent occurs, for example, when the mouse is clicked. Just as you can ignore your phone when you're not expecting or interested in a call, you can ignore AWTEvents. If you don't care about an event, such as when your program contains a component which, when clicked, produces no effect, you simply don't look for a message to occur.

Tip

There is no class named Event; the general event class is AWTEvent.

When you care about events—that is, when you want to listen for an event—you can implement an appropriate interface for your class. Each event class shown in Table 14-2 has a listener interface associated with it so that for every event class, such as <name>Event, there is a similarly named <name>Listener interface.

Tip

Remember that an interface contains only abstract methods, therefore all interface methods are empty. If you implement a listener, you must provide your own methods for all the methods that are part of the interface. Of course, you may leave the methods empty in your implementation.

Tip

Every <name>Event class has a <name>Listener. The MouseEvent class has an additional listener, the MouseMotionListener.

Every <name>Listener interface method has return type `void`, and each takes one argument—an object that is an instance of the corresponding <name>Event class. Thus, the ActionListener interface has a method named actionPerformed(), and its header is `void actionPerformed(ActionEvent e)`. When an action takes place, the actionPerformed() method executes, and e represents an instance of that event. Interface methods, such as actionPerformed(), that are called automatically when an appropriate event occurs, are called **event handlers**.

Tip

If a listener has only one method, there is no need for an adapter. For example, the ActionListener class has one method, actionPerformed(), so there is no ActionAdapter class.

Whether you use a listener or an adapter, you create an event handler when you write code for the listener methods; that is, you tell your class how to handle the event. After you create the handler, you must also register an instance of the class with the component that you want the event to affect. For any <name>Listener, you must use the form object.add<name>Listener(Component) to register an object with the Component that will listen for objects emanating from it. The add<name>Listener() methods, such as addActionListener() and addItemListener(), all work the same way. They register a listener with a component, return `void`, and take a <name>Listener object as an argument. For example, if a Swing applet is an ActionListener and contains a JButton named pushMe, then within the Swing applet, `pushMe.addActionListener(this);` registers this particular applet as a listener for the pushMe JButton. Table 14-3 lists the events with their listeners and handlers.

Table 14-3 Events, their listeners, and their handlers

Event	Listener	Handlers
ActionEvent	ActionListeneraction	Performed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
TextEvent	TextListener	textValueChanged(TextEvent)
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)

ContainerEvent	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)

ComponentEvent	ComponentListener	componentMoved(ComponentEvent)
		componentHidden(ComponentEvent)
		componentResized(ComponentEvent)
		componentShown(ComponentEvent)

FocusEvent	FocusListener	focusGained(FocusEvent)
		focusLost(FocusEvent)

MouseEvent	MouseListenermouse
	
	MouseMotionListener
		Pressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseEntered(MouseEvent)
		mouseExited(MouseEvent)
		mouseClicked(MouseEvent)
		mouseDragged(MouseEvent)
		mouseMoved(mouseEvent)

Table 14-3 Events, their listeners, and their handlers (continued)

Event	Listener	Handlers

KeyEvent	KeyListener	keyPressed(KeyEvent)
		keyTyped(KeyEvent)
		keyReleased(KeyEvent)

WindowEvent	WindowListener	windowActivated(WindowEvent)
		windowClosing(WindowEvent)
		windowClosed(WindowEvent)
		windowDeiconified(WindowEvent)
		windowIconified(WindowEvent)
		windowOpened(WindowEvent)

Next you will create a class that implements KeyListener. You use the **KeyListener interface** when you are interested in actions the user initiates from the keyboard. The KeyListener interface contains three methods—keyPressed(), keyTyped(), and keyReleased(). For most keyboard applications in which the user must type a keyboard key, it is probably not important whether you take resulting action when a user first presses a key, during the key press, or upon the key's release; most likely these events occur in quick sequence. However, on those occasions when you don't want to take action while the user holds down the key, you can place the actions in the keyReleased() method.

It is best to use the keyTyped() method when you want to discover what character was typed. When the user presses a key that does not generate a character (sometimes called **action keys**), such as a function key, then keyTyped() does not execute. The methods keyPressed() and keyReleased() provide the only ways to get information about keys that don't generate characters.

Tip

Java programmers call keyTyped() events "higher level" events because they do not depend on the platform or keyboard layout. In contrast, keyPressed() and keyReleased() events are "lower level" events and do depend on the platform and keyboard layout.

To create a class that implements KeyListener:

1. Open a new file in your text editor, and then enter the following first few lines for the JKeyFrame class that implements KeyListener:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JKeyFrame extends JFrame
 implements KeyListener
{
```

2. Create a `null` container to hold the JFrame components. Create a BorderLayout manager for the component layout. Create a JLabel and a JTextField.

```
  Container con = null;
  BorderLayout border = new BorderLayout();
  JLabel label = new JLabel("Key Typed:");
  JTextField textField = new JTextField(25);
```

3. In the JKeyFrame constructor method, set the JFrame title to JKeyFrame and the default close operation to EXIT_ON_CLOSE. Create the layout object con by calling the getContentPane() method. Set the layout manager to border layout. Add the JLabel and JTextField to the North and South regions of the border layout. Add an ActionListener for each JButton using the keyword `this` to represent the JFrame.

```
  public JKeyFrame()
  {
   setTitle("JKey Frame");
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   con = this.getContentPane();
   con.setLayout(border);
   con.add(textField,"North");
   con.add(label,"South");
   addKeyListener(this);
   textField.addKeyListener(this);
  }
```

4. The following keyTyped() method is one of the three abstract methods contained in KeyListener. To prove that this method is activated when the user presses a key, send the simple message "pressed" to the command line. Use the getKeyChar() method to retrieve the keyboard character typed, and then display the character with the setText() method.

```
public void keyTyped(KeyEvent e)
 {
  System.out.println("typed");
  char c = e.getKeyChar();
  label.setText ("Key Typed: " + c);
 }
```

5. Similarly, implement the following keyPressed() method to print "pressed" at the command prompt:

```
public void keyPressed(KeyEvent e)
 {
  System.out.println("pressed");
 }
```

6. Implement the following keyReleased() method so it prints "released":

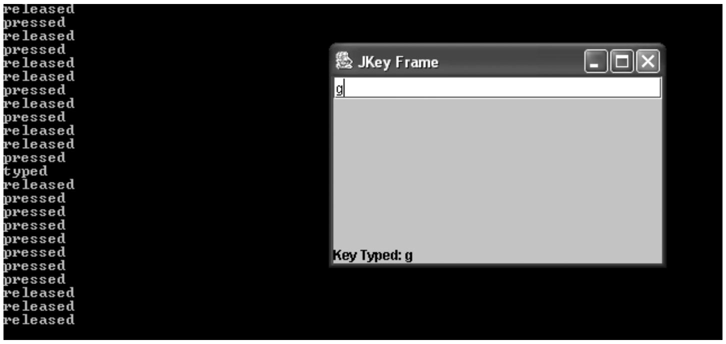
```
public void keyReleased(KeyEvent e)
{
 System.out.println("released");
}
```

7. Add the following main() method that creates a new JFrame named kFrame, sizes it using the setSize() method, and sets its visible property to `true`.

   Remember to add the closing curly brace for the class.

```
  public static void main(String[] arguments)
  {
    JFrame kFrame = new JKeyFrame();
    kFrame.setSize(250,100);
    kFrame.setVisible(true);
  }

 }
```

8. Save the file as **JKeyFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file using the **`javac`** command, and then run the program using the **`java JKeyFrame`** command. When the JFrame appears on your screen, notice that it contains a text field in which you can enter data. When you press an alphabetic keyboard key, the character appears in the text field and in the label, and the command line displays three messages: "pressed", "typed", and "released". Figure 14-10 shows these messages. If you press a key, such as F1 or Alt, that does not generate a character, you see "pressed" and "released", but not "typed". If you hold down a key, such as the Alt key, you can generate several "pressed" messages before receiving the "released" message.



```
released
pressed
released
pressed
released
released
pressed
released
pressed
released
released
pressed
typed
released
pressed
pressed
pressed
pressed
pressed
pressed
pressed
released
released
released
```

Figure 14-10 Output of the JKeyFrame program

Using AWTEvent Class Methods

In addition to the handler methods included with the event listener interfaces, the AWTEvent classes themselves contain methods. You use many of these methods to determine the nature of and the facts about an event in question. For example, the ComponentEvent class contains a getComponent() method that returns the Component involved in the event. You use the getComponent() method when you create an application with several components, and then the getComponent() method allows you to determine which Component is generating the event. The WindowEvent class contains a similar method, getWindow(), that returns the Window that was the source of the event. Table 14-4 lists some useful methods for many of the event classes.

Tip

All Components have the methods addComponentListener(), addFocusListener(), addMouseListener(), and addMouseMotionListener().

Table 14-4 Useful Event class methods

Class	Method	Purpose
EventObject	`Object getSource()`	Returns the Object involved in the event
ComponentEvent	`Component getComponent()`	Returns the Component involved in the event
WindowEvent	`Window getWindow()`	Returns the Window involved in the event
ItemEvent	`Object getItem()`	Returns the Object that was selected or deselected
ItemEvent	`int getStateChange()`	Returns an integer named ItemEvent.SELECTED or ItemEvent.DESELECTED
InputEvent	`int getModifiers()`	Returns an integer to indicate which mouse button was clicked
InputEvent	`int getWhen()`	Returns a time indicating when the event occurred
InputEvent	`boolean isAltDown()`	Returns whether [Alt] was down when the event occurred
InputEvent	`boolean isControlDown()`	Returns whether the Ctrl key was down when the event occurred
InputEvent	`boolean isShiftDown()`	Returns whether the Shift key was down when the event occurred
KeyEvent	`int getKeyChar()`	Returns the Unicode character entered from the keyboard
MouseEvent	`int getClickCount()`	Returns the number of mouse clicks; lets you identify the user's double-clicks
MouseEvent	`int getX()`	Returns the x-coordinate of the mouse pointer
MouseEvent	`int getY()`	Returns the y-coordinate of the mouse pointer
MouseEvent	`Point getPoint()`	Returns the Point Object that contains x- and y-coordinates of the mouse location

You can call any of the methods listed in Table 14-4 by using the object-dot-method format that you use with all class methods. For example, if you have an InputEvent named inEv, and an integer named modInt, then the statement `modInt = inEv.getModifiers();` is valid. You use the getModifiers() method with an InputEvent object, and you can assign the return value to an integer variable. Thus, when you use any of the handler methods from Table 14-3, such as actionPerformed() or itemStateChanged(), they provide you with an appropriate event object. You can use the event object within the handler method to obtain information; you simply add a dot and the desired method name from Table 14-3.

Using Event Methods from Higher in the Inheritance Hierarchy

When you use an event such as KeyEvent, you can use any of the event's methods. Through the power of inheritance, you can also use methods that belong to any class that is a superclass of the event with which you are working. For example, any KeyEvent has access to the InputEvent, ComponentEvent, AWTEvent, EventObject, and Object methods, as well as the KeyEvent methods.

Next you will use an EventObject method with an ActionEvent. You can accomplish this because every ActionEvent is a descendant of EventObject. Therefore, when you create a Component with several JButton objects, you can use ObjectEvent's getSource() method to determine the source of the ActionEvent.

To write a class that uses the EventObject method getSource() with an ActionEvent:

1. Open a new file in your text editor, and then type the following first few lines of the JButtonFrame class that implements the ActionListener interface to respond to JButton clicks:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JButtonFrame extends JFrame
 implements ActionListener
{
```

2. Create the following three JButtons from which the user can choose to change the Frame's background color. Create a `null` container to hold the JFrame components and a container to hold the FlowLayout manager for the component layout.

```
  JButton redButton = new JButton("Red");
  JButton blueButton = new JButton("Blue");
  JButton greenButton = new JButton("Green");
  Container con = null;
  FlowLayout flow = new FlowLayout();
```

3. Begin writing the JButtonFrame constructor and set the layout manager to FlowLayout, the JButtonFrame title to JButtonFrame, and the default close operation to EXIT_ON_CLOSE. Create the layout object con by calling the getContentPane(), and then add the three JButtons to the container.

```
public JButtonFrame()
{
 setTitle("JButton Frame");
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 con = this.getContentPane();
 con.setLayout(flow);
 con.add(redButton);
 con.add(blueButton);
 con.add(greenButton);
```

Tip

Earlier in this chapter, you learned that the default layout for a JFrame is BorderLayout.

4. Continue entering the JButtonFrame constructor by setting the background and foreground of the container. Register the JButtonFrame as an ActionListener for each of the three JButton objects, and then close the constructor method.

```
  con.setBackground(Color.white);
  con.setForeground(Color.black);
  redButton.addActionListener(this);
  blueButton.addActionListener(this);
  greenButton.addActionListener(this);
}
```

5. Add the following main() method that creates a new JButtonFrame named bFrame, sizes it using the setSize() method, and sets its visible property to `true`.

```
public static void main(String[] args)
{
 JButtonFrame bFrame = new JButtonFrame();
 bFrame.setSize(350,250);
 bFrame.setVisible(true);
}
```

6. Because JButtonFrame implements ActionListener, you are required to write code for ActionListener's only method, actionPerformed(). The actionPerformed() method provides you with an ActionEvent object with which you can use the EventObject method named getSource() to return the source of the event as an Object class instance. Using the `if...else` structure allows you to compare the

source Object with possible event sources and take the appropriate action. Remember to add the closing curly braces for the method and class.

```
public void actionPerformed(ActionEvent e)
  {
    Object source = e.getSource();
    if(source == redButton)
      con.setBackground(Color.red);
    else if (source == blueButton)
      con.setBackground(Color.blue);
    else if (source == greenButton)
      con.setBackground(Color.green);
  }
}
```

7. Save the file as **JButtonFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file using the **`javac`** command, and then run the program using the **`java JButtonFrame`** command. The output of the program after the Blue button has been clicked is shown in Figure 14-11. Click any of the three color JButtons and note the change in the JFrame's background color. Note that the JFrame listens for action on each of the JButtons, and the single actionPerformed() method executes no matter which JButton is clicked. You achieve different background colors in the JFrame because you use the ObjectEvent method getSource() with the ActionEvent generated by each button click.

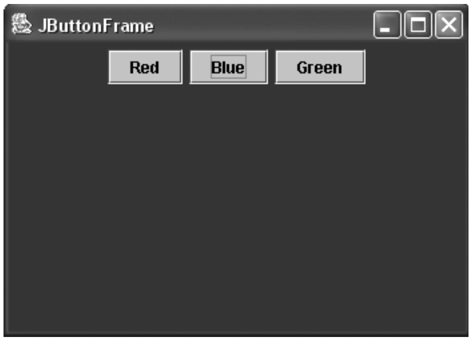


Figure 14-11 Output of the JButtonFrame program after the Blue button has been clicked

Handling Mouse Events

Even though Java program users sometimes type characters from a keyboard, when you write GUI programs, you probably expect users to spend most of their time operating a mouse. The MouseMotionListener interface provides you with methods named MouseDragged( ) and MouseMoved( ) that detect the mouse being rolled or dragged across a component surface. The MouseListener interface provides you with methods named mousePressed(), mouseClicked(), and mouseReleased() that are analogous to the keyboard event methods keyPressed(), keyTyped(), and keyReleased(). With a mouse, however, you are interested in more than its key presses; you sometimes simply want to know where a mouse is pointing. The additional interface methods **mouseEntered()** and **mouseExited()** inform you when the user has positioned the mouse over a component (entered) or moves the mouse off a component (exited). To illustrate, you will create a JMouseFrame class that employs these methods. In addition, you will use three MouseEvent methods—getX() and getY(), which return the mouse coordinates, and getClickCount() which helps you distinguish between single- and double-clicks.

To write the JMouseFrame class:

1. Open a new file in your text editor, and then type the following first few lines of the JMouseFrame class:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMouseFrame extends JFrame
 implements MouseListener
{
```

2. Create the following `null` container and three integer variables. Two hold the x- and y-positions of the mouse; the third holds the size of a circle you will draw when the mouse is clicked.

```
  Container con = null;
  int x, y;
  int size;
```

3. Enter the following JMouseFrame constructor, which sets the title and adds the MouseListener:

```
  public JMouseFrame()
  {
   setTitle("Mouse Frame");
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   con = this.getContentPane();
   addMouseListener(this);
  }
```

4. Enter the following mousePressed() method to get the x- and y-coordinates from the MouseEvent that initiates the mousePressed() method call. The variables x and y hold the exact position of the mouse location at the time of the event. Also, the mousePressed() method repaints the screen.

```
  public void mousePressed(MouseEvent e)
  {
   x = e.getX();
   y = e.getY();
   repaint();
  }
```

Tip

You learned about the repaint() method in Chapter 9.

5. Enter the following code to set the size variable to 10 or 4, depending on whether the user single- or double-clicks the mouse.

```
public void mouseClicked(MouseEvent e)
{
 if (e.getClickCount() == 2)
  size = 10;
 else size = 4;
  repaint();
}
```

6. Enter the following code to change the JFrame background color to yellow when the user positions the mouse over the JFrame, and then change the background to black when the user places the mouse somewhere else on the screen:

```
public void mouseEntered(MouseEvent e)
{
 con.setBackground(Color.yellow);
}
public void mouseExited(MouseEvent e)
{
 con.setBackground(Color.black);
}
```

7. You don't need any special code for the mouseReleased() method, but you must provide the following method because MouseListener is an abstract interface:

```
public void mouseReleased(MouseEvent e)
{
}
```

8. Recall from Chapter 10 that the Graphics method drawOval() requires four arguments. Envision a rectangle surrounding an oval and provide arguments for the x- and y-coordinates of the upper-left corner and the width and height of the rectangle. The mouseClicked() method sets the size to either 4 or 10, depending on the value returned by getClickCount(). To draw a circle with a diameter of either 8 or 20 pixels, use `x - size` and `y - size` for the upper-left corner of the rectangle, and use `size * 2` for the width and the height. Add the closing curly brace for the method:

```
public void paint(Graphics g)
{
 g.drawOval(x - size, y - size, size * 2, size * 2);
}
```

9. Add the following main() method that creates a new JMouseFrame named mFrame, sizes it using the setSize() method, and sets its visible property to `true`. Add the closing curly brace for the class.

```
  public static void main(String[] args)
  {
   JMouseFrame mFrame = new JMouseFrame();
   mFrame.setSize(250,150);
   mFrame.setVisible(true);
  }
}
```

10. Save the file as **JMouseFrame.java** in the Chapter.14 folder on your Student Disk. Compile the file, and then run the program using the **`java JMouseFrame`** command. When the JFrame appears on your screen, roll the mouse over the JFrame surface so it turns yellow. Roll your mouse off the JFrame surface and it turns black. Roll the mouse over the JFrame surface and click; a small circle appears at your mouse position. When you click in a new position, the circle relocates. When you double-click in the same position, a larger circle appears in the new mouse position, as shown in Figure 14-12.

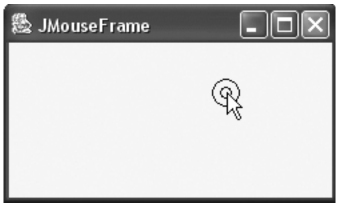


Figure 14-12 Output of the JMouseFrame program when the mouse is clicked and double-clicked in the same position

Chapter Summary

- You can use the BorderLayout class with any container that has five or fewer components. The components fill the screen in five regions named North, South, East, West, and Center. The compiler determines the exact size of each component based on the component's contents. When you use BorderLayout with a container and you add fewer than five components, any empty regions disappear.
- You can use the FlowLayout class to arrange Components in rows across the width of a container. When you use FlowLayout, each component retains its default size and automatically is centered within its assigned container.
- You use the GridLayout class to arrange Components into equal rows and columns. As you add new components to a GridLayout, they are positioned left-to-right across each row in sequence.

- The CardLayout class allows you to arrange Components as if they were stacked like index or playing cards. Only one component is visible at a time.
- A JPanel is similar to a JWindow in that a JPanel is a surface on which you can place components. A JPanel is also a container, which means it can contain other components. By using JPanels within other JPanels, you can create an infinite variety of screen layouts.
- The GridBagLayout class allows you to add components to precise locations within the grid, as well as to indicate that specific components should span multiple rows or columns within the grid. The BoxLayout class allows you to add components in a horizontal row or vertical column.
- Events are Objects that the user initiates, such as key presses and mouse clicks. The parent class for all event objects is named EventObject, which descends from the Object class. EventObject is the parent of AWTEvent, which in turn is the parent to specific event classes such as ActionEvent and ComponentEvent.
- When you want to listen for an event, you can implement an appropriate interface for your class, such as ActionListener or WindowListener. The class becomes an event listener. For every event class named <name>Event, there is a listener that is similarly named—<name>Listener. Every <name>Listener interface method has return type `void`, and each takes one argument—an object that is an instance of the corresponding <name>Event class.
- Interface methods that are automatically called when an appropriate event occurs are called event handlers.
- The KeyListener interface contains three methods: keyPressed(), keyTyped(), and keyReleased().
- The MouseListener interface provides you with methods named mousePressed(), mouseClicked(), and mouseReleased(). The additional interface methods mouseEntered() and mouseExited() inform you when the user positions the mouse over a component (entered) or moves the mouse off a component (exited).

REVIEW QUESTIONS

1. If you add fewer than five components to a BorderLayout ______________.
   a. any empty component regions disappear
   b. the remaining components expand to fill the available space
   c. both a and b
   d. none of the above

2. When you resize a Container that uses BorderLayout, ________________.
   a. the Container and the regions both change in size
   b. the Container changes in size, but the regions retain their original sizes
   c. the Container retains its size, but the regions change or might disappear
   d. nothing happens
3. When you create a JFrame named myFrame, you can set its layout manager to BorderLayout with the statement ________________.
   a. `myFrame.setLayout = new BorderLayout();`
   b. `myFrame.setLayout(new BorderLayout());`
   c. `setLayout(myFrame = new BorderLayout());`
   d. `setLayout(BorderLayout(myFrame));`
4. Which is the correct syntax for adding a JButton named b1 to a Container named con when using CardLayout?
   a. `con.add(b1);`
   b. `con.add("b1");`
   c. `con.add("Options", b1);`
   d. none of the above
5. You can use the ________________ class to arrange components in a single row or column of a container.
   a. FlowLayout
   b. BorderLayout
   c. CardLayout
   d. BoxLayout

6. When you use a ________________, the components you add fill their region; they do not retain their default size.
   a. FlowLayout
   b. BorderLayout
   c. FixedLayout
   d. RegionLayout
7. The statement ________________ ensures that components are placed left-to-right across the Swing applet surface until the first row is full, at which point a second row is started at the applet surface's left edge.
   a. `setLayout(FlowLayout.LEFT);`
   b. `setLayout(new FlowLayout(LEFT));`
   c. `setLayout(new FlowLayout(FlowLayout.LEFT));`
   d. `setLayout(FlowLayout(FlowLayout.LEFT));`

8. The GridBagLayout class allows you to ________________.
   a. add components to precise locations within the grid
   b. indicate that specific components should span multiple rows or columns within the grid.
   c. both a and b
   d. none of the above
9. The statement setLayout(new GridLayout(2,7)); establishes a GridLayout with ________________ horizontal row(s).
   a. zero
   b. one
   c. two
   d. seven
10. As you add new components to a GridLayout, ________________.
    a. they are positioned left-to-right across each row in sequence
    b. you can specify exact positions by skipping some positions
    c. both of the above
    d. none of the above
11. A JPanel is a ________________.
    a. Window
    b. Container
    c. both of the above
    d. none of the above
12. The ________________ class allows you to arrange components as if they are stacked like index or playing cards.
    a. GameLayout
    b. CardLayout
    c. BoxLayout
    d. GridBagLayout
13. AWTEvent is the child class of ________________.
    a. EventObject
    b. Event
    c. ComponentEvent
    d. ItemEvent

14. When a user clicks a JPanel, the action generates a(n) ________________.
    a. ActionEvent
    b. MouseEvent
    c. ButtonEvent
    d. none of the above
15. Event handlers are ________________.
    a. abstract classes
    b. concrete classes
    c. listeners
    d. methods
16. The return type of getComponent() is ________________.
    a. Object
    b. Component
    c. int
    d. void
17. The KeyEvent method getKeyChar() returns a(n) ________________.
    a. int
    b. char
    c. KeyEvent
    d. AWTEvent
18. The MouseEvent method that allows you to identify double-clicks is ________________.
    a. getDouble()
    b. isClickDouble()
    c. getDoubleClick()
    d. getClickCount()
19. You can use the ________________ method to determine the Object where an ActionEvent originates.
    a. getObject()
    b. getEvent()
    c. getOrigin()
    d. getSource()

20. The mousePressed() method is originally defined in the ______________.
    a. MouseListener interface
    b. MouseEvent event
    c. MouseObject object
    d. AWTEvent class

EXERCISES

1. Create a JFrame and set the layout to BorderLayout. Place a JButton containing the name of a politician in each region (left, center, and right, or West, Center, and East). Each politician's physical position should correspond to your opinion of his or her political stance. Save the program as **JPoliticalFrame.java** in the Chapter.14 folder of your Student Disk.
2. Modify the JWesternPanel class you created in this chapter so the user can choose Northern states in addition to Western states. Create the necessary Northern state buttons and activate one of them to display the city location of Event Handlers Incorporated in the Northern region when clicked. Save the program as **JDemoNorth.java** in the Chapter.14 folder of your Student Disk.
3. Use the CardLayout class to write a program that displays a series of cards that make a Royal Flush in hearts (Ace, King, Queen, Jack, and 10). Save the program as **JRoyalFlush.java** in the Chapter.14 folder of your Student Disk.
4. Create 26 JButtons, each labeled with a single, different letter of the alphabet. Create a Swing applet to hold five JPanels in a five-by-one grid. Place six JButtons within the first four JPanels and two JButtons within the fifth JPanel of the Swing applet. Add a JLabel to the fifth JPanel. When the user clicks a JButton, the text of the JLabel is set to "Folder X", where X is the letter of the alphabet that is clicked. Save the program as **JFileCabinet.java** in the Chapter.14 folder of your Student Disk.
5. Create a JFrame that holds four buttons with the names of four different fonts. Draw any String using the font that the user selects. Save the program as **JFontFrame.java** in the chapter.14 folder of your Student Disk.
6. Create a JFrame that uses BorderLayout. Place a JButton in the Center region. Each time the user clicks the JButton, change the background color in one of the regions. Save the program as **JColorFrame.java** in the Chapter.14 folder of your Student Disk.
7. Create a JFrame with JPanels, a JButton, and a JLabel. When the user clicks the JButton, reposition the JLabel to a new location in a different JPanel. Save the program as **JMovingFrame.java** in the Chapter.14 folder of your Student Disk.
8. Write a JFrame application whose appearance and behavior mimics that of the Chap14JPanelApplet Swing applet in this chapter. Save the program as **JFrameApp.java** in the Chapter.14 folder of your Student Disk.

9. Create a class named JPanelOptions that extends JPanel, and whose constructor accepts two colors and a String. Use the colors for background and foreground to display the String. Create a Swing applet named JTeamColors with GridLayout. Display four JPanelOptions JPanels to display the names (in their team colors) of four of your favorite sports teams. Save the program as **JTeamColors.java** in the Chapter.14 folder of your Student Disk.
10. Write a program that lets you determine the integer value returned by the InputEvent method getModifiers() when you press your left, right, or (if you have one) middle mouse button. Save the program as **JLeftOrRight.java** in the Chapter.14 folder of your Student Disk.
11. Write a Swing applet that displays car maintenance services (oil change, tune-up, etc.). Allow the user to select any number of services. If the user clicks the right mouse button, display a message that the user wants service ASAP. Display the choices. Save the program as **JMaintenance.java** in the Chapter.14 folder of your Student Disk.
12. Write a Swing applet that uses a JPanel to show the messages "Mouse Entered" and "Mouse Exited" when the mouse enters and exits the Swing applet. Also, when the mouse is clicked on the Swing applet, a message "Mouse Clicked here" should appear. Save the program as **JMouse.java** in the Chapter.14 folder of your Student Disk.
13. Write a Swing applet to show the messages "Don't Move it! Drag the Mouse!" and "Don't Drag it! Move the Mouse!" when the mouse is alternately rolled and dragged on the Swing applet. Save the program as **JMouseMotion.java** in the Chapter.14 folder of your Student Disk.
14. Each of the following files in the Chapter.14 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugFourteen1.java will become FixDebugFourteen1.java. DebugFourteen2 and DebugFourteen3 are Swing applets. You can use the file TestDebugFourteen.html on your Student Disk to test these Swing applets.
    a. DebugFourteen1.java
    b. DebugFourteen2.java
    c. DebugFourteen3.java
    d. DebugFourteen4.java

Case Project

The Programmers Organization wants you to reprogram one of its Java applications so it uses panels and layout managers. The program JExperience is in the Chapter.14 folder of your Student Disk. Although the program compiles and runs, the appearance of the components in the GUI could be greatly enhanced. Using the skills you learned in this chapter, enhance the layout of the program. Save the program as **JNewExperience.java** in the Chapter.14 folder of your Student Disk.

CHAPTER

15
EXCEPTION HANDLING

In this chapter, you will:

- ♦ Learn about exceptions
- ♦ Try code and catch Exceptions
- ♦ Use the Exception getMessage() method
- ♦ Throw and catch multiple Exceptions
- ♦ Use the `finally` block
- ♦ Understand the limitations of traditional error handling
- ♦ Specify the Exceptions a method can throw
- ♦ Handle Exceptions uniquely with each `catch`
- ♦ Trace Exceptions through the call stack
- ♦ Create your own Exceptions

You're muttering to yourself at your desk at Event Handlers Incorporated.

"Anything wrong?" Lynn Greenbrier asks as she passes by.

"It's these errors!" you complain.

"Aren't you going overboard?" Lynn asks. "Everyone makes errors when they code programs."

"Oh, I expect typos and compiler errors while I'm developing my programs," you say, "but no matter how well I write my code, the user can still mess everything up by inputting bad data. The Event Planning Department told me that it has events planned for the 32nd day of the month and for negative five people. Even if my code is perfect, the user can enter mistakes."

"Then your code isn't perfect yet," Lynn says. "Besides writing programs that can handle ordinary situations, you must enable your programs to handle exceptions."

Learning About Exceptions

An **exception** is an unexpected or error condition. The programs you write can generate many types of potential exceptions, such as when:

- You issue a command to read a file from a disk, but the file does not exist there.
- You write data to a disk, but the disk is full or unformatted.
- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide a value by zero, access an array with a subscript that is too large, or calculate a value that is too large for the answer's variable type.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional". The object-oriented techniques to manage such errors comprise the group of methods known as **exception handling**.

Tip

Providing for exceptions involves an oxymoron; you must expect the unexpected.

Like all other classes in the Java programming language, exceptions are Objects; their class name is Exception. In Java, there are two basic classes of errors: Error and Exception. Both of these classes descend from the Throwable class, as shown in Figure 15-1.

```
Throwable
|
+--Exception
|   |
|   +--IOException
|   |
|   +--RuntimeException
|   |   |
|   |   +--ArithmeticException
|   |   |
|   |   +--ArrayIndexOutOfBoundsException
|   |   |
|   |   +--Others..
|   |
|   +--Others..
|
+--ErrorException
    |
    +--OutOfMemoryException
    |
    +--InternalErrorException
    |
    +--Others..
```

Figure 15-1 Structure of the Exception class

Tip

With JDK version 1.4.0, Java acknowledges over 40 categories of Exceptions with unusual names such as ActivationException, AlreadyBoundException, AWTException, CloneNotSupportedException, PropertyVetoException, and UnsupportedFlavorException. See the *http://java.sun.com* Web site for more details about these and other Exceptions.

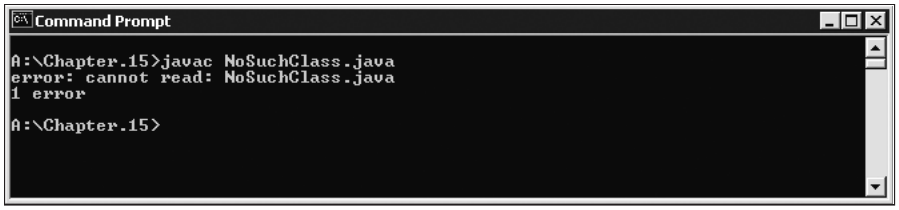
The **Error class** represents more serious errors from which your program usually cannot recover. These errors are the ones you probably have made in your own programs when you spelled a class name incorrectly or stored a required class in the wrong folder. When a program cannot locate a required class or your system runs out of memory, an Error condition occurs. Of course, a person can recover from such errors by spelling a name correctly, moving a file to the correct folder, or by physically installing more memory, but a program cannot recover from these kinds of mistakes.

The **Exception class** comprises less serious errors that represent unusual conditions that arise while a program is running, and from which the program *can* recover. Some examples of Exception class errors include using an invalid array subscript or performing certain illegal arithmetic operations.

When your code causes a program error, whether inadvertently or purposely, you can determine whether the type of Throwable object generated is an Error or an Exception by examining the message that you receive from Java after the error occurs. Next you will generate an unrecoverable Error.

To purposefully cause an unrecoverable Error:

1. Go to the command prompt for Chapter 15.
2. Type **`javac NoSuchClass.java`**, and then press **[Enter]**. Unless you have created a file named NoSuchClass.java in the current directory, the error message you receive should look like Figure 15-2.



```
A:\Chapter.15>javac NoSuchClass.java
error: cannot read: NoSuchClass.java
1 error

A:\Chapter.15>
```

Figure 15-2 Error message generated by missing file

Tip: Even though you generated an Error with an uppercase E (that is, you generated an instance of the Error class), the error message displays with a lowercase e.

The "cannot read" Error in Figure 15-2 must be remedied by typing a different class name, or storing a file with the name NoSuchClass.java in the Chapter.15 folder. In other words, a person must take action before the command can successfully execute; there is no program code you could write that would prevent the Error message. However, when you generate a recoverable Exception, which is less severe than an Error, you see a different type of message. An Exception message indicates that you could have prevented the message by using specific code within your program. Next you will generate a recoverable Exception.

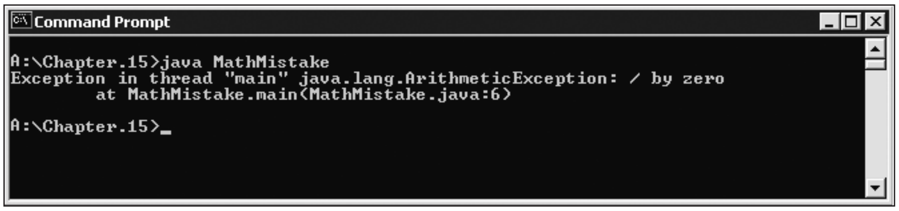
To purposefully cause an Exception:

1. Open a new file in your text editor, and then type the following MathMistake class that attempts to divide by zero:

```
public class MathMistake
{
       public static void main(String[] args)
       {
              int num = 13, denom = 0, result;
              result = num / denom;
       }
}
```

Tip: You never should write a program that purposefully divides a value by zero. However, this situation certainly could occur if a variable used as a denominator gets its value as the result of user input.

2. Save the file as **MathMistake.java** in the Chapter.15 folder on your Student Disk, and then compile using the `javac MathMistake.java` command.
3. After the program compiles successfully, run the program using the `java MathMistake` command. Your result should look like Figure 15-3. You can see that the Exception is a java.lang. ArithmeticException, which is a subclass of Exception. You also get some information about the error ("/ by zero"), the method that generated the error (MathMistake.main), and the file and line number for the error (MathMistake.java, line 6); your line number might be different if you include comment lines at the beginning of your program.



```
A:\Chapter.15>java MathMistake
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MathMistake.main(MathMistake.java:6)

A:\Chapter.15>_
```

Figure 15-3 Exception generated by the MathMistake class

Just because an Exception occurs, you don't necessarily have to deal with it. In the MathMistake class, you simply let the offending program terminate. However, the program termination is abrupt and unforgiving. When a program divides two numbers (or even performs a less trivial task such as playing a game with the user or balancing a checkbook), the user might get annoyed if the program ends abruptly. However, if the program is used for air-traffic control or to monitor a patient's vital statistics during surgery, an abrupt conclusion could be disastrous. Object-oriented error-handling techniques provide more elegant (and safer) solutions.

Programmers had to deal with error conditions long before object-oriented methods were conceived. Probably the most-often-used error-handling solution has been to terminate the offending program. For example, you can change the main() method of the MathMistake class to halt the program before dividing by zero, as follows:

```
public class MathMistake
{
        public static void main(String[] args)
        {
               int num = 13, denom = 0, result;
               if(denom == 0)
                      System.exit(1);
                      result = num / denom;
        }
}
```

Tip: You first used the System.exit() method in Chapter 5 when you wrote code to close a dialog box. At that time, you used 0 as the argument to System.exit() to indicate the program was ending normally. Here, you use 1, which conventionally indicates a problem or error situation.

When you use the System.exit() method, the current application ends and control returns to the operating system. The convention is to return 1 if an error is causing program termination, or 0 if the program is ending normally. Using this exit() method circumvents displaying the error message because the program ends before the error occurs.

Exception handling provides a more elegant solution for handling error conditions. In object-oriented terminology, you "try" a procedure that might cause an error. A method that detects an error condition or Exception "throws an Exception", and the block of code that processes the error "catches the Exception".

Trying Code and Catching Exceptions

When you create a segment of code in which something might go wrong, you place the code in a **`try`** block, which is a block of code you attempt to execute, while acknowledging that an Exception might occur. A `try` block consists of the following elements:

- The keyword `try`
- An opening curly brace
- Statements that might cause Exceptions
- A closing curly brace

You must code at least one `catch` block immediately following a `try` block. A **`catch` block** is a segment of code that can handle an Exception that might be thrown by the `try` block that precedes it. Each `catch` block can "catch" one type of Exception. You create a `catch` block by typing the following elements:

- The keyword `catch`
- An opening parenthesis
- An Exception type
- A name for an instance of the Exception type
- A closing parenthesis
- An opening curly brace
- Statements that take the action you want to use to handle the error condition
- A closing curly brace

If a method `throws` an Exception that will be caught, you must also use the keyword `throws`, followed by an Exception type in the method header. Figure 15-4 shows the general format of a `try...catch` pair.

```
public class someMethod throws someException
{
   try
   {
         //Statements that might cause an Exception
   }
   catch(someExceptionan ExceptionInstance)
   {
         //What to do about it
   }
//Statements here execute even if there was no Exception
}
```

Figure 15-4 General format of a `try...catch` pair

Tip: A `catch` block looks a lot like a catch() method that takes an argument that is some type of Exception. However, it is not a method; it has no return type, and you can't call it directly.

Tip: Some programmers refer to a `catch` block as a `catch` clause.

In Figure 15-4, someException represents the Exception class or any of its subclasses. If an Exception occurs during the execution of the `try` block, then the statements in the `catch` block will execute. If no Exception occurs within the `try` block, then the `catch` block will not execute. Either way, the statements following the `catch` block execute normally. Next you will alter the MathMistake class so it catches the division-by-zero Exception.

To catch an ArithmeticException:

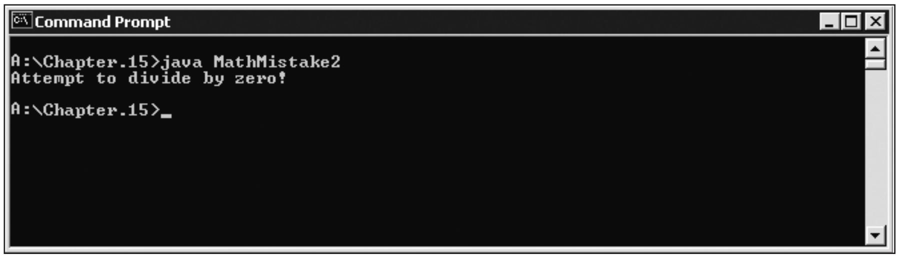
1. Use your text editor's Save As command to save the MathMistake.java file as **MathMistake2.java**, and then change the class name in the class header from MathMistake to **`MathMistake2`**.
2. Position your insertion point at the end of the main() method header (`public static void main(String[] args)`), press the **Spacebar**, and then type **`throws ArithmeticException`**.
3. Position your insertion point at the end of the line that declares the three integer variables, and then press **[Enter]** to start a new line.
4. Type **`try`**, press **[Enter]**, and then type an opening curly brace.
5. Position your insertion point at the end of the line that performs division (`result = num / denom;`), press **[Enter]**, and then type a closing curly brace for the `try` block.
6. Press **[Enter]**, and then type a `catch` block that sends a message to the command line, as follows:

```
catch(ArithmeticException error)
{
      System.out.println("Attempt to divide by zero!");
}
```

Tip: If you want to send error messages to a different location from "normal" output, you can use System.err instead of System.out. For example, if a program writes a report to a specific disk file, you might want errors to write to a different disk file or to the screen.

15

7. Save the file, compile it, and then execute the program. The output looks like Figure 15-5, which shows that the Exception was caught successfully and your error message printed.



```
A:\Chapter.15>java MathMistake2
Attempt to divide by zero!

A:\Chapter.15>_
```

Figure 15-5 Output of MathMistake2

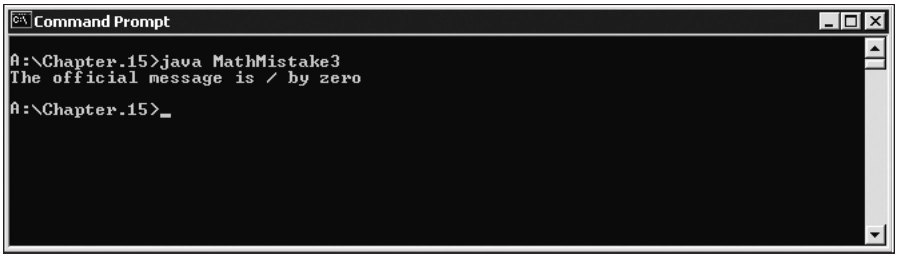
USING THE EXCEPTION GETMESSAGE() METHOD

When the MathMistake2 program prints the error message ("Attempt to divide by zero!"), you cannot confirm that division by zero was the source of the error. In reality, *any* ArithmeticException generated within the `try` block in the program would be caught by the `catch` block in the method. Instead of writing your own message, you can use the getMessage() method that ArithmeticException inherits from the Throwable class. To retrieve Java's message about any Throwable Exception named someException, you code `someException.getMessage()`. In the next steps, you will use the getMessage() method instead of creating your own.

To use the getMessage() method with the MathMistake class:

1. Use your text editor's Save As command to save the MathMistake2.java file as **MathMistake3.java**.
2. Change the class header to **`public class MathMistake3`**.
3. Within the `catch` block, remove the existing println() statement and replace it with the following:

   ```
   System.out.println("The official message is " + error.get
   Message());
   ```
4. Save the file, and then compile and run it. Figure 15-6 shows the output. Java's analysis of the situation prints instead of your own.



```
A:\Chapter.15>java MathMistake3
The official message is / by zero

A:\Chapter.15>_
```

Figure 15-6 Output of MathMistake3

Of course, you might want to do more in a `catch` block than print an error message; after all, Java did that for you without catching any Exceptions. You also might want to add code to correct the error; such code would force the arithmetic to divide by one rather than by zero. Next you will add code to the `catch` block to catch the exception.

To add corrective code to the `catch` block of the MathMistake class:

1. Save the MathMistake3 file as **MathMistake4.java**.
2. Change the class header to reflect the new class name.
3. Position your insertion point at the end of the println() statement within the `catch` block of the main() method of the class, and then press **[Enter]** to

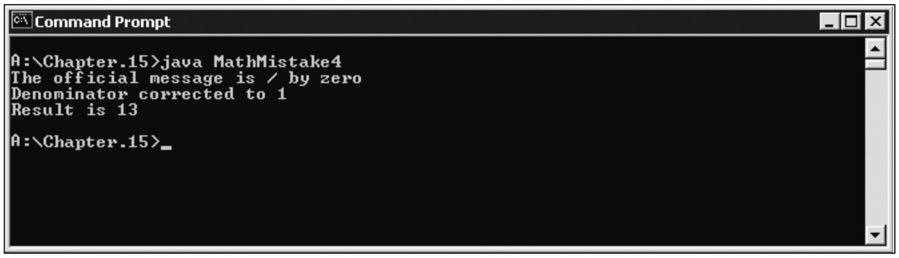
start a new line. Type the following message to indicate the action you are taking: **`System.out.println("Denominator corrected to 1");`**. Follow this statement with a recalculation of the result variable by typing the following code on a new line: **`result = num / 1;`**.

Tip: You can achieve the same result by coding `result = num;` instead of `result = num / 1;`. Explicitly dividing by one simply makes the code's intention clearer.

4. Position your insertion point after the closing curly brace of the `catch` block, and then press **[Enter]** to start a new line. Add the following println() statement to show the result:

   ```
   System.out.println("Result is " + result);
   ```

5. Save the program, and then compile and execute it. Figure 15-7 shows the result, which confirms that the `catch` block provides you with a usable result.



```
A:\Chapter.15>java MathMistake4
The official message is / by zero
Denominator corrected to 1
Result is 13

A:\Chapter.15>_
```

Figure 15-7 Output of MathMistake4

THROWING AND CATCHING MULTIPLE EXCEPTIONS

You can place as many statements as you need within a `try` block, and you can `catch` as many Exceptions as you want. If you `try` more than one statement, only the first error-generating statement `throws` an Exception. As soon as the Exception occurs, the logic transfers to the `catch` block, which leaves the rest of the statements in the `try` block unexecuted.

When a program contains multiple `catch` blocks, they are examined in sequence until a match is found for the type of Exception that occurred. Then the matching `catch` block executes and each remaining `catch` block is bypassed.

For example, consider the program in Figure 15-8. The main() method in the TwoMistakes class `throws` two types of Exceptions: ArithmeticExceptions and IndexOutOfBoundsExceptions. (An IndexOutOfBoundsException occurs when an array subscript is not within the allowed range.)

```
public class TwoMistakes
{
   public static void main(String[]args)
         throws ArithmeticException, IndexOutOfBoundsException
   {
         int num= 13,denom= 0,result;
         int[] array= {22,33,44};
         try
         {
               result= num/denom;//First try
               result= array[num];//Second try
         }
         catch(ArithmeticExceptionerror)
         {
               Systemoutprintln("Arithmetic error");
         }
         catch(IndexOutOfBoundsException error)
         {
               System.out.println("Index error");
         }
   }

}
```

Figure 15-8 TwoMistakes class

The TwoMistakes class declares three integers and an integer array with three elements. In the main() method, the `try` block executes, and at the first statement within the `try` block, an Exception occurs because the denom in the division problem is zero. The `try` block is abandoned, and the logic transfers to the first `catch` block. Division by zero causes an ArithmeticException, and because the first `catch` block receives an ArithmeticException, the message "Arithmetic error" prints. In this example, the second `try` statement is never attempted, and the second `catch` block is skipped.

If you make one minor change to the class in Figure 15-8, the process changes. You can force the division in the `try` block to succeed by substituting a constant value for the denom variable or by reversing the positions of num and denom in the line containing the `// First try` comment. With either of these changes, division by zero does not take place. The line containing `// First try` succeeds, and the program proceeds to the `// Second try` statement. This statement attempts to access element 13 of a three-element array, so it `throws` an IndexOutOfBoundsException. The `try` block is abandoned, and the first `catch` block is examined and found unsuitable because it does not catch an IndexOutOfBoundsException. The program logic proceeds to the second catch block whose Exception argument type is a match for the thrown Exception, so the message "Index error" prints.

Sometimes you want to execute the same code no matter which Exception type occurs. For example, within the TwoMistakes program in Figure 15-8, each of the two catch blocks prints a unique message. Instead, you might want both the ArithmeticException's **`catch`** block and the IndexOutOfBoundsException's **`catch`** block to use the getMessage() method. Because ArithmeticExceptions and IndexOutOfBoundsExceptions are both subclasses of Exception, you can rewrite the TwoMistakes class as shown in Figure 15-9, using a single generic **`catch`** block that can **`catch`** any type of Exception.

```
public class TwoMistakes
{
   public static void main(String[]args) throws
           ArithmeticException, IndexOutOfBoundsException
   {
         int num= 13,denom= 0,result;
         int[] array= {22,33,44};
         try
         {
               result= num/denom;//First try
               result= array[num];//Second try
         }
         catch(Exceptionerror)
         {
               System.out.println("Error is"+ error.getMessage());
         }
   }

}
```

Figure 15-9 TwoMistakes class using a single, generic `catch` block

The **`catch`** block in Figure 15-9 accepts a more generic Exception argument type than that thrown by either of the potentially error-causing **`try`** statements, so the generic **`catch`** block can act as a "catch-all" block. When either an arithmetic or array error occurs, the thrown error is "promoted" to an Exception error in the **`catch`** block. Through inheritance, ArithmeticExceptions and IndexOutOfBoundsExceptions are Exceptions, and an Exception is Throwable, so you can use the Throwable class getMessage() method.

When you list multiple **`catch`** blocks following a **`try`** block, you must be careful that some **`catch`** blocks don't become unreachable. For example, if successive **`catch`** blocks catch an IndexOutOfBoundsException and an ordinary Exception, then IndexOutOfBoundsException errors will cause the first **`catch`** to execute and other Exceptions will "fall through" to the more general Exception **`catch`** block. However, if you reverse the sequence of the **`catch`** blocks, then even IndexOutOfBoundsExceptions will be caught by the Exception **`catch`**. The IndexOutOfBoundsException **`catch`** block is unreachable because the Exception **`catch`** block is in its way and the class will not compile.

USING THE `finally` BLOCK

When you have actions you must perform at the end of a `try...catch` sequence, you can use a **`finally` block**. The code within a `finally` block executes whether or not the preceding `try` block identifies Exceptions. Usually, you use a `finally` block to perform clean-up tasks that must happen whether or not any Exceptions occurred, and whether or not any Exceptions that occurred were caught. Figure 15-10 shows the format of a `try…catch` sequence that uses a `finally` block.

```
public class someMethod throws someException
{
   try
   {
         //Statements that might cause an Exception
   }
   catch(someExceptionan ExceptionInstance)
   {
         //What to do about it
   }
   finally
   {
         //Statements here always execute
   }

}
```

Figure 15-10 General form of a `try...catch` block with a `finally` block

Compare Figure 15-10 to Figure 15-4 shown earlier in this chapter. With the program in Figure 15-4, when the `try` code works without error, control passes to the statements at the end of the method. Also, when the `try` code fails and `throws` an Exception, if the Exception is caught, then the `catch` block executes, and again, control passes to the statements at the end of the method. At first glance, it seems as though the statements at the end of the method always execute. However, the last set of statements might never execute for at least two reasons:

- It is possible that an unplanned Exception will occur.
- The `try` or `catch` block might contain a `System.exit();` statement.

Any `try` block might throw an Exception for which you did not provide a `catch` block. After all, Exceptions occur all the time without your handling them, as one did in the first MathMistake program in this chapter. In the case of an unhandled Exception, program execution stops immediately, the Exception is sent to the operating system for handling, and the current method is abandoned. Likewise, if the `try` block contains an exit() statement, execution stops immediately.

When you include a `finally` block, you are assured that the `finally` statements will execute before the method is abandoned, even if the method concludes prematurely. For example, programmers often use a `finally` block when the program uses data files that must be closed. You will learn more about writing to and reading from data files in Chapter 16. For now, however, consider the pseudocode that represents part of the logic for a typical file-handling program:

```
try
{
       Open the file
       Read the file
       Place the file data in an array
       Calculate an average from the data
       Display the average
}
catch(IOException e)
{
       Issue an error message
}
finally
{
       If the file is open, close it
}
```

The preceding pseudocode represents a program that opens a file; if the file does not exist or is empty, an input/output exception, or IOException, is thrown and the `catch` block handles the error. However, because the program uses an array, it is possible that even though the file opened successfully, an uncaught ArrayIndexOutOfBoundsException might occur. In such an event, you want to close the file before proceeding. By using the `finally` block, you ensure that the file is closed because the code in the `finally` block executes before control returns to the operating system. The code in the `finally` block executes no matter which of the following outcomes of the `try` block occurs:

- The `try` ends normally.
- The `catch` executes.
- An Exception causes the method to abandon prematurely—perhaps the array is not large enough to hold the data, or calculating the average results in division by zero. These Exceptions would not allow the `try` block to finish, nor would they cause the `catch` block to execute.

Tip

If a `try` block calls the System.exit() method, and the `finally` block calls the same method, it is the exit() method in the `finally` block that will actually execute. The `try` block's exit() method call will be abandoned.

Understanding the Limitations of Traditional Error Handling

Before the conception of object-oriented programming languages, potential program errors were handled using somewhat confusing, error-prone methods. For example, a traditional, non-object-oriented, procedural program might perform three methods that depend on each other using code that provides error checking similar to the pseudocode in Figure 15-11.

```
call methodA
if methodA worked
{
   call methodB
   if methodB worked
   {
      call methodC
      if methodC worked
         everything's okay so print finalResult
      else
         set errorCode to 'C'
   }
   else
      set errorCode to 'B'
}
else set errorCode to 'A'
```

Figure 15-11 Pseudocode representing traditional error checking

The program in Figure 15-11 performs methodA; it then performs methodB only if methodA is successful. Similarly, methodC executes only when methodA and methodB are successful. When any method fails, the program sets an appropriate errorCode to A, B, or C. (Presumably, the errorCode is used later in the program.) The program is difficult to follow, and the purpose of the program (and its presumed outcome when there are no errors)—to print the finalResult—is lost in the maze of `if` statements. Also, you can easily make coding mistakes within such a program because of the complicated nesting, indenting, and opening and closing of curly braces.

Compare the same program logic using Java's object-oriented error-handling technique shown in Figure 15-12. Using the `try…catch` object-oriented technique provides the same results as the traditional method, but the real statements of the program (calling methods A, B, and C, and printing finalResult) are placed together where their logic is easy to follow. The `try` steps should usually work without generating errors; after all, the errors are "exceptions". It is convenient to see these business-as-usual steps in one location. The unusual, exceptional events are grouped and moved out of the way of the primary action.

```
try
{
   methodA(and maybe throw anException)
   methodB(and maybe throw anException)
   methodC(and maybe throw anException)
   everything's okay so print finalResult
}
catch(methodA's error)
{
   set errorCode to 'A'
}
catch(methodB's error)
{
   set errorCode to 'B'
}
catch(methodC's error)
{
   set errorCode to 'C'
}
```

Figure 15-12 Pseudocode representing object-oriented Exception handling

Specifying the Exceptions a Method Can Throw

When you write a method that might throw an Exception, you can type the clause `throws <name>Exception` after the method header to indicate the type of Exception that might be thrown. Every Java method you write has the potential to throw an Exception. Some Exceptions, such as an InternalErrorException, can occur anywhere, at any time. However, for most Java methods that you write, you do not use a `throws` clause. For example, you have used a `throws` clause only a few times in the many programs you have written while working through this book. Most of the time, you let Java handle any Exception by shutting down the program. Imagine how unwieldy your programs would become if you were required to provide for every possible error, including equipment failures and memory problems. Most exceptions never have to be explicitly thrown or caught.

Tip

You never have to throw Error or RuntimeException exceptions explicitly. Most of the errors you received when you made mistakes in your Java programs are RuntimeExceptions—unplanned exceptions that occur during a program's execution.

One exception to the rule of not throwing Exceptions involves the IOException. You learned in Chapter 5 that you must include a `throws` clause in the method header of programs that allow keyboard input. In Chapter 16 you will discover that you also must include a `throws` clause in programs that use file input. However, even when you are not required to handle an Exception, you might choose to do so. When your method will throw an Exception that you want to handle, you must include the `throws` clause in the method header.

Tip InterruptedException is another example of a `throws` clause that Java requires that you use when you are working with threads. You will learn about threads in Chapter 17.

When a method you write `throws` an Exception, the method can `catch` the Exception, although it is not required to do so. There are many times when you won't want a method to handle its own Exception. With many methods, you want the method to check for errors, but you do not want to require a method to handle an error if it finds one. The calling program might need to handle the error differently, depending on its purpose. For example, one program that divides values might need to terminate if division by zero occurs. A different program simply might want the user to reenter the data to be used. The method that contains the division statement can throw the error, and the calling program can assume the responsibility for handling the error detected by the method.

Tip You know a method can throw without catching because you have written methods that use keyboard input. With those methods, you threw an Exception, but you did not provide a `catch` block.

Java requires that you use the `throws` clause in the header of a method that might `throw` an Exception so that programs that use your methods are notified of the potential for an Exception. When you use any method, to be able to use the method to its full potential, you must know the method's name and three additional pieces of information:

- The method's return type
- The type and number of arguments the method requires
- The type and number of Exceptions the method `throws`

To use a method, you must first know what types of arguments the method that you send it requires. You can call a method without knowing its return type, but if you do so, you can't benefit from any value that the method returns. (Also, if you use a method without knowing its return type, you probably don't understand the purpose of the method.)Likewise, you can't make sound decisions about what to do in case of an error if you don't know what types of Exceptions a method might throw. A method's header, including its name, any arguments, and any `throws` clause, is called the **method's signature**.

Next you will create a class that contains two methods that throw Exceptions but don't `catch` them. The PickMenu class allows Event Handlers Incorporated customers to choose a dinner menu selection as part of their event-planning process. Before you create PickMenu, you will create the Menu class that lists dinner choices for customers and allows them to make a selection.

To create the Menu class:

1. Open a new file in your text editor, and then enter the following class header and the opening curly brace for the Menu class:

   ```
   public class Menu
   {
   ```

2. Type the following String array for three entrée choices:

   ```
   String[] entreeChoice = {"Rosemary Chicken",
      "Beef Wellington", "Maine Lobster"};
   ```

3. Add the displayMenu() method, which lists each entrée option with a corresponding number the customer can type to make a selection. Even though the allowable entreeChoice array subscripts are 0, 1, and 2, most users would expect to type 1, 2, or 3. So, you code `x + 1` rather than `x` in the println() prompt.

   ```
   public void displayMenu()
   {
          System.out.println
            ("Type your selection, then
          press [Enter].");
          for(int x = 0; x < entreeChoice.length; ++x)
                  System.out.println("Type " + (x + 1) +
                     " for " + entreeChoice[x]);
   }
   ```

4. Create the following getSelection() method, which requires an integer argument and returns the name of one of the selected menu items. Because the user enters a value that is one higher than the actual subscript, you need to subtract one from `x` when accessing the array. Finally, include the closing curly brace for the Menu class.

   ```
      public String getSelection (int x)
      {
          return(entreeChoice[x - 1]);
      }
      }
   ```

5. Save the file as **Menu.java** in the Chapter.15 folder on your Student Disk.

6. Compile the class using the **`javac`** command.

Next you can create the PickMenu class, which lets the customer choose from the available dinner entrée options. The PickMenu class declares a Menu, an integer that holds the user's numeric menu choice, and a String named guestChoice that holds the name of the entrée that the customer selects.

To enable the PickMenu class to operate with different kinds of Menus in the future, you pass a Menu to PickMenu's constructor. This technique provides two advantages: First, when the menu options change, you can alter the contents of the Menu.java file without changing any of the code in any programs that use Menu. Second, you can extend Menu, perhaps to VegetarianMenu, LowSaltMenu, or KosherMenu, and still use the existing PickMenu class. When you pass any Menu or Menu subclass into the PickMenu constructor, the correct customer options will appear.

Tip: You have written many programs using GUI dialog boxes for input and output. However, you will use command-line prompts and input here to better illustrate Exception handling.

To create the PickMenu class:

1. Open a new file in your text editor, and then add the following first few lines of the PickMenu class with its three data fields (a Menu, and both a number and a String that reflect the customer's choice):

```
public class PickMenu
{
       Menu briefMenu;
       int choice;
       String guestChoice = new String();
```

2. Enter the following PickMenu constructor, which receives an argument representing a Menu. The constructor assigns the Menu that is the argument to the local Menu, and then calls the setChoice() method, which prompts the user to select from the available menu. The PickMenu() constructor method must `throw` an Exception because it contains the setGuestChoice() method, which uses keyboard input, and any method that uses keyboard input or calls a method that uses keyboard input must `throw` the potential Exception.

```
public PickMenu(Menu theMenu) throws Exception
{
       briefMenu = theMenu;
       setGuestChoice();
{
```

3. The following setGuestChoice() method displays the menu and reads keyboard data entry, so the method `throws` an Exception. Start the method by declaring a character and String for input:

```
public void setGuestChoice() throws Exception
{
       char newChar;
       String inputString = new String();
```

4. Add the following data-entry procedure, which is similar to others that you have written:

```
System.out.println("Choose from the following menu:");
briefMenu.displayMenu();
newChar = (char)System.in.read();
```

```
while(newChar >= '0' && newChar <= '9')
{
      inputString = inputString + newChar;
      newChar = (char)System.in.read();
}
System.in.read();
```

5. Add the following code to convert the entered String to an integer, and then use it as an argument to the getSelection() method that you wrote in the Menu class. Because briefMenu is a Menu (an instance of the Menu class), it has access to the getSelection() method. When you pass an integer to the getSelection() method, it returns one of the Strings in the menu. Here, you assign the returned String to the guestChoice field. Finally, you end the setGuestChoice() method with a closing curly brace.

```
      choice = Integer.parseInt(inputString);
      guestChoice = briefMenu.getSelection(choice);
}
```

6. Add the following getGuestChoice() method. This method is simpler; it returns the String that represents the customer's menu selection. Finally, include the closing curly brace for the PickMenu class.

```
   public String getGuestChoice()
   {
       return(guestChoice);
   }
}
```

7. Save the file as **PickMenu.java** in the Chapter.15 folder on your Student Disk, and then compile it using the `javac` command.

You created a Menu class that simply holds a list of food items, displays itself, and allows you to retrieve a specific item. You also created a PickMenu class that has fields that hold a user's specific selection from a given menu and methods to get and set values for those fields. The PickMenu class contains two methods that throw Exceptions, but no methods that contain ways to `catch` those Exceptions. Next you will write a program that uses the PickMenu class. This program can `catch` Exceptions that PickMenu `throws`.

To write the PlanMenu class:

1. Open a new file in your text editor, and start entering the following PlanMenu class, which will have just one method—a main() method:

```
public class PlanMenu
{
       public static void main(String[] args)
       {
```

2. Construct the following Menu named briefMenu, and also declare a PickMenu object that you name **entrée**. You do not want to construct a PickMenu object yet because you want to be able to `catch` the Exception

that the PickMenu constructor might throw. Therefore, you want to wait and construct the PickMenu object within a `try` block. For now, you will just declare entrée and assign it `null`. Also, you will declare a String that will hold the customer's menu selection.

```
Menu briefMenu = new Menu();
PickMenu entree = null;
String guestChoice = new String();
```

3. Write the following `try` block that constructs a PickMenu item. If the construction is successful, the next statement assigns a selection to the entrée object. Because entrée is a PickMenu object, it has access to the getGuestChoice() method in the PickMenu class, and you can assign the method's returned value to the guestChoice String.

```
try
{
      PickMenu selection = new PickMenu(briefMenu);
      entree = selection;
      guestChoice = entree.getGuestChoice();
}
```

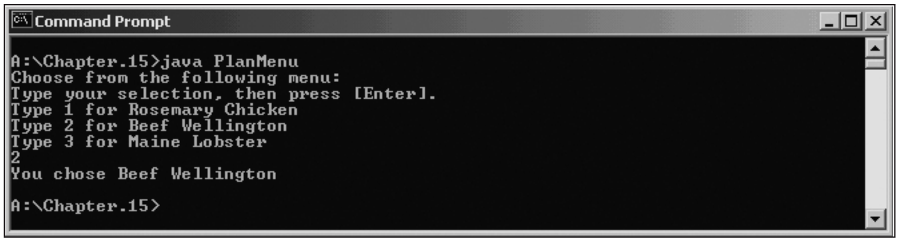
4. The `catch` block must immediately follow the `try` block. When the `try` block fails, guestChoice will not have a valid value, so recover from the Exception by assigning a value to guestChoice within the following `catch` block:

```
catch(Exception error)
{
      guestChoice = "an invalid selection";
}
```

5. Use the following code to print the customer's choice at the end of the PlanMenu program, and then add closing curly braces for the main() method and the class:

```
          System.out.println("You chose " + guestChoice);
     }
}
```

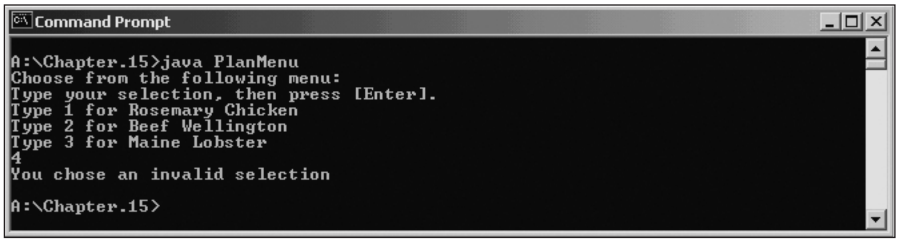
6. Save the file as **PlanMenu.java** in the Chapter.15 folder on your Student Disk, and then compile and execute it. Choose an entrée selection by typing its number from the menu, and then compare your results to Figure 15-13.



```
A:\Chapter.15>java PlanMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
2
You chose Beef Wellington

A:\Chapter.15>
```

Figure 15-13 Sample run of the PlanMenu program

7. The PlanMenu program works well when you enter a valid menu selection. One way that you can force an Exception to take place is to enter an invalid menu selection at the prompt. Run the PlanMenu program again, and type **4**, **A**, or any invalid value at the prompt. Entering 4 produces an ArrayIndexOutOfBoundsException, and entering A produces a NumberFormatException. If the program lacked the `try...catch` pair, either entry would halt the program. However, because the setGuestChoice() method in the PickValue class `throws` either type of Exception and the PlanMenu program catches it, guestChoice takes on the value "an invalid selection" and the program ends smoothly, as shown in Figure 15-14.



```
A:\Chapter.15>java PlanMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
4
You chose an invalid selection

A:\Chapter.15>
```

Figure 15-14 Exceptional run of the PlanMenu program

HANDLING EXCEPTIONS UNIQUELY WITH EACH `catch`

An advantage to using object-oriented exception-handling techniques is that you gain the ability to appropriately deal with Exceptions as you make conscious decisions about how to handle them. When you use a class and one of its methods `throws` an Exception, the class that `throws` the Exception does not have to `catch` it. Instead, your calling program can `catch` the Exception, and you can decide what you want to do. Just as a police officer can deal with a speeding driver differently depending on circumstances, you can react to Exceptions specifically for your current purposes. For example, the PickMenu class you created `throws` Exceptions. When you write new programs that use the PickMenu class, you can decide to handle error conditions differently within each program you write.

Next you will extend the Menu class to create a new class named VegetarianMenu. Subsequently, when you write a program that uses PickMenu with the VegetarianMenu, you can deal with any Exception differently than you did when you wrote the PlanMenu program.

To create the VegetarianMenu class:

1. Open a new file in your text editor, and then type the following class header for the VegetarianMenu class VegetarianMenu that extends Menu:

   ```
   public class VegetarianMenu extends Menu
   {
   ```

2. Provide new menu choices for the VegetarianMenu as follows:

   ```
   String[] vegEntreeChoice ={ "Spinach Lasagna",
           "Cheese Enchiladas", "Fruit Plate"};
   ```

3. Add the following constructor that calls the superclass constructor and assigns each vegetarian selection to the Menu superclass entreeChoice array, and then add the closing curly brace for the class:

   ```
     public VegetarianMenu()
     {
           super();
           for(int x = 0; x < vegEntreeChoice.length; ++x)
                   entreeChoice[x] = vegEntreeChoice[x];
     }
     }
   ```

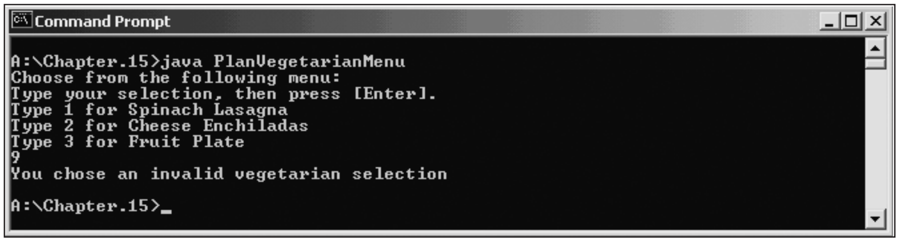
4. Save the class as **VegetarianMenu.java** in the Chapter.15 folder on your Student Disk, and then compile it.

5. Now write a program that uses VegetarianMenu. You could write any program, but for demonstration purposes, you can simply modify PlanMenu.java. Open the **PlanMenu.java** file in your text editor, and then immediately save it as **PlanVegetarianMenu.java**.

6. Change the class name in the header to **`PlanVegetarianMenu`**.
7. Change the first statement within the main() method as follows so it declares a VegetarianMenu instead of a Menu:

   ```
   VegetarianMenu briefMenu = new VegetarianMenu();
   ```
8. Change the guestChoice assignment statement in the `catch` block as follows so it is specific to the program that uses the VegetarianMenu:

   ```
   guestChoice = "an invalid vegetarian selection";
   ```
9. Save the file in the Chapter.15 folder on your Student Disk, compile it, and then run the program. When you see the vegetarian menu, enter a valid selection and confirm that the program works correctly. Run the program again and enter an invalid selection. The error message, shown in Figure 15-15, identifies your invalid entry as "an invalid vegetarian selection". Remember that you did not change the PickMenu class. Your new PlanVegetarianMenu program uses the PickMenu class that you wrote and compiled before a VegetarianMenu ever existed. However, because PickMenu `throws` uncaught Exceptions, you can handle those Exceptions as you see fit in any new programs in which you `catch` them.



```
A:\Chapter.15>java PlanVegetarianMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Spinach Lasagna
Type 2 for Cheese Enchiladas
Type 3 for Fruit Plate
9
You chose an invalid vegetarian selection

A:\Chapter.15>_
```

Figure 15-15 Exceptional run of the PlanVegitarianMenu program

TRACING EXCEPTIONS THROUGH THE CALL STACK

When one method calls another, the computer's operating system must keep track of where the method call came from, and program control must return to the calling method when the called method is completed. For example, if methodA calls methodB, the operating system has to "remember" to return to methodA when methodB ends. Likewise, if methodB calls methodC, then while methodC executes, the computer must "remember" that it is going to return to methodB and, eventually, to return methodA. The memory location known as the **call stack** is where the computer stores the list of locations to which the system must return.

When a method `throws` an Exception, and if the method does not `catch` the Exception, then the Exception is thrown to the next method up the call stack, or in other words, to the method that called the offending method. Figure 15-16 shows how the call stack works. If methodA calls methodB, and methodB calls methodC, and methodC `throws` an Exception, then Java first looks for a `catch` block in methodC. If none exists, then Java looks for the same thing in methodB. If methodB does not have a `catch` block, then Java looks to methodA. If methodA cannot `catch` the Exception, then it is thrown to the operating system.

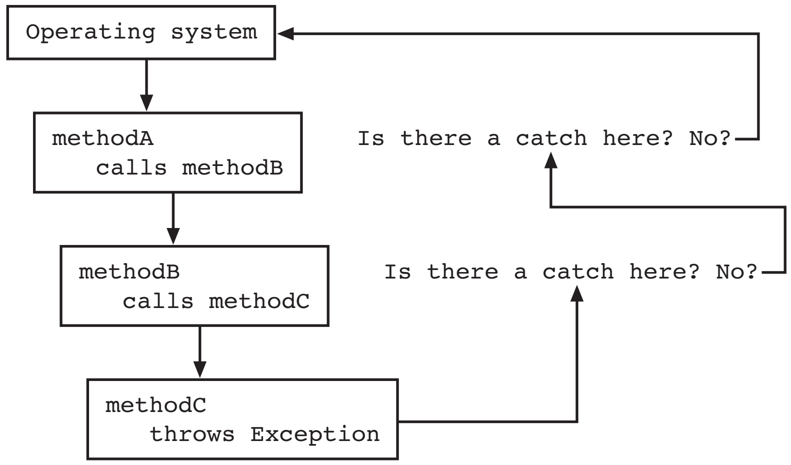


Figure 15-16 Cycling through the call stack

The technique of cycling through the methods in the stack has great advantages because it allows methods to handle Exceptions wherever the programmer has decided it is most appropriate. However, when a program uses several classes, this system's disadvantage is that it is very difficult for the programmer to locate the original source of an Exception.

You have already used the Throwable method getMessage() to obtain information about an Exception. Another useful Exception method is the printStackTrace() method. When you catch an Exception, you can call printStackTrace() to display a list of methods in the call stack so you can determine the location of the Exception.

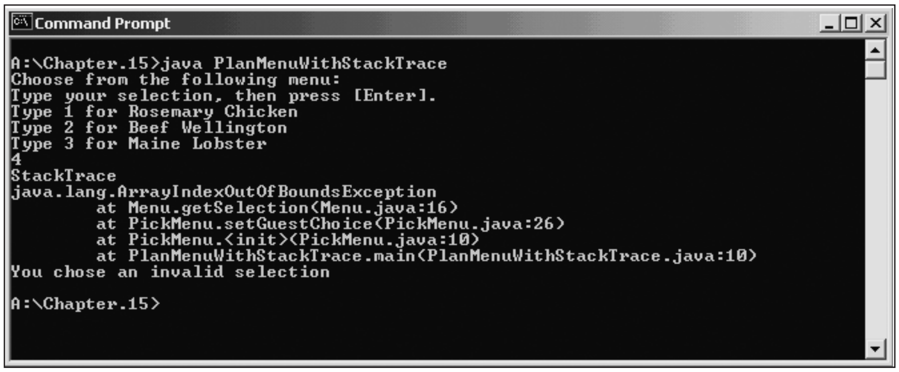
To use the printStackTrace() method:

1. Open the **PlanMenu.java** file in your text editor, and then save it as **PlanMenuWithStackTrace.java**.
2. Change the class header to **`PlanMenuWithStackTrace`**.

3. Position your insertion point within the `catch` block after the statement `guestChoice = "an invalid selection";`, and then press **[Enter]** to start a new line.
4. Type the following two new statements to identify and print the stack trace:

```
System.out.println("Stack Trace");
error.printStackTrace();
```

5. Save the file in the Chapter.15 folder on your Student Disk, and then compile and execute it. After the menu appears, enter an invalid selection. If you entered 4, your screen looks like Figure 15-17. If you read the list that follows the Stack Trace heading, you see that an ArrayIndexOutOfBoundsException occurred in the method Menu.getSelection(). That method was called by the PickMenu.setGuestChoice() method, which in turn was initiated by the PickMenu constructor. The PickMenu constructor was called from the PlanMenuWithStackTrace.main() method. You see the line number as additional information within each method where the Exception occurred (your line numbers might be different). If you did not understand why entering 4 caused an error, you would use the stack trace information to examine the Menu.getSelection() method as the first source of the error. Using printStackTrace() can be a helpful debugging tool.
6. Run the **PlanMenuWithStackTrace** program again, and then enter **A** for the user selection. You can see from the stack trace that this time the Exception does not originate in the Menu.getSelection() method. This program stops at the parseInt() method before the program attempts getSelection().



```
A:\Chapter.15>java PlanMenuWithStackTrace
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
4
StackTrace
java.lang.ArrayIndexOutOfBoundsException
        at Menu.getSelection(Menu.java:16)
        at PickMenu.setGuestChoice(PickMenu.java:26)
        at PickMenu.<init>(PickMenu.java:10)
        at PlanMenuWithStackTrace.main(PlanMenuWithStackTrace.java:10)
You chose an invalid selection

A:\Chapter.15>
```

Figure 15-17 Exceptional run of the PlanMenuWithStackTrace program

Tip: Often, you do not want to place a printStackTrace() method call in a finished program. The typical program user has no interest in the cryptic messages that print. However, while you are developing a program, printStackTrace() can be a useful tool for diagnosing your program's problems.

Creating Your Own Exceptions

Java provides over 40 categories of Exceptions that you can throw in your programs. However, Java's creators could not predict every condition that might be an Exception in your programs. For example, you might want to declare an Exception when your bank balance is negative or when an outside party attempts to access your e-mail account. Most organizations have specific rules for exceptional data; for example, an employee number must not exceed three digits, or an hourly salary must not be less than the legal minimum wage. Of course, you can handle these potential error situations with `if` statements, but Java also allows you to create your own Exceptions.

To create your own throwable Exception, you must extend a subclass of Throwable. Recall from Figure 15-1 that Throwable has two subclasses, Exception and Error, which are used to distinguish between recoverable and nonrecoverable errors. Because you always want to create your own Exceptions for recoverable errors, you should extend your Exceptions from the Exception class. You can extend any existing Exception subclass, such as ArithmeticException or NullPointerException, but usually you want to extend directly from Exception.

Tip: When you create an Exception, it's conventional to end its name with Exception.

Next you will create a PartyException class for Event Handlers Incorporated. The PartyException class has just one method—a constructor. You can include data fields and other methods within the PartyException class if you want. For example, you might want the PartyException class to contain a customized toString() method that you can use to display party details. To keep this example simple, however, you will include only the constructor. The constructor will take a String argument representing the name of the party, such as the *Jones* party. You will pass this String to the Exception superclass so you can use the String within superclass methods, such as getMessage().

To write your own Exception:

1. Open a new file in your text editor, and then type the following PartyException class:

```
public class PartyException extends Exception
{
       public PartyException(String s)
       {
```

```
                super(s);
        }
}
```

2. Save the file as **PartyException.java** in the Chapter.15 folder on your Student Disk, and then compile it.

Next you will create a Party class that holds information about any party hosted by Event Handlers Incorporated. The Party class holds two fields—the name of the party host and the number of guests—and it contains a constructor that requires values for both fields. Event Handlers does not host parties with fewer than 10 guests. Therefore, you want to test the guest number in the constructor, and throw a PartyException when the guest number is less than 10. The PartyException class constructor requires a String argument, so pass the name of the party host to the Exception. That way, you can use the host's name in error messages generated by the Exception class.

Tip

You can throw any type of Exception at any time, not just Exceptions of your own creation. For example, within any program you can code `throw(new RuntimeException());`. Of course, you would want to do so only with good reason because Java handles RuntimeExceptions for you by stopping the program. Because you cannot anticipate every possible error, Java's automatic response is often the best course of action.

To create the Party class:

1. Open a new file in your text editor, and then type the following Party class:

```
public class Party
{
        String host = new String();
        int guests;
        public Party(String hst, int gst) throws
        PartyException
        {
                host = hst;
                guests = gst;
                if(gst < 10)
                        throw(new PartyException(hst));
        }
}
```

2. Save the file as **Party.java** in the Chapter.15 folder on your Student Disk, and then compile it.

Next you will write a program that instantiates a few Party objects. When you run the program, you can observe which objects generate PartyExceptions.

To write the ThrowParty program:

1. Open a new text file, and then type the following first few lines of the ThrowParty program and its main() method:

   ```
   public class ThrowParty
   {
          public static void main(String[] args)
   ```

2. Enter the following code to attempt to construct three Party objects in a `try` block:

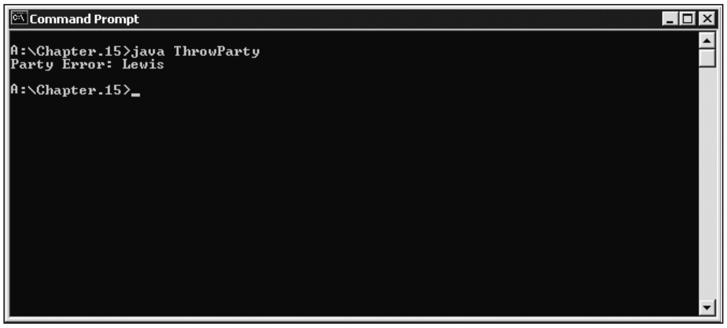
   ```
   try
   {
          Party first = new Party("Jones",15);
          Party second = new Party("Lewis",5);
          Party third = new Party("Newman",10);
   }
   ```

3. Enter the following code to `catch` any PartyExceptions and use the Exception class getMessage() method to display a message, and then add two closing curly braces:

   ```
     catch(PartyException error)
     {
      System.out.println("Party Error: " +
         error.getMessage ());
     }
    }
   }
   ```

4. Save the file as **ThrowParty.java** in the Chapter.15 folder on your Student Disk.

5. Compile **ThrowParty.java**, and then run the program. Compare your results to Figure 15-18. The two Party objects constructed with 10 or more guests compiled successfully, but the Party object with only 5 guests generated a PartyException.



```
A:\Chapter.15>java ThrowParty
Party Error: Lewis

A:\Chapter.15>_
```

Figure 15-18 Output of the ThrowParty program

You should not create an excessive number of special Exception types for your classes, especially if the Java development environment already contains an Exception that will `catch` the error. Extra Exception types add complexity for other programmers who will use your classes. However, when appropriate, specialized Exception classes provide an elegant way for you to handle error situations. They enable you to separate your error code from the usual, nonexceptional sequence of events. They also allow for errors to be passed up the stack and traced.

Chapter Summary

- An exception is an unexpected or error condition. The object-oriented techniques to manage such errors comprise the group of methods known as exception handling. In Java, there are two basic classes of errors—Error and Exception. Both of these classes descend from the Throwable class.
- In object-oriented terminology, you "try" a procedure that might not complete correctly. A method that detects an error condition or Exception "throws an Exception", and the block of code that processes the error "catches the Exception".
- Exceptions inherit the getMessage() method from the Throwable class.
- You can place as many statements as you need within a `try` block; only the first error-generating statement `throws` an Exception. You can catch as many Exceptions as you want, or you can let the operating system handle them.
- When you have actions that always must occur at the end of a `try...catch` sequence, you use a `finally` block.
- In a traditional, non-object-oriented program, error-checking code is complex. Object-oriented exception-handling allows you to isolate error-handling code.
- You can use the clause `throws <name>Exception` after the method header to indicate the type of Exception that might be thrown.
- An advantage of using object-oriented exception-handling techniques is they afford you the ability to make your reaction to Exceptions specific for your current purposes—you can handle the same Exception differently in every application, or choose not to handle it at all.
- The call stack is the memory location where the computer stores the list of locations to which the system must return after method calls. When a method `throws` an Exception, and if the method does not `catch` the Exception, then the Exception is thrown to the next method "up" the call stack. You can display this list using the printStackTrace() method.
- You can create your own Exceptions by extending the Exception class.

Review Questions

1. In object-oriented programming terminology, an unexpected or error condition is a(n) ________________.
   a. anomaly
   b. aberration
   c. deviation
   d. exception
2. All Java Exceptions are ________________.
   a. Errors
   b. RuntimeExceptions
   c. Throwables
   d. Omissions
3. Which of the following statements is true?
   a. Exceptions are more serious than Errors.
   b. Errors are more serious than Exceptions.
   c. Errors and Exceptions are equally serious.
   d. Exceptions and Errors are the same thing.
4. The method that ends the current application and returns control to the operating system is ________________.
   a. System.end()
   b. System.done()
   c. System.exit()
   d. System.abort()
5. In object-oriented terminology, you ________________ a procedure that might not complete correctly.
   a. `try`
   b. `catch`
   c. `handle`
   d. `encapsulate`
6. A method that detects an error condition or Exception ________________ an Exception.
   a. tries
   b. catches
   c. handles
   d. encapsulates

7. A `try` block includes all of the following elements except ________________.
   a. the keyword `try`
   b. the keyword `catch`
   c. curly braces
   d. statements that might cause Exceptions
8. The segment of code that handles or takes appropriate action following an exception is a ________________ block.
   a. `try`
   b. `catch`
   c. `throws`
   d. `handles`
9. You ________________ within a `try` block.
   a. must place only a single statement
   b. can place any number of statements
   c. must place at least two statements
   d. must place a `catch` block
10. If you `try` three statements, and include three `catch` blocks, and the second `try` statement `throws` an Exception, then ________________.
    a. the first `catch` block executes
    b. the first two `catch` blocks execute
    c. only the second `catch` block executes
    d. the first matching `catch` block executes
11. When a `try` block does not generate an Exception and you have included multiple `catch` blocks, ________________.
    a. they all execute
    b. only the first one executes
    c. only the first matching one executes
    d. no `catch` blocks execute
12. The `catch` block that begins `catch (Exception e)` can catch Exceptions of type ________________.
    a. IOException
    b. ArithmeticException
    c. both of the above
    d. none of the above

13. The code within a `finally` block executes when the try block ________________.
    a. identifies one or more Exceptions
    b. does not identify any Exceptions
    c. either a or b
    d. neither a nor b
14. An advantage to using a `try...catch` block is that exceptional events are ________________.
    a. eliminated
    b. reduced
    c. integrated with regular events
    d. isolated from regular events
15. Which methods can `throw` an Exception?
    a. Methods with a `throws` clause
    b. Methods with a `catch` block
    c. Methods with both a `throws` clause and a `catch` block
    d. Any method
16. A method can ________________.
    a. check for errors but not handle them
    b. handle errors but not check for them
    c. either of the above
    d. neither of the above
17. When you use any method, you must know three pieces of information to use the method to its full potential; but you don't need to know ________________.
    a. the method's return type
    b. the type of arguments the method requires
    c. the number of statements within the method
    d. the type of Exceptions the method `throws`
18. The memory location where the computer stores the list of locations to which the system must return is known as the ________________.
    a. registry
    b. call stack
    c. chronicle
    d. archive

19. You can get a list of the methods through which an Exception has traveled by using the ________________ method.
    a. getMessage()
    b. callStack()
    c. getPath()
    d. printStackTrace()
20. To create your own Exception that you can throw, you must extend a subclass of ________________.
    a. Object
    b. Throwable
    c. Exception
    d. Error

EXERCISES

1. Write a program named GoTooFar in which you declare an array of five integers and store five values in the array. Initialize a subscript to zero. Write a `try` block in which you access each element of the array, subsequently increasing the subscript by 1. Create a `catch` block that catches the eventual ArrayIndexOutOfBoundsException, and then print to the screen the message, "Now you've gone too far." Save the program as **GoTooFar.java** in the Chapter.15 folder of your Student Disk.
2. The Integer.parseInt() method requires an integer argument. Write a program in which you try to parse a String. Catch the NumberFormatExceptionError that is thrown, and then display an appropriate error message. Save the program as **TryToParseString.java** in the Chapter.15 folder of your Student Disk.
3. Write an application program that prompts the user to enter a number to use as an array size, then attempt to declare an array using the entered size. If the array is created successfully, display an appropriate message. Use a `catch` block that executes if the array size is non-numeric or negative. Save the program as **NegativeArray.java** in the Chapter.15 folder of your Student Disk.
4. Write a program that throws and catches an ArithmeticException. Declare a variable and assign it a value. Test the variable, and if it is negative, throw an ArithmeticException. Otherwise, use the Math.sqrt() method to determine the square root. Save the program as **SqrtError.java** in the Chapter.15 folder of your Student Disk.
5. Create an EmployeeException class whose constructor receives a String that consists of an employee's ID and pay rate. Create an Employee class with two fields, idNum and hourlyWage. The Employee constructor requires values for both fields. Upon construction, throw an EmployeeException if the hourlyWage is less

than \$6.00 or over \$50.00. Write a program that establishes at least three Employees with hourlyWages that are above, below, and within the allowed range. Save the program as **ThrowEmployee.java** in the Chapter.15 folder of your Student Disk.

6. a. Create an IceCreamConeException class whose constructor receives a String that consists of an ice cream cone's flavor and number of scoops. Create an IceCreamCone class with two fields—iceCreamFlavor and scoops. The IceCreamCone constructor calls two data-entry methods—getFlavor() and getScoops(). The getScoops() method `throws` an IceCreamConeException when the scoop quantity exceeds 3. Write a program that establishes several IceCreamCone objects and handles the Exception. Save the program as **ThrowIceCream.java** in the Chapter.15 folder of your Student Disk.

   b. Modify the IceCreamCone getFlavor() method to ensure that the user enters a valid flavor. Allow at least four flavors of your choice. If the user's entry does not match a valid flavor, throw an IceCreamConeException. Write a program that establishes several IceCreamCone objects and handles the new Exception. Save the program as **ThrowIceCream2.java** in the Chapter.15 folder of your Student Disk.

7. Write a program that displays a student ID number and asks the user to enter a numeric test score for the student. Create a ScoreException class, and throw a ScoreException for that class if the user does not enter a valid score (less than or equal to 100). Catch the ScoreException and then display an appropriate message. Save the program as **TestScore.java** in the Chapter.15 folder of your Student Disk.

8. Write a program that displays a student ID number and asks the user to enter a test letter grade for the student. Create an Exception class named GradeException, and throw a GradeException if the user does not enter a valid letter grade. `Catch` the GradeException and then display an appropriate message. Save the program as **TestGrade.java** in the Chapter.15 folder of your Student Disk.

9. Write an applet that prompts the user for a color name. If it is not red, white, or blue, throw an Exception. Otherwise, change the applet's background color appropriately. Save the program as **RWBApplet.java** in the Chapter.15 folder of your Student Disk.

10. Write an applet that prompts the user for an ID number and an age. Create an Exception class and throw an Exception of that class if the ID is not in the range of valid ID numbers (zero through 899), or if the age is not in the range of valid ages (0 through 89). Catch the Exception and then display an appropriate message. Save the program as **BadIDAndAge.java** in the Chapter.15 folder of your Student Disk.

11. Each of the following files in the Chapter.15 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugFifteen1.java will become FixDebugFifteen1.java. You will also use a file named DebugEmployeeIDException.java with the DebugFifteen4.javafile.
    a. DebugFifteen1.java
    b. DebugFifteen2.java
    c. DebugFifteen3.java
    d. DebugFifteen4. java

CASE PROJECT

Gadgets by Mail sells many interesting items through its catalogs. Write a program that prompts the user for an item number and quantity ordered. Create an Exception class named OrderException, and throw an OrderException if the user does not enter a valid item number (at least 111, but no more than 999) or quantity (at least 1, but no more than 50). Catch the OrderException and display an appropriate message. If the item number and quantity are valid, display the final price, which is $2.00 per item, no matter what the item number is.

CHAPTER

16 File Input and Output

In this chapter, you will:

- ♦ Use the File class
- ♦ Understand data file organization and streams
- ♦ Use streams
- ♦ Write to and read from a file
- ♦ Write formatted file data
- ♦ Read formatted file data
- ♦ Use a variable filename
- ♦ Create random access files

Haven't I seen you spending a lot of time at the keyboard lately?" asks Lynn Greenbrier one day at Event Handlers Incorporated.

"I'm afraid so," you answer. "I'm trying to write a program that displays a month's scheduled events, one at a time. Every time I run it, I have to enter the data for every event—the host's name, the number of guests, and so on."

"You're typing all the data over and over again?" Lynn asks in disbelief. "It's time for me to show you how to save data to a file."

Previewing a Program That Uses File Data

Event Handlers Incorporated stores a record of each scheduled event in a file. Any employee in the company can view the scheduled events on-screen. You will create a similar program in this chapter; however, you can now use a completed version of the Chap16ReadEventFile program that is saved in the Chapter.16 folder on your Student Disk.

Tip: For convenience, the data file used in this example is stored on a floppy disk. However, business files are usually stored on a hard disk, either locally or on a network server. No matter where a data file is physically located, the process of saving and retrieving the file is the same.

To use the Chap16ReadEventFile class:

1. Go to the command prompt for the Chapter.16 folder on your Student Disk, type **`java Chap16ReadEventFile`**, and then press **[Enter]**. This program lets you view previously stored data about events, one event at a time.
2. Click the **View Event** button. The data for the Albertson event appears and shows that the event is scheduled for the 12th day of the month with 100 guests. See Figure 16-1.

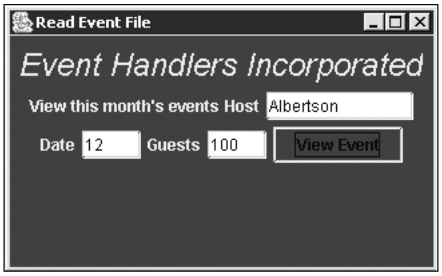


Figure 16-1 Chap16ReadEventFile program

3. Click the **View Event** button again to view the data for five additional events. Click the **Close** button to exit the program at any time. The program will also exit automatically when you reach the end of the stored data file.

Using the File Class

Computer users use the term **file** to describe the objects that they store on permanent storage devices, such as hard, floppy, or zip disks, reels of magnetic tape, or compact discs. Some files are **data files** that contain facts and figures, such as employee numbers, names, and salaries; some files are **program files**, also called applications, that store software instructions. Other files can store graphics, text, or operating system instructions.

Although their contents vary, files have many common characteristics—each file occupies a section of disk (or other storage device) space, has a name and a specific time of creation. You can use Java's **File class** to gather file information. The File class does not provide any opening, processing, or closing capabilities for files. Rather, you use the File class to obtain information about a file, such as whether it exists or is open, its size, and its last modification date.

You must include the statement `import java.io.*` in any program that uses the File class. The java.io package contains all the classes you use in file processing. The File class is a direct descendant of the Object class. You can create a File object using a constructor that includes a filename; for example, `File someData = new File("data.txt");`, where data.txt is a file on the default disk drive. You can also specify a path for the file, as in `File someData = new File("A:\\Chapter.16\\data.txt");`, in which the argument to the constructor contains a disk drive and path. Table 16-1 lists some useful File class methods.

Tip

The *io* in java.io stands for *i*nput/*o*utput.

Tip

Recall that the backslash (\) is used as part of an escape sequence in Java. You must type two backslashes to indicate a single backslash to the operating system. You learned about the escape sequence in Chapter 2.

Table 16-1 Selected File class methods

Method Signature	Purpose
`boolean canRead()`	Returns `true` if a file is readable
`boolean canWrite()`	Returns `true` if a file is writeable
`boolean exists()`	Returns `true` if a file exists
`String getName()`	Returns the file's name
`String getPath()`	Returns the file's path
`String getParent()`	Returns the name of the folder in which the file can be found
`long length()`	Returns the file's size
`long lastModified()`	Returns the time the file was last modified; this time is system dependent and should be used only for comparison with other files' times, and not as an absolute time

Next you will write a class that examines a file and prints appropriate messages concerning its status.

To create a class that uses a File object:

1. Open a new file in your text editor, and then type the following first few lines of a class that checks a file's status:

   ```
   import java.io.*;
   public class CheckFile
   {
          public static void main(String[] args)
          {
   ```

2. Enter the following line to create a File object that represents a disk file named data.txt. Although the filename on the disk is data.txt, within the program the filename is myFile.

   ```
   File myFile = new File("data.txt");
   ```

3. Enter the following `if…else` statements to test for the file's existence. If the File object myFile exists, print its name and size, and then test whether the file can be read or written. If the file does not exist, simply print a message indicating that fact.

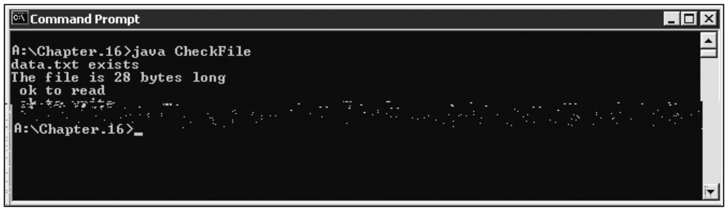
   ```
   if(myFile.exists())
   {
          System.out.println(myFile.getName() + " exists");
          System.out.println("The file is " +
             myFile.length () + " bytes long");
          if(myFile.canRead())
                 System.out.println(" ok to read");
          if(myFile.canWrite())
                 System.out.println(" ok to write");
   }
   else
       System.out.println("File does not exist");
   ```

4. Add a closing curly brace for the main() method and a closing curly brace for the class.

5. Save the file as **CheckFile.java** in the Chapter.16 folder on your Student Disk, and then compile the file.

6. Open a new file in your text editor, and then type the company name, **Event Handlers Incorporated!**. Save this file as **data.txt** in the current directory (the Chapter.16 folder on your Student Disk).

7. Run the CheckFile program using the command `java CheckFile`. The output appears in Figure 16-2. The file is 28 bytes long because each character you typed, including spaces and punctuation, consumes one byte of storage.



```
A:\Chapter.16>java CheckFile
data.txt exists
The file is 28 bytes long
 ok to read
 ok to write

A:\Chapter.16>_
```

Figure 16-2 Output of the CheckFile program

Help If you added comments to the beginning of your data.txt file or mistyped the company name, then the total number of characters in your file might differ from 28.

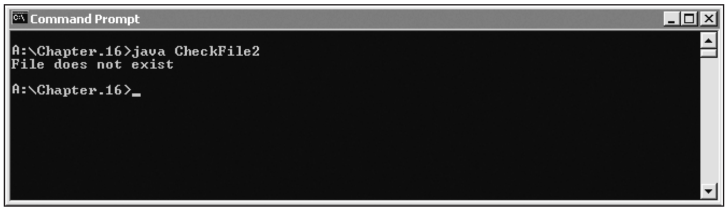
Next you will change the program to test for a file that does not exist.

To check for a nonexistent file:

1. Open the **CheckFile.java** file from the Chapter.16 folder in your text editor, and then immediately save it as **CheckFile2.java**.
2. Change the class name to **`CheckFile2`**.
3. Change the filename in the File constructor so that it refers to a nonexistent file:

   ```
   File myFile = new File("nodata.txt");
   ```

4. Save the file, and then compile and run the program. Unless you have a file named nodata.txt on your Student Disk, the output looks like Figure 16-3.



```
A:\Chapter.16>java CheckFile2
File does not exist

A:\Chapter.16>_
```

Figure 16-3 Output of the CheckFile2 program

In the preceding steps, the program found the file named data.txt because the file was physically located in the current directory from which you were working. You can check the status of files in other directories by using a File constructor with two String arguments. The first String represents a path to the filename, and the second String represents the filename. For example, `File someFile = new File("\\com\\EventHandlers","data.txt");` refers to the data.txt file located in the EventHandlers folder within the com folder in the root directory.

Next you will create a second data file so that you can compare its size and time stamp with the data.txt file.

To create a data2.txt file and a program for comparing data2.txt to data.txt:

1. Open a new file in your text editor, and then type a shorter version of the company name, **Event Handlers**.
2. Save the file as **data2.txt** in the Chapter.16 folder on your Student Disk.
3. Open a new file in your text editor, and then type the following first few lines of the CheckTwoFiles program:

```
import java.io.*;
public class CheckTwoFiles
{
       public static void main(String[] args)
       {
```

4. Enter the following code to declare two file objects:

```
File f1 = new File("data.txt");
File f2 = new File("data2.txt");
```

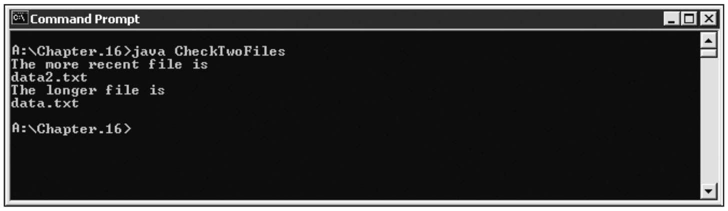
5. Enter the following code to determine whether both files exist. If they do, comparing their creation times determines which file has the more recent time stamp, and then the program prints the filename of that file. (*Note:* Do not add a closing curly brace for the `if` statement in this step; in the next step you will continue to add statements that belong within this `if` structure.)

```
if(f1.exists() && f2.exists())
{
        System.out.println("The more recent file is ");
        if(f1.lastModified() > f2.lastModified())
                 System.out.println(f1.getName());
        else
                 System.out.println(f2.getName());
```

6. Enter additional statements within the `if` block that executes when both files exist. These statements compare the length of the files and print the name of the longer file—that is, the one that contains more bytes:

```
System.out.println("The longer file is ");
if(f1.length() > f2.length())
         System.out.println(f1.getName());
else
         System.out.println(f2.getName());
```

7. Add three curly braces—one to end the `if` statement that checks whether both files exist, one for the main() method, and one for the class.
8. Save the file as **CheckTwoFiles.java** in the Chapter.16 folder of your Student Disk, and then compile and run the program. The output appears in Figure 16-4. Note that the data2.txt file was created after the data.txt file, so it is more recent, but it has fewer characters.



```
A:\Chapter.16>java CheckTwoFiles
The more recent file is
data2.txt
The longer file is
data.txt

A:\Chapter.16>
```

Figure 16-4 Output of the CheckTwoFiles program

UNDERSTANDING DATA FILE ORGANIZATION AND STREAMS

Most businesses generate and use large quantities of data every day. You can store data in variables within a program, but this type of storage is temporary. When the program ends, the variables no longer exist and the data is lost. Variables are stored in the computer's main or primary memory, which is called RAM (random access memory). When you need to retain data for any significant amount of time, you must save the data on a permanent, secondary storage device such as a floppy disk, hard drive, or compact disc (CD).

Tip

Because you can erase data from files, some programmers prefer the term *persistent* storage over permanent storage. In other words, you can remove data so it is not permanent, but the data remains in the file even when the computer loses power, so, unlike RAM, the data will persist, or persevere.

Data used by businesses is stored in a data hierarchy, as shown in Figure 16-5. The smallest, useful piece of data that is of interest to most people is the character. A character is any one of the letters, numbers, or other special symbols, such as punctuation marks, you can read and to which you can assign meaning. Characters are made up of bits (the zeros and ones that represent computer circuitry), but people who use data are not concerned with whether the internal representation for an A is 01000001 or 10111110.

Tip

Java uses Unicode to represent its characters. You first learned about Unicode in Chapter 2.

When businesses use data, they group characters into fields. A **field** is a group of characters that has some meaning. For example, the characters *T*, *o*, and *m* might represent your first name. Other data fields might represent items such as last name, Social Security number, zip code, and salary.

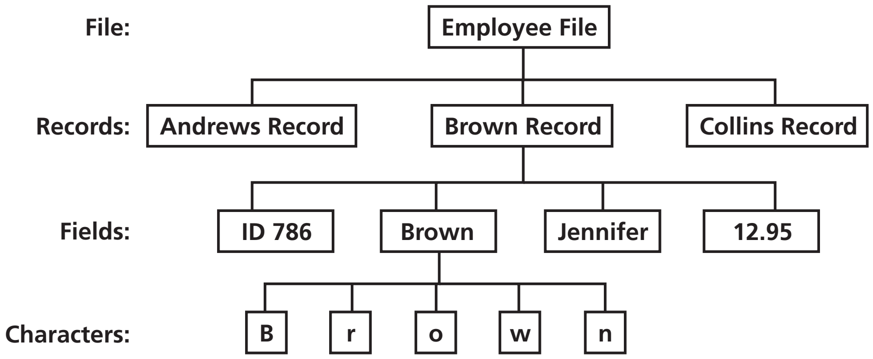


Figure 16-5 Data hierarchy

Fields are grouped together to form **records**. An individual's first and last names, Social Security number, zip code, and salary represent that individual's record. When programming in Java, you have created many classes, such as an Employee class or a Student class. You can think of the data typically stored in each of these classes as a record. These classes contain individual variables that represent data fields. A business's data records usually represent a person, item, sales transaction, or some other concrete object or event.

Records are grouped to create files. **Files** consist of related records, such as a company's personnel file that contains one record for each company employee. Some files have only a few records; perhaps your professor maintains a file for your class with 25 records—one record for each student. Other files contain thousands or even millions of records. For example, a large insurance company maintains a file of policyholders, and a mail-order catalog company maintains a file of available items.

Before a program can use a data file, the program must open the file. Similarly, when you finish using a file, the program should close the file. If you fail to close an input file—that is, a file from which you are reading data, there usually are not any serious consequences; the data still exists in the file. However, if you fail to close an output file—that is, a file to which you are writing data, the data may become inaccessible. You should always close every file you open, and you should close the file as soon as you no longer need it. When you leave a file open for no reason, you use computer resources and your computer's performance suffers. Also, particularly within a network, another program might be waiting to use the file.

While people view files as a series of records with each record containing data fields, Java views files as a series of bytes. When you perform an input operation in a program, you can picture bytes flowing into your program from an input device through a **stream**, which functions as a pipeline or channel. When you perform output, some bytes flow out of your program through another stream to an output device, as shown in Figure 16-6. A stream is an object, and like all objects, streams have data and methods. The methods allow you to perform actions such as opening, closing, and flushing (clearing) the stream.

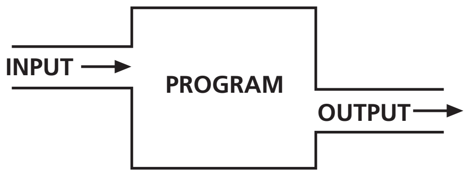


Figure 16-6 File streams

Most streams flow in only one direction; each stream is either an input or output stream. You might open several streams at once within your program. For example, three streams are required by a program that reads a data disk. One input stream checks the data for invalid values, and then one output stream writes some records to a file of valid records; another output stream writes other records to a file of invalid records.

Tip

Random access files use streams that flow in two directions. You will use a random access file later in this chapter.

Using Streams

Figure 16-7 shows a partial structure of Java's Stream classes; it shows that InputStream and OutputStream are subclasses of the Object class. **InputStream** and **OutputStream** are abstract classes that contain methods for performing input and output. As abstract classes, these classes contain methods that must be overridden in their child classes. The capabilities of the most commonly used classes that provide input and output are summarized in Table 16-2.

Table 16-2 Description of selected classes used for input and output

Class	Description
InputStream	Abstract class containing methods for performing input
OutputStream	Abstract class containing methods for performing output
FileInputStream	Child of InputStream that provides the capability to read from disk files
FileOutputStream	Child of OutputStream that provides the capability to write to disk files
PrintStream	Child of FilterOutputStream, which is a child of OutputStream; PrintStream handles output to a system's standard (or default) output device, usually the monitor
BufferedInputStream	Child of FilterInputStream, which is a child of InputStream; BufferedInputStream handles input from a system's standard (or default) input device, usually the keyboard

```
Object
|
+--InputStream
|   |
|   +--FileInputStream
|   |
|   +--FilterInputStream
|     |
|     +--DataInputStream
|     |
|     +--BufferedInputStream
|
+--OutputStream
|   |
|   +--FileOutputStream
|   |
|   +--FilterOutputStream
|     |
|     +--DataOutputStream
|     |
|     +--BufferedOutputStream
|     |
|     +--PrintStream
|
+--RandomAccessFile
```

Figure 16-7 Partial structure of the Stream classes

Java's System class declares a PrintStream object. This object is System.out, which you have used extensively in this book. Besides System.out, the System class defines an additional PrintStream object named System.err. The output from System.err and System.out can go to the same device; in fact, System.err and System.out are both directed to the command line on the monitor. The difference is that System.err usually is reserved for error messages, and System.out is reserved for valid output. You can direct either System.err or System.out to a new location, such as a disk file or printer. For example, you might want to keep a hard copy log of the error messages generated by a program, but direct the standard output to a disk file.

Figure 16-7 shows that the InputStream class is parent to FilterInputStream, which is parent to BufferedInputStream. The object System.in is a BufferedInputStream object. The System.in object captures keyboard input. Recall that you have used this object with its read() method. A **buffer** is a small memory location that you use to hold data temporarily. The BufferedInputStream class allows keyboard data to be held until the user presses [Enter]. That way, the user can backspace over typed characters to change the data before the program stores it. This allows the operating system—instead of your program—to handle the complicated tasks of deleting characters as the user backspaces, and then replacing the deleted characters with new ones.

Tip

Using a buffer to hold input or output improves program performance. Input and output operations are relatively slow compared to computer processor speeds. Holding input or output until there is a "batch" makes programs run faster.

You can create your own InputStream and OutputStream objects and assign to them System.in and System.out, respectively. Then you can use the InputStream's read() method to read in one character at a time from the location you choose. The read() method returns an integer that represents the Unicode value of the typed character; it returns a value of -1 when it encounters an end-of-file condition, known as EOF.

Tip

You can also identify EOF by throwing an EOFException. You will use this technique later in this chapter.

Next you will create InputStream and OutputStream objects so you can read from the keyboard and write to the screen. Of course, you have already written many programs that read from the keyboard and write to the screen without using these objects. By using them here with the default input/output devices, you can easily modify the InputStream and OutputStream objects at a later time; then you can use whatever input and output devices you choose.

To create a program that reads from the keyboard and writes to the screen:

1. Open a new file in your text editor, and then type the following first few lines of a program that will allow a user to input data from the keyboard and then will echo that data to the screen. The class name is ReadKBWriteScreen:

   ```
   import java.io.*;
   public class ReadKBWriteScreen
   {
   ```

2. Add the following header and opening curly brace for the main() method. The main() method `throws` an IOException because you will perform input and output operations.

   ```
   public static void main(String[] args) throws IOException
   {
   ```

3. Enter the following code to declare InputStream and OutputStream objects, as well as an integer to hold each character the user types:

   ```
   InputStream istream;
   OutputStream ostream;
   int c;
   ```

4. Enter the following code to assign the System.in object to istream, and the System.out object to ostream. Then add a prompt telling the user to type some characters.

```
istream = System.in;
ostream = System.out;
System.out.println("Type some characters ");
```

5. Use the following `try` block to read from the file. If an IOException occurs, you can print an appropriate message. Within the `try` block, execute a loop that reads from the keyboard until the end-of-file condition occurs (when the read() method returns -1). While there is not an end-of-file condition, send the character to the ostream object.

Tip: You learned about `try` blocks in Chapter 15.

```
try
{
      while((c = istream.read()) != -1)
             ostream.write(c);
}
```

6. Use the following `catch` block to handle any IOException:

```
catch(IOException e)
{
       System.out.println("Error: " + e.getMessage());
}
```

7. Regardless of whether an IOException occurs, you want to close the streams. Use the following `finally` block to ensure that the streams are closed:

```
finally
{
       istream.close();
       ostream.close();
}
```

8. Add a closing curly brace for the main() method and another for the class.
9. Save the file as **ReadKBWriteScreen.java** in the Chapter.16 folder on your Student Disk, and then compile and run the program. At the command line, type any series of characters and then press **[Enter]**. As you type characters, the buffer holds them until you press [Enter], at which time the stored characters echo to the screen. Do not try to end the program yet.

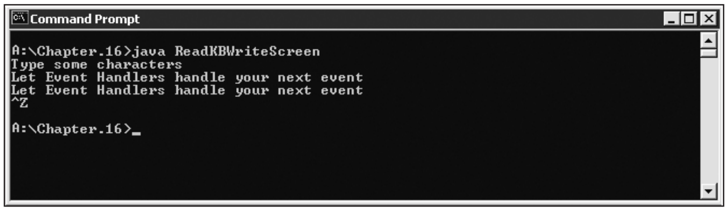
The `while` loop in the ReadKBWriteScreen program continues until the read() method returns -1. However, you cannot end the program by typing -1. Typing a minus sign (-) and a one (1) causes two additional characters to be sent to the buffer, and neither of those characters represents -1. Instead, you must press [Ctrl] + Z, which forces the read() method to return -1, and which the operating system recognizes as the end of the file. Next you will end the ReadKBWriteScreen program.

Tip

Pressing [Ctrl] + Z to end a program is an operating system command, not a Java command.

To end the ReadKBWriteScreen program:

1. At the command line, press **[Ctrl]+Z**, and then press **[Enter]**. The program ends. Figure 16-8 shows a typical program execution. Notice that the keystroke combination [Ctrl]+Z appears on the screen as ^Z.



```
A:\Chapter.16>java ReadKBWriteScreen
Type some characters
Let Event Handlers handle your next event
Let Event Handlers handle your next event
^Z

A:\Chapter.16>_
```

Figure 16-8 Typical execution of ReadKBWriteScreen program

2. Execute the program again. This time, type a line of characters, press **[Enter]**, and observe the echoed output. Then type another line, and press **[Enter]**. You can type as many lines as you want. Press **[Ctrl]+Z**, and then press **[Enter]**. The program will continue to echo your lines to the screen until you end the program.

Tip

Do not press [Ctrl]+C to end the program. Doing so breaks out of the program before its completion and does not properly close the files.

Writing to and Reading from a File

Instead of assigning files to the standard input and output devices, you can also assign a file to the InputStream or OutputStream. For example, you can read data from the keyboard and store it permanently on a disk. To accomplish this, you can construct a FileOutputStream object and assign it to the OutputStream. If you want to change a program's output device, you don't have to make any other changes to the program other than assigning a new object to the OutputStream; the rest of the program's logic remains the same. Java lets you assign a file to a stream object so that screen output and file output work in exactly the same manner.

You can associate a File object with the output stream in one of two ways:

- You can pass the filename to the constructor of the FileOutputStream class.

- You can create a File object passing the filename to the File constructor. Then you can pass the File object to the constructor of the FileOutputStream class.

The second method has some benefits—if you create a File object, you can use the File class methods such as exists() and lastModified() to retrieve file information. In the next set of steps you will use a FileOutputStream to write keyboard-entered data to a file you create.

Tip: Because applets are designed for distribution over the Internet, you are not allowed to use an applet to write to files on a client's workstation. Applets that write to a client's file could destroy a client's existing data.

To create a program that writes keyboard data to a file:

1. Save the ReadKBWriteScreen.java file as **ReadKBWriteFile.java** in the Chapter.16 folder on your Student Disk.
2. Change the class header to **`public class ReadKBWriteFile`**.
3. Position your insertion point at the end of the line that defines the ostream object (`OutputStream ostream;`), and then press **[Enter]** to start a new line. On the new line, define a File object as follows:

   ```
   File outFile = new File("datafile.dat");
   ```

4. Replace the statement that assigns System.out to the ostream object with the statement:

   ```
   ostream = new FileOutputStream(outFile);
   ```

5. Save the file, and then compile and execute the program. At the command line, type **Event Handlers handles events of all sizes**, and then press **[Enter]**. After you press [Enter], the characters will not appear on the screen; instead, they are output to a file named datafile.dat that is written in the default directory, the current directory from which you are working—in this case, the Chapter.16 directory.
6. Press **[Ctrl]+Z**, and then press **[Enter]** to stop the program.
7. In your text editor, open the **datafile.dat** file. The characters are an exact copy of the ones you entered at the keyboard.

You could enter any number of characters to the output stream before ending the program and they would be saved in the output file. If you run the ReadKBWriteFile program again, the program will overwrite the existing datafile.dat file with your new data.

Reading from a File

The process you use to read data from a file is similar to the one you use to write data to a file. You can assign a File object to the input stream, as you will do in the next steps.

To read data from a file:

1. In your text editor, open the **ReadKBWriteScreen.java** file you created earlier in this chapter, and then immediately save the file as **ReadFileWriteScreen.java** in the Chapter.16 folder of your Student Disk.
2. Change the class header to **`public class ReadFileWriteScreen`**.
3. Position your insertion point to the right of the statement that declares the OutputStream object named ostream, and then press **[Enter]** to start a new line. On the new line, enter the following code to create a File object to refer to the datafile.dat file you created:

   ```
   File inFile = new File("datafile.dat");
   ```
4. Change the statement that assigns the System.in object to istream `(istream = System.in)` so that you can use the File object for input instead of the keyboard by replacing it with the following:

   ```
   istream = new FileInputStream(inFile);
   ```
5. Remove the statement that prompts the user for input; **`System.out.println("Type some characters ");`**. A disk file does not need a prompt.
6. Save the file, and then compile and run the program. The data you stored in the datafile.dat file ("Event Handlers handles events of all sizes") appears on the screen, and the program ends.

Writing Formatted File Data

You do not usually want to read data files as a series of characters. For example, you might have a data file that contains personnel records that include an employee ID number, name, and salary for each employee in your organization. Rather than reading a series of bytes, it is more useful to be able to read such a file in groups of bytes that constitute an integer, a String, and a double. You can use the DataInputStream and DataOutputStream classes to accomplish formatted input and output.

DataOutputStream objects enable you to write binary data to an OutputStream. Much of the data that you write with DataOutputStream objects is not readable in a text editor because it is not stored as characters. Instead the data is formatted correctly for its type. For example, a double with the value 123.45 is not stored as six separate readable characters that can correctly display in a text editor. Instead, numeric values are stored in a more compact form that you can read later with a DataInputStream object.

The DataOutput interface is implemented by DataOutputStream. The DataOutput interface includes methods such as writeBoolean(), writeChar(), writeDouble(), writeFloat(), and writeInt(). Each method writes data in the correct format for the data type its name indicates. You can use the method writeUTF() to write Unicode format strings.

Tip

The meaning of the acronym UTF is disputed by various sources. The most popular interpretations include Unicode Transformation Format, Unicode Transfer Format, and Unicode Text Format.

When you create a DataOutputStream, you can assign a FileOutputStream object to it so that your data is stored in a file. Using DataOutputStream with a FileOutputStream allows you to use the correct write method that is appropriate for your data. When you use a DataOutputStream connected to FileOutputStream, this approach is known as **chaining the stream objects**. That is, if you define a DataOutputStream object with a statement such as `DataOutputStream out;`, then when you call the DataOutputStream constructor, you pass a FileOutputStream object to it (for example, `out = new DataOutputStream(new FileOutputStream("someFile"));`).

In the next series of steps, you will create a full-blown project for Event Handlers Incorporated like the one you saw in the Preview activity. The program uses a GUI interface to capture data about an event from a user, and writes that data to an output file using the DataOutput interface. The data required includes the host's name, the date, and the number of guests. For simplicity, this program accepts event dates for the current month only, so the date field is an integer. Figure 16-9 shows a preliminary sketch of the user's interface.

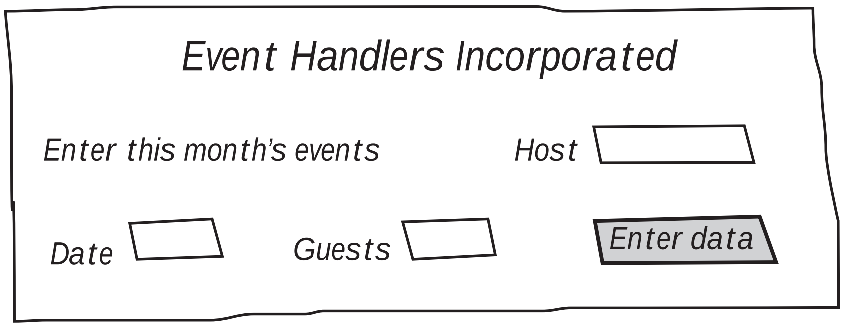


Figure 16-9 Sketch of the user's interface

To create a JFrame for data entry:

1. Open a new file in your text editor, and then type the following first few lines of the CreateEventFile class. CreateEventFile is a JFrame that reacts to a mouse click when you click an object within the JFrame. Therefore, you must implement ActionListener.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CreateEventFile extends JFrame
  implements ActionListener
{
```

2. Enter the following code to create a JLabel for the company name and a Font object to use with the company name:

```
private JLabel companyName =
   new JLabel("Event Handlers Incorporated");
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
```

3. Enter the following code to create a prompt that tells the user to enter data, and JTextFields for the host, date, and guests. Because a host's name is usually several characters long, the field for the host's name should be wider than the fields for the date and number of guests.

```
private JLabel prompt =
   new JLabel("Enter this month's events");
private JTextField host = new JTextField(10);
private JTextField date = new JTextField(4);
private JTextField guests = new JTextField(4);
```

4. Enter the following code to create a JLabel for each of the JTextFields. Include a JButton object that the user can click when a data record is completed and ready to be written to the data file. Then add a Container to hold the JFrame components.

```
private JLabel hLabel = new JLabel("Host");
private JLabel dLabel = new JLabel("Date");
private JLabel gLabel = new JLabel("Guests");
private JButton enterDataButton =
   new JButton("Enter data");
private Container con = getContentPane();
```

5. When you write the user's data to an output file, you will use the DataOutputStream class, so create a DataOutputStream object as follows:

```
DataOutputStream ostream;
```

6. Save the work you have done so far as **CreateEventFile.java** in the Chapter.16 folder on your Student Disk.

Next you will add the CreateEventFile's constructor to the class. The constructor calls its parent's constructor, which is the JFrame class constructor, and passes it a title to use for the JFrame. The constructor also attempts to open an events.dat file for output. If the open fails, the constructor's `catch` block handles the Exception; otherwise you add all the JTextFields, JLabel, and JButton Components to the JFrame.

To write the CreateEventFile class constructor:

1. In the CreateEventFile.java file, press **[Enter]** to start a new line below the statement that declares the DataOutputStream object, type the following constructor header and opening curly brace, and then call the superclass constructor:

```
public CreateEventFile()
{
   super("Create Event File");
```

2. Add the following `try...catch` block to handle the file creation:

```
try
{
   ostream = new DataOutputStream
   (new FileOutputStream("events.dat"));
}
catch(IOException e)
{
   System.err.println("File not opened");
   System.exit(1);
}
```

Tip: Notice the use of the System.err object to display an error message. Alternately, you can display the message on System.out.

3. After the file is open, use the following code to set the JFrame's size, choose a layout manager, and add all the necessary Components to the JFrame:

```
setSize(320,200);
con.setLayout(new FlowLayout());
companyName.setFont(bigFont);
con.add(companyName);
con.add(prompt);
con.add(hLabel);
con.add(host);
con.add(dLabel);
con.add(date);
con.add(gLabel);
con.add(guests);
con.add(enterDataButton);
```

4. To finish the JFrame constructor, enter the following code to register the JFrame as a listener for the JButton, make the JFrame visible, and set the default close operation for the JFrame. Finally, add a closing curly brace for the constructor.

```
   enterDataButton.addActionListener(this);
   setVisible(true);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

5. Save the file. Don't compile the file yet; you will add more code in the next set of steps.

When the users see the JFrame, they can enter data in each of the available JTextFields. When users complete a record for a single event, they click the JButton, which causes the actionPerformed() method to execute. This method must retrieve the text from each of the JTextFields and write it to a data file in the correct format. You will write a useable actionPerformed() method now.

To add the actionPerformed() method to the CreateEventFile program:

1. At the end of the existing code within the CreateEventFile.java file, press **[Enter]** to start a new line below the constructor method, and then type the following header for the actionPerformed() method. Within the method, create an integer variable that will hold the number of guests at an event.

```
public void actionPerformed(ActionEvent e1)
{
  int numGuests;
```

2. Use a `try` block to hold the data retrieval and the subsequent file-writing actions so that you can handle any I/O errors that occur. You will use the parseInt() method to convert the JTextField guest number to a usable integer, but you will accept the host and date fields as simple text. You can use the appropriate DataOutputStream methods to write formatted data to the output file.

Tip: You first learned about parseInt() and the Integer class in Chapter 7.

```
try
{
   numGuests = Integer.parseInt(guests.getText());
   ostream.writeUTF(host.getText());
   ostream.writeUTF(date.getText());
   ostream.writeInt(numGuests);
```

3. Continue the `try` block by removing the data from each JTextField after it is written to the file. That way, each JTextField will be clear and ready to receive data for the next record. Notice that to clear the fields, you use a pair of quotes with no space between them. Then end the `try` block.

```
 host.setText("");
 date.setText("");
 guests.setText("");
}
```

4. There are two types of Exceptions that you might want to deal with in this application. Because the host name and date fields are text, the user can enter any type of data. However, the guest field must be an integer. When you use the parseInt() method with data that cannot be converted to an integer (such

as alphabetic letters), a NumberFormatException error occurs. In this case, you can write an error message to the standard error device and explain the problem as follows:

```
catch(NumberFormatException e2)
{
   System.err.println("Invalid number of guests");
}
```

5. A second and more serious Exception occurs when the program cannot write the output file, so you should `catch` the potential IOException, print an error message, and exit using the following code:

```
catch(IOException e3)
{
   System.err.println("Error writing file");
   System.exit(1);
}
```

6. Add two closing curly braces—one for the actionPerformed() method and one for the class. Then save and compile the file.

Next you will create a program that uses the CreateEventFile JFrame. The program's only task is to instantiate a CreateEventFile JFrame object.

To write a program that uses a CreateEventFile JFrame:

1. Open a new file in your text editor, and then type the following EventFile program that establishes a CreateEventFile object:

```
public class EventFile
{
  public static void main(String[] args)
  {
     CreateEventFile cef = new CreateEventFile();
  }
}
```

2. Save the file as **EventFile.java** in the Chapter.16 folder on your Student Disk, and then compile it using the **`javac`** command. When it compiles successfully, run the program, which should look like Figure 16-10.

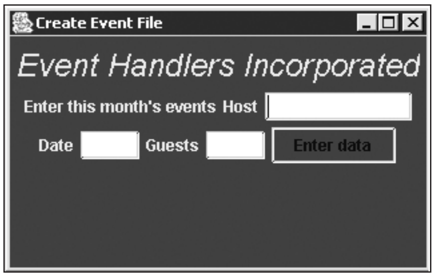


Figure 16-10 User interface for the EventFile program

3. Type sample data into the JTextFields in the JFrame. Specifically, type **Sagami** as the event host, on the **3**(rd), with **150** guests. After entering the data into the three data fields, click the **Enter data** button. Your data is sent to the file, and the fields are cleared. Now you can enter a second record (make up your own record information), and then click the **Enter data** button again. Repeat this process until you have entered five data records. While entering at least one record, type non-numeric data in the guest field. Notice the error message that displays on the standard error device at the command line.
4. Click the **Close** button in the CreateEventFile JFrame to close it.
5. Examine your Student Disk using any file-management program or the DOS command-line directory command, dir. Confirm that your program created the events.dat data file in the Chapter.16 folder on your Student Disk. You will write a program to read the file in the next series of steps.

Reading Formatted File Data

DataInputStream objects enable you to read binary data from an InputStream. The DataInput interface is implemented by DataInputStream. The DataInput interface includes methods such as readByte(), readChar(), readDouble(), readFloat(), readInt(), and readUTF(). In the same way that the different write() methods of DataOutput correctly format data you write to a file, each DataInput read() method correctly reads the type of data indicated by its name.

When you want to create a DataInputStream object that reads from a file, you use the same chaining technique you used for output files. In other words, if you define a DataInputStream object as `DataInputStream in;`, then you can associate it with a file when you call its constructor, as in `in = new DataInputStream (FileInputStream("someFile"));`.

When you read data from a file, you need to determine when the end of the file has been reached. Earlier in this chapter, you learned that you can determine EOF by checking for a return value of -1 from the read() method. Alternately, if you attempt each file read() from within a `try` block, you can catch an EOFException. When you catch an EOFException, it means you have reached the end of the file and you should take appropriate action, such as closing the file.

Tip

Most Exceptions represent error conditions. An EOFException is more truly an "exception" in that most read() method calls do not result in EOF. For example, when a file contains 999 records, only the 1,000th, or last, read() for a file causes an EOFException.

Next you will create a JFrame in which employees of Event Handlers Incorporated can view each individual record stored in the events.dat file. The user interface will look like the interface used in the CreateEventFile JFrame, but the user will not enter data within this JFrame. Instead, the user will click a JButton to see each succeeding record in the event.dat file.

To create a JFrame for viewing file data:

1. Open a new file in your text editor, and then type the following first few lines of the ReadEventFile class:

   ```
   import java.io.*;
   import java.awt.*;
   import java.awt.event.*;
   import javax.swing.*;
   public class ReadEventFile extends JFrame
      implements ActionListener
   {
   ```

2. Enter the following code to declare all of the JLabels, JTextFields, and associated values that will appear in the JFrame. The text of the prompt and JButton have changed, but these statements basically echo the statements in the CreateEventFile.java file.

   ```
   private JLabel companyName =
      new JLabel("Event Handlers Incorporated");
   Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
   private JLabel prompt =  new
      JLabel("View this month's events");
   private JTextField host = new JTextField(10);
   private JTextField date = new JTextField(4);
   private JTextField guests = new JTextField(4);
   private JButton viewEventButton = new
      JButton("View Event");
   private JLabel hLabel = new JLabel("Host");
   private JLabel dLabel = new JLabel("Date");
   private JLabel gLabel = new JLabel("Guests");
   private Container con = getContentPane();
   ```

3. Enter the following code to declare a DataInputStream object. Then write the ReadEventFile constructor method that uses a `try...catch` block to open a file. Notice that you can chain the DataInputStream object and a FileInputStream object using the same technique you used for output. (*Note:* If you have stored your event.dat file in a location other than A:\Chapter.16, then change the location in the FileInputStream constructor in your own program.)

   ```
   DataInputStream istream;
   public ReadEventFile()
   {
      super("Read Event File");
      try
      {
   ```

```
            istream = new DataInputStream
              (new FileInputStream
              ("A:\\Chapter.16\\events.dat"));
      }
      catch(IOException e)
      {
             System.err.println("File not opened");
             System.exit(1);
      }
```

4. After successfully opening the file, set the JFrame size, layout manager, and Font for the JFrame as follows:

```
setSize(325,200);
con.setLayout(new FlowLayout());
companyName.setFont(bigFont);
```

5. Add the JFrame's Components as follows:

```
con.add(companyName);
con.add(prompt);
con.add(hLabel);
con.add(host);
con.add(dLabel);
con.add(date);
con.add(gLabel);
con.add(guests);
con.add(viewEventButton);
```

6. Enter the following code to ensure that the JFrame listens for JButton messages, to make the JFrame visible, and to set the default close operation:

```
viewEventButton.addActionListener(this);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

7. Add the closing curly brace for the ReadEventFile constructor.

8. Type the beginning of the following actionPerformed() method. This method declares variables for the file field data, and then uses a `try` block to call the appropriate read() method for each field. Each data field then appears in the correct JTextField.

```
public void actionPerformed(ActionEvent e1)
{
String theHost, theDate;
int numGuests;
try
{
    theHost = istream.readUTF();
    theDate = istream.readUTF();
    numGuests = istream.readInt();
```

```
        host.setText(theHost);
        date.setText(theDate);
        guests.setText(String.valueOf(numGuests));
    }
```

9. Code the following two `catch` blocks for the `try` block that reads the data fields. The first `catch` block catches the EOFException and calls a closeFile() method. The second `catch` block catches IOExceptions and exits the program if there is a problem with the file. Notice that the Exceptions have unique names (e2 and e3) because you cannot declare two data items with the same name within the same method.

```
catch(EOFException e2)
{
   closeFile();
}
catch(IOException e3)
{
   System.err.println("Error reading file");
   System.exit(1);
}
```

10. Add the closing curly brace for the actionPerformed() method.
11. Write the following closeFile() method that closes the DataInputStream object and exits the program:

```
public void closeFile()
{
   try
   {
          istream.close();
          System.exit(0);
   }
   catch(IOException e)
   {
          System.err.println("Error closing file");
          System.exit(1);
   }
}
```

12. Add the closing curly brace for the class.
13. Save the file as **ReadEventFile.java** in the Chapter.16 folder on your Student Disk, and then compile the program using the `javac` command.

Next you will write a short host program that creates a ReadEventFile JFrame so Event Handlers employees can examine the existing file of scheduled events.

To write a program that creates a JFrame and displays file data:

1. Open a new file in your text editor, and then type the following program that instantiates a ReadEventFile object:

```
import java.io.*;
public class EventFile2
{
  public static void main(String[] args)
  {
     ReadEventFile ref = new ReadEventFile();
  }
}
```

2. Save the file as **EventFile2.java** in the Chapter.16 folder on your Student Disk, compile and execute the program, and then click the **View Event** button. Figure 16-11 shows the interface with the first record from the events.dat data file visible. (Your data might differ.) Click the **View Event** button again to see the second record. Continue clicking the **View Event** button until you reach the end of the file; when you do, the JFrame closes and the program ends.

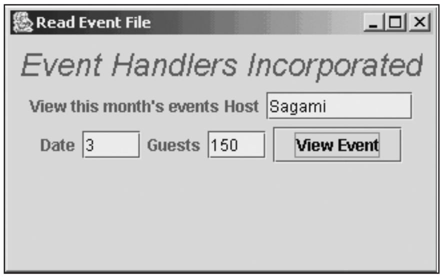


Figure 16-11 Output of the EventFile2 program

USING A VARIABLE FILENAME

A program that reads a data file and displays its contents for you is useful. A program that can read any data file, regardless of what you name it, is even more useful. Suppose Event Handlers Incorporated keeps different event files for each month. When you execute the EventFile2 program, it would be convenient to name the file you want to display. For example, your command line might be `java EventFile2 eventsApril.dat` or `java EventFile2 October.dat`. The same program would execute, but use the data file that corresponds to the requested month. Next you will modify the EventFile2 program to accept a variable filename from the user.

To modify the EventFile2 program to use a variable filename:

1. Open the **ReadEventFile.java** file and immediately save it as **ReadNamedFile.java**. Be sure to change the class name to match.

2. Change the ReadEventFile constructor to the new class name and add an argument that is a String. The value for this argument will originate from the first element in the String array (`String[] args`) that you have included in every main() method you have written.

   ```
   public ReadNamedFile(String fileName)
   ```

3. Within the `try` block of the ReadNamedFile constructor, remove the reference to the filename events.dat, and replace it with the variable filename that represents the String you will use when you call this constructor. In other words, change the istream assignment to the following:

   ```
   istream = new DataInputStream
       (new FileInputStream(fileName));
   ```

4. Save the file, and then compile the class.

5. Open a new file in your text editor and type the following program that creates an instance of the ReadNamedFile class. As the argument to the ReadNamedFile constructor, use the first (and only) String in the array of String arguments that you will pass to the main() method from the command line.

   ```
   import java.io.*;
   public class EventFile3
   {
      public static void main(String[] args)
      {
             ReadNamedFile ref = new ReadNamedFile(args[0]);
      }
   }
   ```

6. Save the file as **EventFile3.java** in the Chapter.16 folder on your Student Disk, and compile the program.

7. You will test the program using the existing file event.dat. To execute the program, at the command line type **`java EventFile3 events.dat`**. The program executes, correctly showing you the events in the file as before. One at a time, view each of the records in the file just like you did in the last set of steps.

8. Attempt to execute the program using a nonexisting filename, for example **`java EventFile3 noSuchFile.dat`**. You should see the message "File not opened".

CREATING RANDOM ACCESS FILES

The files you wrote to and read from in this chapter are **sequential access files**, which means that you access the records in sequential order from beginning to end. For example, if you wrote an Event record with host name Adams, and then you created an Event record with host name Brown, when you retrieve the records you see that they remain in the original data-entry order. Businesses store data in sequential order when they use the records for **batch processing**, or processing that involves performing the same tasks with many records, one after the other. For example, when a company produces paychecks, the records for the pay period are gathered in a batch and the checks are calculated and printed in sequence.

For many applications, sequential access is inefficient. These applications, known as **real-time** applications, require that a record be accessed immediately while a client is waiting. For example, if a customer telephones a department store with a question about a monthly bill, the customer service representative does not need to access every customer account in sequence. With tens of thousands of account records to read, it would take too long to access the customer's record. Instead, customer service representatives require **random access files**, files in which records can be located in any order. Because they enable you to locate a particular record directly (without reading all of the preceding records), random access files are also called **direct access files**. You can use Java's RandomAccessFile class to create your own random access files.

The RandomAccessFile class contains the same read(), write(), and close() methods as InputStream and OutputStream, but it also contains a seek() method that lets you select a beginning position within a file before you read or write data. For example, if you declare a RandomAccessFile object named myFile, then the statement `myFile.seek(200);` selects the 200th position within the file. The 200th position represents the 201st byte because, as with Java arrays, the numbering of file positions begins at zero. The next read() or write() method will operate from the newly selected starting point.

When you store records in a file, it is often more useful to be able to access the 200th record, rather than the 201st byte. In this case, you multiply each record's size by the position you want to access. For example, if you store records that are 50 bytes long, you can access the nth record using the statement `myFile.seek((n-1) * 50);`.

When you declare a RandomAccessFile object, you include a filename as you do with other file objects. You also include r or rw within double quotation marks as a second argument to indicate that the file is open for reading only ("r"), or for both reading and writing ("rw"). For example, `RandomAccessFile myFile = new RandomAccessFile("C:\\Temp\\someData.dat","rw");` opens the someData.dat file so that either the read() or write() method can be used on the file. This feature is particularly useful in random access processing. Consider a business with 20,000 customer accounts. When the customer who has the 14,607th record in the file acquires a new telephone number, it is convenient to directly access the 14,607th record, the read() method confirms it represents the correct customer, and then the write() method writes the new telephone number to the file in the location the old telephone number was stored previously. You will demonstrate the seek() method in the next set of steps.

To show how the seek() method works:

1. Open a new file in your text editor, and then type the following first few lines of the AccessRandomly class:

   ```
   import java.io.*;
   public class AccessRandomly
   {
        public static void main(String[] args) throws
        IOException
        {
   ```

2. Enter the following code to declare an OutputStream object for output, an integer to hold data temporarily, and a new RandomAccessFile. You will use the data file for reading only, so you include "r" as the second argument in the RandomAccessFile constructor.

   ```
   OutputStream ostream;
   int c;
   RandomAccessFile inFile =
     new RandomAccessFile("datafile.dat","r");
   ```

3. Enter the following code to assign ostream as the standard output device. Then try accessing the 10th file position (which is represented by the number 9). Read this position, and then display its contents.

   ```
   ostream = System.out;
   try
   {
      inFile.seek(9);
      c = inFile.read();
      System.out.print("The character in position 9 is ");
      ostream.write(c);
   }
   ```

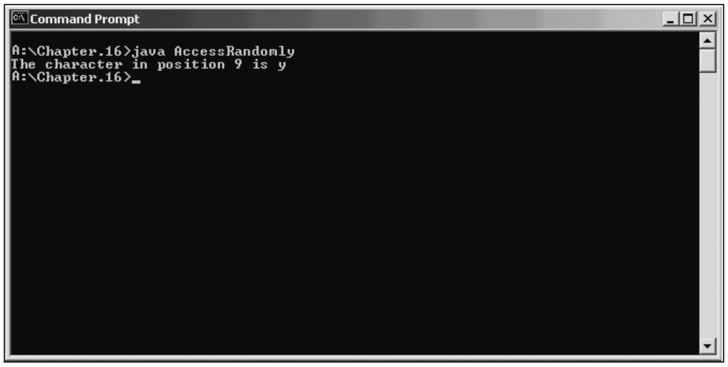
4. Add the following `catch` clause that executes if there is trouble accessing the file, and then add a `finally` block to close the files:

   ```
   catch(IOException e)
   {
      System.out.println("Error: " + e.getMessage());
   }
   finally
   {
      inFile.close();
      ostream.close();
   }
   ```

5. Add two closing curly braces—one for the main() method and one for the class.

6. Save the file as **AccessRandomly.java** in the Chapter.16 folder on your Student Disk, and then compile it using the `javac` command.

7. Before you run the file, create a datafile.dat file by running the ReadKBWriteFile program you created earlier in this chapter. At the command line, type **`java ReadKBWriteFile`**, and then press **[Enter]**.
8. At the prompt, type **This is my random access file**. Press **[Enter]**, press **[Ctrl]+Z**, and then press **[Enter]** to end the program. (The *y* in *my* is the 10th character you type.)
9. At the command line, run the AccessRandomly program by typing **`java AccessRandomly`**. The output is the tenth character in the file, as shown in Figure 16-12.



```
A:\Chapter.16>java AccessRandomly
The character in position 9 is y
A:\Chapter.16>_
```

Figure 16-12 Output of the AccessRandomly program

In the AccessRandomly program, only one read() command was issued, yet the program accessed a byte nine positions into the file. When you access a file randomly, you do not read all the data that precedes the data you are seeking. Accessing data randomly is one of the major features that makes large data systems maintainable.

Chapter Summary

- Files are objects that you store on permanent storage devices, such as floppy disks, CD-ROMs, or external drives. You can use the File class to gather file information.
- Data used by businesses generally is stored in a data hierarchy that includes files, records, fields, and characters.
- Java views a file as a series of bytes, and a stream as an object through which input and output data (in the form of bytes) flow. InputStream and OutputStream are abstract subclasses of Object that contain methods for performing input and output. FileInputStream and FileOutputStream provide the capability to read from and write to files. You can use the InputStream read()method to read in one character at a time.
- You can use the DataInputStream and DataOutputStream classes to accomplish formatted input and output. The DataOutput interface includes methods such as

writeBoolean(), writeChar(), writeDouble(), writeFloat(), and writeInt(). Each method writes data in the correct format for the data type its name indicates. You can use the method writeUTF() to write Unicode format strings.

- DataInputStream objects enable you to read binary data from an InputStream. The DataInput interface includes methods such as readByte(), readChar(), readDouble(), readFloat(), readInt(), and readUTF(). Each DataInput read() method correctly reads the type of data indicated by its name, such as readByte(), readChar(), or readDouble().
- You can provide a variable filename to a program using the command line.
- Random access files, or direct access files, are files in which records can be located in any order. The RandomAccessFile class contains the same read(), write(), and close() methods as InputStream and OutputStream, but it also contains a seek() method that lets you select a beginning position within a file before you read or write.

Review Questions

1. Files always ________________.
   a. hold software instructions
   b. occupy a section of storage space
   c. remain open during the execution of a program
   d. all of the above
2. The File class enables you to ________________.
   a. open a file
   b. close a file
   c. determine a file's size
   d. all of the above
3. The ________________ package contains all the classes you use in file processing.
   a. java.file
   b. java.io
   c. java.lang
   d. java.process
4. The statement `File aFile = new File("myFile");` creates a file ________________.
   a. on the disk in drive A
   b. on the hard drive (drive C)
   c. in the Temp folder on the hard drive (drive C)
   d. on the default disk drive

5. The File method canWrite() returns a(n) ________________ value.
   a. int
   b. Boolean
   c. Object
   d. `void`
6. Data used by businesses is stored in a data hierarchy that includes the following items, from largest to smallest:
   a. file, field, record, character
   b. record, file, field, character
   c. file, record, field, character
   d. record, field, file, character
7. A group of characters that has meaning is a ________________.
   a. file
   b. record
   c. field
   d. byte
8. Files consist of related ________________.
   a. records
   b. fields
   c. data segments
   d. archives
9. Before a program can use a data file, the program must ________________ the file.
   a. create
   b. open
   c. store
   d. close
10. When you perform an input operation in a Java program, you use a ________________.
    a. pipeline
    b. channel
    c. moderator
    d. stream

11. Most streams flow ________________.
    a. in
    b. out
    c. either in or out, but only in one direction
    d. both in and out concurrently
12. The output from System.err and System.out ________________ go to the same device.
    a. must
    b. cannot
    c. might
    d. might on a mainframe system, but never would on a PC
13. A small memory location that is used to temporarily hold data is a ________________.
    a. stream
    b. buffer
    c. bulwark
    d. channel
14. The read() method returns a value of -1 when it encounters a(n) ________________.
    a. input error
    b. integer
    c. end-of-file condition
    d. negative value
15. Much of the data that you write with DataOutputStream objects is not readable in a text editor because ________________.
    a. it does not exist in any physical sense
    b. it is stored in a non-character format
    c. you can read it only with a special piece of hardware called a Data Reader
    d. Java's security features prohibit it
16. You use a DataOutputStream connected to FileOutputStream by using a method known as ________________.
    a. sequencing
    b. iteration
    c. piggybacking
    d. chaining

17. When you `catch` an EOFException, it means you have ________________.
    a. failed to find the end of the file
    b. forgotten to open a file
    c. forgotten to close a file
    d. reached the end of a file
18. Which of the following applications is most likely to use random file processing?
    a. a program that schedules airline reservations
    b. a credit card company's end-of-month billing program
    c. a college's program that lists honor students at the end of each semester
    d. a manufacturing company's quarterly inventory reporting system
19. The method that the RandomAccessFile class contains that does not exist in the InputStream class is ________________.
    a. read()
    b. close()
    c. seek()
    d. delete()
20. You can open a RandomAccessFile object for ________________.
    a. reading
    b. writing
    c. both of the above
    d. none of the above

EXERCISES

1. Create a file using any word-processing program or text editor. Write a program that displays the file's name, parent, size, and time of last modification. Save the program as **FileStatistics.java** in the Chapter.16 folder on your Student Disk.
2. Create two files using any word-processing program or text editor. Write a program that determines whether the two files are located in the same folder. Save the program as **SameFolder.java** in the Chapter.16 folder on your Student Disk.
3. Write a program that determines which, if any, of the following files are stored in the Chapter.16 folder of your Student Disk: autoexec.bat, SameFolder.java, Chap16ReadEventFile.class, and Hello.java. Save the program as **FindSelectedFiles.java** in the Chapter.16 folder on your Student Disk.
4. a. Create a JFrame that allows the user to enter a series of friends' names and phone numbers and creates a file from the entered data. Save the JFrame as **CreatePhoneList.java** in the Chapter.16 folder on your Student Disk. To use

the JFrame, create a program named **UsePhoneList.java** that instantiates a CreatePhoneList object.

b. Write a program that reads the file created by the UsePhoneList program and displays one record at a time in a JFrame. Save the JFrame as **ReadPhoneList.java** in the Chapter.16 folder on your Student Disk. Save the program that uses this JFrame as **UsePhoneList2.java**.

5. a. Write a program for a mail-order company. The program uses a data-entry screen in which the user types an item number and a quantity. Write each record to a file. Save the program as **MailOrderWrite.java** in the Chapter.16 folder on your Student Disk.

   b. Write a program that reads the data file created by the MailOrderWrite program and displays one record at a time on the screen. Save the program as **MailOrderRead.java** in the Chapter.16 folder on your Student Disk.

6. a. Write a program for a mail-order company. The program uses a data-entry screen in which the user types an item number and a quantity. The valid item numbers and prices are as follows:

Item Number	Price ($)
101	4.59
103	29.95
107	36.50
125	49.99

   When the user enters an item number, check the number to make sure that it is valid. If it is valid, write a record that includes item number, quantity, price each, and total price. Save the program as **MailOrderWrite2.java** in the Chapter.16 folder on your Student Disk.

   b. Write a program that reads the data file created by the MailOrderWrite2 program and displays one record at a time on the screen. Save the program as **MailOrderRead2.java** in the Chapter.16 folder on your Student Disk.

7. a. Write a program that allows a user to enter an integer representing a file position. Access the requested position within a file and display the character there. Save the program as **SeekPosition.java** in the Chapter.16 folder on your Student Disk.

   b. Modify the program created by the SeekPosition program so that you display the next five characters after the requested position. Save the program as **Seek2.java** in the Chapter.16 folder on your Student Disk.

8. a. Write a program that creates a JFrame with text fields for order processing for a tee-shirt manufacturer. Include JTextFields for size, color, and slogan. Write each complete record to a file. Save the JFrame as **TeeShirtWrite.java** in the Chapter.16 folder on your Student Disk. Write a program named **UseTeeShirt.java** that displays the JFrame.

b. Write a program that reads the data file created by the TeeShirtWrite program and displays one record at a time in a JFrame on the screen. Save the program as **TeeShirtRead.java** in the Chapter.16 folder on your Student Disk. Write a program named **UseTeeShirt2.java** that displays the JFrame.

9. Write a program that allows the user to type any number of characters into a file. Then display the file contents backward. Save the program as **ReadBackwards.java** in the Chapter.16 folder on your Student Disk.

10. a. Create a JFrame that allows you to enter student data—ID number, last name, and first name. Include two buttons and instruct the user to click "Grad" or "Undergrad" after entering the data for each student. Depending on the user's choice, write the data to either a file containing graduate students or a separate file containing undergraduate students. Name the JFrame **GradAndUndergrad.java** and create a program named **UseGradAndUndergrad.java** that instantiates a JFrame object. Save both files in the Chapter.16 folder of your Student Disk.

    b. Create a JFrame named **StudentRead.java** that, in turn, accesses all the records in the graduate and the undergraduate files created in the GradAndUndergrad program. Write a program named **UseStudentRead.java** that instantiates a JFrame object. Save both files in the Chapter.16 folder of your Student Disk.

11. Each of the following files in the Chapter.16 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. Notice that DebugSixteen3.java and DebugSixteen4.java have companion files that are part of each exercise. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugSixteen1.java will become FixDebugSixteen1.java.

    a. DebugSixteen1.java

    b. DebugSixteen2.java

    c. DebugSixteen3.java

    d. DebugSixteen4.java

CASE PROJECT

Create a data-entry and retrieval system for Mowers Inc., a lawn-mowing service. Use a JFrame to enter data for the customer's name and lawn size in square feet. An output file holds the customer name, lawn size, and fee per mowing—$50 for lawns under 1,000 square feet and $75 for lawns 1,000 square feet or more. Name the JFrame **MowersInc.java** and save it in the Chapter.16 folder of your Student Disk. Name the client program that creates an instance of the JFrame **UseMowersInc.java**. Retrieve the created records with a JFrame named **MowersInc2.java** and a client program named **UseMowersInc2.java**.

CHAPTER

17 MULTITHREADING AND ANIMATION

In this chapter, you will:

- Understand multithreading
- Learn about a Thread's life cycle
- Use the Thread class
- Use the sleep() method
- Set Thread priority
- Use the Runnable interface
- Create an animated figure
- Reduce flickering
- Use pre-drawn animated Image objects
- Understand garbage collection
- Put animation in a Web browser page

I thought I had learned a lot about Java," you tell Lynn Greenbrier during your six-month performance review at Event Handlers Incorporated. "I've learned to write applets and applications, extend classes, and write data files. I can even use sound and images in my applets."

"You've come very far," Lynn agrees. "I'm proud of you!"

"But there's something missing. I want to create applets with moving figures for Event Handlers. I want programs in which different activities take place on the screen at the same time."

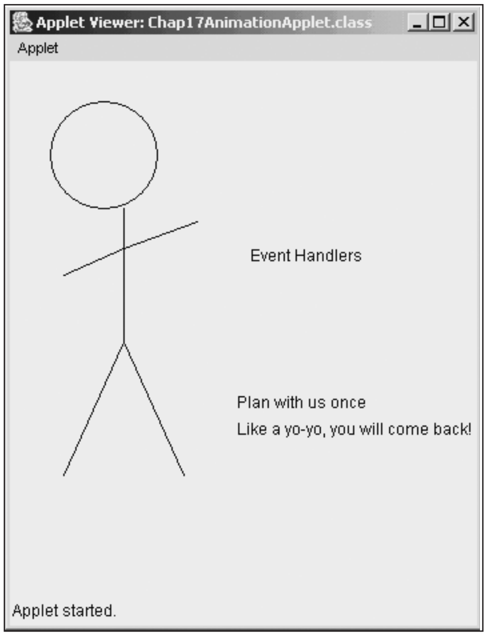
"You almost know everything you need to accomplish your goals," Lynn explains. "You know about applets, exceptions, inheritance, and graphics. All you have to do is put it all together. Let me tell you a little about threads, and then you can get started on animation."

PREVIEWING A PROGRAM THAT DISPLAYS ANIMATION

Event Handlers Incorporated wants an animated stick figure to appear on its Web site. The figure appears to be using the Event Handlers company name as a yo-yo; that is, the words "Event Handlers" move up and down as if controlled by the moving arm of the stick figure. You will create a similar program in this chapter, but now you can use a completed version of the Chap17AnimationApplet applet that is saved in the Chapter.17 folder on your Student Disk.

To use the Chap17AnimationApplet class:

1. Go to the command prompt for the Chapter.17 folder on your Student Disk, type **`appletviewer Chap17Animation.html`**, and then press **[Enter]**. You will see a moving figure similar to the one captured in Figure 17-1.



Event Handlers

Plan with us once
Like a yo-yo, you will come back!

Applet started.

Figure 17-1 Chap17AnimationApplet applet

2. When you are finished viewing the applet, click the Applet Viewer's **Close** button.

Understanding Multithreading

A **thread** is the flow of execution of one set of program statements. When you execute a program statement by statement, from beginning to end, you are following a thread. Each of the programs you have written while working through this book has had a single thread; this means that at any one time, Java was executing only a single program statement.

Single-thread programs contain statements that execute in very rapid sequence, but only one statement executes at a time. When a computer contains a single central processing unit (CPU, or processor), it can execute only one computer instruction at a time, regardless of its processor speed. When you use a computer with multiple CPUs, the computer can execute multiple instructions simultaneously.

The Java programming language allows you to launch, or start, multiple threads, no matter which type of processor you use. Using multiple threads of execution is known as **multithreading**. As already noted, if you use a computer system that contains more than one CPU (such as a very large mainframe or supercomputer), multiple threads can execute simultaneously. Figure 17-2 illustrates how multithreading executes in a multiprocessor system.

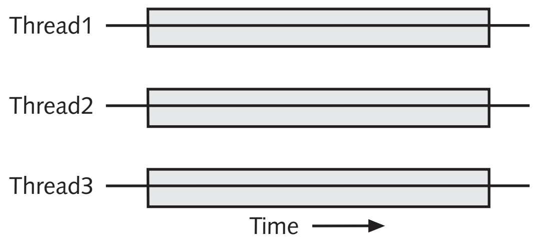


Figure 17-2 Executing multiple threads in a multiprocessor system

If you use a computer with a single processor, the multiple threads share the CPU's time, as shown in Figure 17-3. The CPU devotes a small amount of time to one task, and then devotes a small amount of time to another task. The CPU never actually performs two tasks at the same instant. Instead, it performs a piece of one task, and then a piece of another task. The CPU performs so quickly that each task seems to execute without interruption.

Perhaps you have seen an expert chess player participate in chess games with several opponents at once. The chess player makes a move on the first playing board, and then moves to the second board against a second opponent, while the first opponent analyzes his next move. The master can move to the third board, make a move, and return to the first board before the first opponent is even ready to respond. To the first opponent, it might seem as though the expert player is devoting all of her time to him. Because the expert is so fast, she can play other opponents in the first opponent's "downtime". Executing multiple threads on a single CPU works in a similar way. The CPU transfers its attention from thread to thread so quickly that the tasks don't even "miss" the CPU's attention.

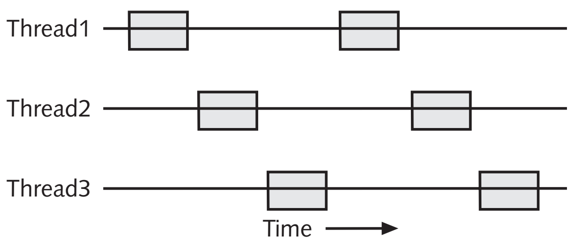


Figure 17-3 Executing multiple threads in a single-processor system

You use multithreading to improve the performance of your programs. Multithreaded programs often run faster, but what is more important is that they are more user-friendly. With a multithreaded program, your user can continue to click buttons while your program is reading a data file. With multithreading, an animated figure can display on one part of the screen while the user makes menu selections on another part of the screen. When you use the Internet, multithreading increases in importance. For example, you can begin to read a long text file or listen to an audio file while they are still downloading. Web users are likely to abandon a site if downloading a file takes too long. When you use multithreading to perform concurrent tasks, you are more likely to retain visitors to your Web site—this is particularly important if your site sells a product or service.

Tip

Programmers sometimes use the terms "thread of execution" or "execution context" to describe a thread. They also refer to a thread as a lightweight process because it is not a full-blown program. Rather, a thread must run within the context of a full, heavyweight program.

Learning About a Thread's Life Cycle

A Thread can be in one of five states during its life: new, ready, running, inactive, or finished. When you create a Thread, it is in the new state; the only method you can use with a new Thread is the method to start it. When you call the Thread's start() method, the Thread enters the ready state. A ready Thread is **runnable**, which means that it can run. However, a runnable Thread might not be in the running state because the CPU might be busy elsewhere. Just as your runnable automobile cannot pass through an intersection until the traffic officer waves you on, a runnable Thread cannot run until the CPU allocates some time to it.

Tip

If you call a Thread method that the Thread's present state does not allow, Java automatically `throws` an IllegalThreadStateException. For example, you cannot call the stop() method for a Thread that is new and has not been started.

When a Thread begins to execute, it is in the running state. A Thread runs until it becomes inactive or finishes. A Thread enters the inactive state when you call the

Thread's sleep() or suspend() method, or it might become inactive if it must wait for another Thread to finish and for the CPU to have available time. (You will see the sleep() method at work when you write the DemoHelloGoodbyeThreads program later in this chapter.) When a Thread completes the execution of its run() method, it is in the finished or dead state. A Thread can also enter the finished state before its run() method completes if you call the Thread's stop() method. You can use the isAlive() method to determine whether a Thread is currently alive, which means that it has started but has not stopped. The isAlive() method returns the Boolean value `false` if a Thread is new or finished, and `true` if it is in any other state.

Table 17-1 describes several useful methods of the Thread class.

Table 17-1 Selected Thread class methods

Method	Description
start()	Starts the Thread causing the run() method to execute
stop()	Stops the Thread
suspend()	Suspends the Thread until you use the resume() method
resume()	Resumes the Thread you suspended
isAlive()	Returns `true` or `false` to indicate whether the Thread is currently running
setPriority(int)	Lets you set a priority from 1 to 10 for a Thread by passing an integer (You will learn about Thread priorities later in this chapter.)
sleep(int)	Lets you pause thread execution for a specified number of milliseconds

Using the Thread Class

Technically, every program you have created is a thread. You can also create a thread by extending the Thread class, which is defined in the java.lang package. The Thread class contains a method named run(). You override the run() method in your extended class to tell the system how to execute the Thread. For example, you can write the HelloThread class shown in Figure 17-4, which prints a "Hello" message to the screen 100 times. The HelloThread class contains a single method—the run() method—which prints a space, "Hello", and another space in a loop that executes 100 times.

```
class HelloThread extends Thread
{
        public void run()
        {
               for(int x=0; x<100; ++x)
                       System.out.print(" Hello ");
        }
}
```

Figure 17-4 The HelloThread class

When you create a class that extends Thread, you inherit the start() method. You use the start() method with an instantiated Thread object; it tells the system to start execution of the Thread. For example, you can write a program that instantiates and starts a HelloThread object, as shown in Figure 17-5.

```
class DemoHelloThread
{
        public static void main(String[] args)
        {
               HelloThread hello = new HelloThread();
               hello.start();
        }
}
```

Figure 17-5 The DemoHelloThread class

The DemoHelloThread class instantiates a HelloThread object named hello. When you use the start() method with the hello object, the run() method within the HelloThread class executes. The output appears in Figure 17-6; the "Hello" message prints 100 times.

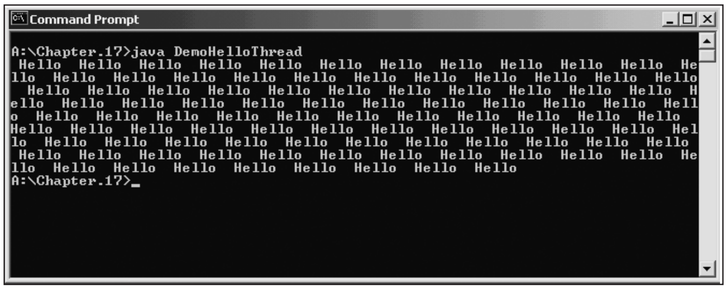

```
A:\Chapter.17>java DemoHelloThread
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  H
ello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hell
o  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hel
lo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
A:\Chapter.17>_
```

Figure 17-6 Output of the DemoHelloThread program

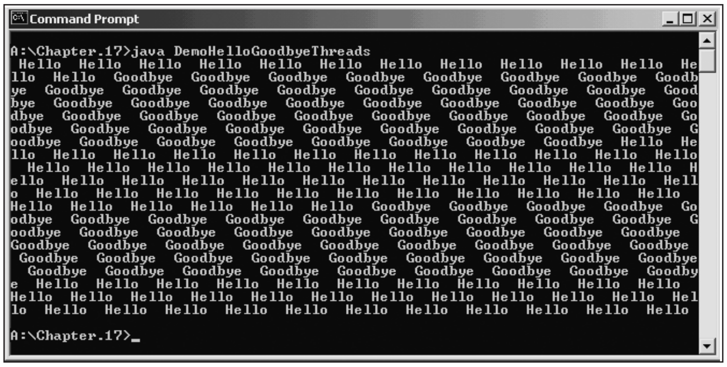
You can achieve multithreading by starting more than one Thread object. For example, consider the GoodbyeThread class in Figure 17-7; it is identical to the HelloThread class except for the message it prints. When you create a demonstration class like DemoHelloGoodbyeThreads (see Figure 17-8), and instantiate one HelloThread and one GoodbyeThread object within its main() method, then the output might appear as shown in Figure 17-9.

```
class GoodbyeThread extends Thread
{
       public void run()
       {
              for(int x=0; x<100; ++x)
              System.out.print(" Goodbye ");
       }
}
```

Figure 17-7 GoodbyeThread class

```
class DemoHelloGoodbyeThreads
{
       public static void main(String[] args)
       {
               HelloThread hello = new HelloThread();
               GoodbyeThread goodbye = new GoodbyeThread();
               hello.start();
               goodbye.start();
       }
}
```

Figure 17-8 DemoHelloGoodbyeThread class

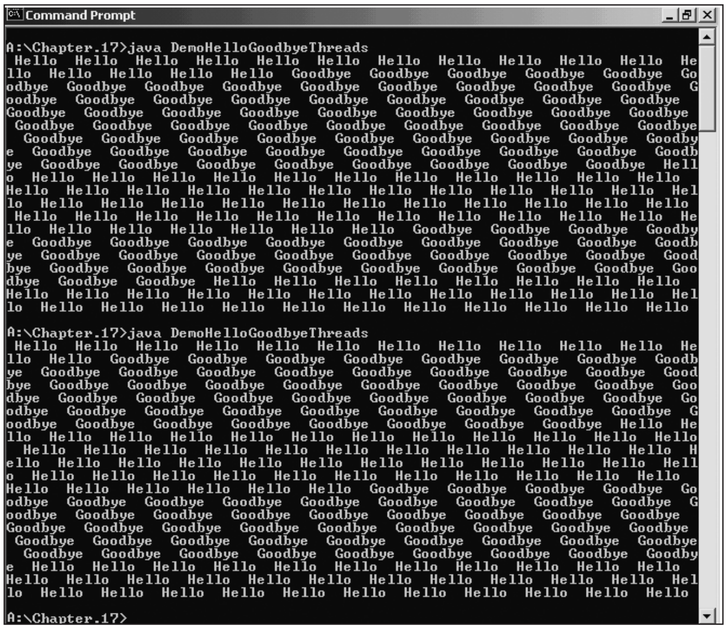


```
A:\Chapter.17>java DemoHelloGoodbyeThreads
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodb
ye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Good
bye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goo
dbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Go
odbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  G
oodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Hello  He
llo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  H
ello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hell
o  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Go
odbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  G
oodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
 Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodby
e  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hel
lo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello

A:\Chapter.17>_
```

Figure 17-9 Output of the DemoHelloGoodbyeThreads program

17

When you run a program like DemoHelloGoodbyeThreads, your output might not appear exactly like Figure 17-9. In fact, if you run your program multiple times, your output might look like Figure 17-10, which shows two subsequent runs of the program. Notice that the first execution displays 16 "Hello" messages before the first "Goodbye" message appears, but the second execution displays 13 "Hello" messages prior to the first "Goodbye". Other differences also appear in the order in which the two executions are displayed. Depending on available resources, the operating system might alternate between "Hello" and "Goodbye", or execute several repetitions of the HelloThread in a row, followed by several repetitions of the GoodbyeThread. Each Thread completes its 100 repetitions, but you have no guarantee as to the exact pattern of execution; your only guarantee is that the threads execute concurrently.



```
A:\Chapter.17>java DemoHelloGoodbyeThreads
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Hello  Hello  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Go
odbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  G
oodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
 Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodby
e  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodb
ye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Hell
o  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hel
lo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Hello  Hello  Hello  Hello  Hello  Goodbye  Goodbye  Goodbye  Goodby
e  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodb
ye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Good
bye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goo
dbye  Goodbye  Goodbye  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hel
lo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello

A:\Chapter.17>java DemoHelloGoodbyeThreads
 Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  He
llo  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodb
ye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Good
bye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goo
dbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Go
odbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  G
oodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Hello  He
llo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  H
ello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hell
o  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Go
odbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  G
oodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
 Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye
  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  Goodby
e  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello
Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hel
lo  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello  Hello

A:\Chapter.17>
```

Figure 17-10 Multiple runs of the DemoHelloGoodbyeThreads program

Next you will extend the Thread class so it repeats a character on the screen. Then you will create three Thread objects to observe how they execute concurrently.

To create a class that extends Thread:

1. Open a new file in your text editor, and then type the following first few lines of the ShowThread class. This class has two data members: a character that you will display on the screen and an integer that holds the number of repetitions to be displayed.

```
class ShowThread extends Thread
{
       private char oneChar;
       private int rep = 100;
```

2. Create the following constructor for the ShowThread class. The constructor accepts a character and assigns it to the oneChar variable.

```
public ShowThread(char printChar)
{
       oneChar = printChar;
}
```

3. Enter the following run() method, which prints the character as many times as the rep value specifies:

   ```
   public void run()
   {
          for(int x = 0; x < rep; ++x)
                 System.out.print(oneChar);
   }
   ```

4. Add a closing curly brace for the class.
5. Save the file as **ShowThread.java** in the Chapter.17 folder on your Student Disk, and then compile the file using the **`javac`** command.

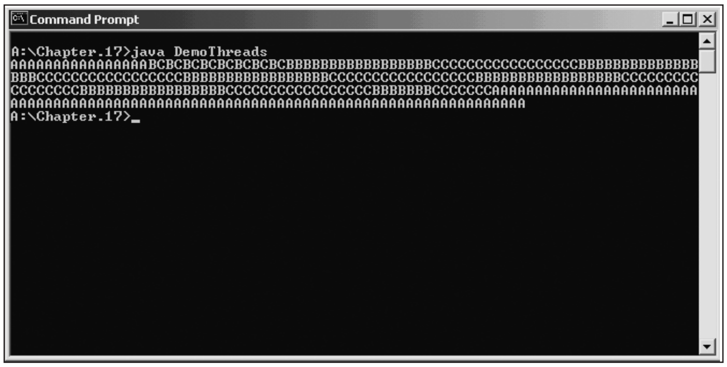
Next you will write a DemoThreads class containing a main() method that declares and uses three ShowThread objects.

To write the DemoThreads class:

1. Open a new file in your text editor, and then type the following class:

   ```
   class DemoThreads
   {
          public static void main(String[] args)
          {
                 ShowThread showA = new ShowThread('A');
                 ShowThread showB = new ShowThread('B');
                 ShowThread showC = new ShowThread('C');
                 showA.start();
                 showB.start();
                 showC.start();
          }
   }
   ```

2. Save the file as **DemoThreads.java** in the Chapter.17 folder on your Student Disk, and then compile and execute the program. The output appears similarly to Figure 17-11, although the execution sequence of your ShowThread objects might vary.
3. Run the program several more times and examine the output for changes in the execution sequence.
4. Open the **ShowThread.java** file in your text editor, and then change the value in the rep variable definition so that it is **`500`** (for example, `private int rep = 500;`). Save the class using the same filename, and then compile and execute it.



```
A:\Chapter.17>java DemoThreads
AAAAAAAAAAAAAAAABCBCBCBCBCBCBCBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBBB
BBBCCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBBBBCCCCCCCCC
CCCCCCCBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCBBBBBBBCCCCCCCAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A:\Chapter.17>_
```

Figure 17-11 Output of the DemoThreads program

5. Recompile the **DemoThreads.java** file, and then execute it. Observe any differences in your output.

USING THE SLEEP() METHOD

One interesting and useful member of the Thread class is the sleep() method. You use the sleep() method to pause a Thread for a specified number of milliseconds. For example, writing `sleep(500);` within a Thread class's run() method causes the execution to rest for 500 milliseconds, or half a second. You might use the sleep() method to pause a program to give a user time to respond to a question or to absorb an image before displaying subsequent images in a series.

Tip

You might also use sleep() to give lower priority threads a chance to execute. The next section explains the concept of priority.

When you use the sleep() method, you must `catch` an InterruptedException—the type of Exception thrown when a program stops running before its natural end. When you use the sleep() method, you intend for your program to be interrupted, so you usually `catch` the Exception and take no action.

To demonstrate using the sleep() method:

1. Open a new file in your text editor, and then type the following lines to begin the SleepThread class:

```
import java.awt.*;
class SleepThread extends Thread
{
```

2. Enter the following first few lines of the run() method, which uses a `for` loop to print 100 integers:

```
public void run()
{
  for(int x = 0; x < 100; ++x)
  {
          System.out.print(x + " ");
```

3. Within the `for` loop, add the following `try` block. You will test the loop counter variable, x, and when it equals 25, pause for three seconds; when it equals 75, pause for five seconds. End the `try` block, and then `catch` the InterruptedException.

```
try
{
   if(x == 25)
           sleep(3000);
   if(x == 75)
           sleep(5000);
}
catch(InterruptedException e)
{
}
```

4. Add three closing curly braces: one for the `for` loop, one for the run() method, and one for the SleepThread class.
5. Save the file as **SleepThread.java** in the Chapter.17 folder on you Student Disk, and then compile it using the `javac` command.
6. Open a new file in your text editor, and then type the following DemoSleepThread class that will create a SleepThread object and demonstrate its use:

```
class DemoSleepThread
{
    public static void main(String[] args)
    {
           SleepThread sThread = new SleepThread();
           sThread.start();
    }
}
```

7. Save the file as **DemoSleepThread.java** in the Chapter.17 folder, and then compile and execute the program. When the program runs, 25 integers display, and then you must wait during a three-second sleep. When the program resumes, 50 more integers display, and then you must wait during a five-second sleep. When the program completes, all 100 integers (0 through 99) appear on the screen.

Setting Thread Priority

Every Java Thread has a **priority**, or rank, in terms of preferential access to the operating system's resources. Threads with the same priority are called **peers**. With some operating systems (like Windows), threads are timesliced; that is, each peer receives a set amount of processor time during which it can execute. When that time is up, even if the thread has not finished its execution, the next thread that has equal priority receives a slice of time. Without timeslicing (as in the Solaris operating system), a thread runs to completion before one of its peers can execute.

Each Thread object's priority is represented by an integer in the range of 1 to 10. If you do not assign a priority to a Thread object, it assumes a default priority of 5. You can change a Thread's priority by using the setPriority() method, or determine a Thread's priority with the getPriority() method. If you extend a Thread from an existing Thread, the child Thread assumes its parent's priority.

The Thread class contains three constants. They are MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY, which represent 1, 5, and 10, respectively. When you set a Thread's priority, you can use an integer, as in `myThread.setPriority(10);`, or you can use one of the three constants, as in `myThread.setPriority(Thread.MAX_PRIORITY);`. You also can use an arithmetic expression, such as `yourThread.setPriority(Thread.MAX_PRIORITY - 2);`.

Tip: If you use the priority constants with your Thread objects, then your Threads will have the appropriate relative priority even if the developers of Java decide to change the priority values in the future.

When you run a Java program, the runnable Thread with the highest priority runs first. If several Threads have the same priority, they run in rotation. Lower-priority Threads can run only when higher-priority Threads are not runnable (such as when they are finished, suspended, or asleep).

In general, when ThreadA has higher priority than ThreadB, ThreadA will be running and ThreadB will be waiting. However, sometimes Java will choose to run ThreadB to avoid starvation. **Starvation** occurs when a Thread cannot make any progress because of the priorities of other Threads. The result of starvation can be similar to creating an infinite loop—the program runs continuously without completing because one or more threads never get the opportunity to execute.

Tip: The ultimate form of starvation is called deadlock. Deadlock occurs when two Threads must wait for each other to do something before either can progress.

Next you will set the priorities of some Threads and observe the effects of those settings.

To demonstrate Thread priorities:

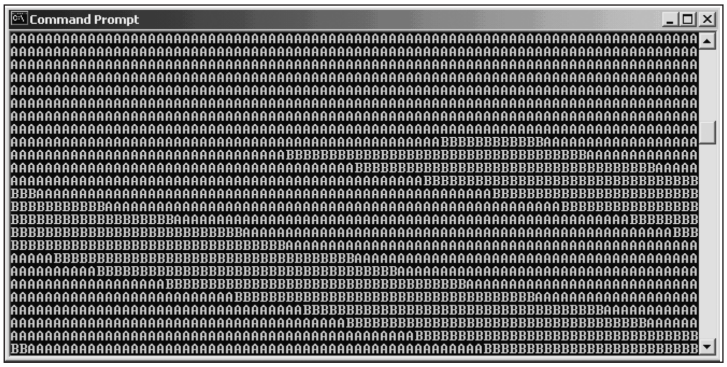
1. Open the **ShowThread.java** file in your text editor, and immediately save it as **ShowThread2.java**.
2. Change the class name and the constructor name to **ShowThread2**.
3. Change the value of the rep variable to **`5000`**. This step is necessary on most systems so that there will be enough executions of the output loop for you to observe the effect of the priority settings.
4. Save the file and compile it using the **`javac`** command.
5. Open the **DemoThreads.java** file in your text editor, and immediately save it as **DemoThreadsPriority.java**.
6. Change the class header to **`class DemoThreadsPriority`**.
7. Change the declaration of the three Threads to use the newly created ShowThread2 class. In other words, the declarations become the following:

   ```
   ShowThread2 showA = new ShowThread2('A');
   ShowThread2 showB = new ShowThread2('B');
   ShowThread2 showC = new ShowThread2('C');
   ```

8. Position your insertion point to the right of the line with the third ShowThread2 declaration (the showC declaration), and then press **[Enter]** to start a new line.
9. Type the following lines to set the showB Thread's priority to 4 and the showC Thread's priority to 6:

   ```
   showB.setPriority(Thread.NORM_PRIORITY-1);
   showC.setPriority(Thread.NORM_PRIORITY+1);
   ```

10. Save the file, and then compile and run the program. The output is more than will fit on your screen, and the middle of the execution looks similar to Figure 17-12. Although the As, Bs, and Cs are somewhat intermingled, the showC Thread, with a priority of 6, finishes before showA and showB. Notice that although the highest priority Thread (showC) finishes first, and the next highest (showA) finishes second, showB is allowed some processor time before showA finishes.



```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBB
```

Figure 17-12 Middle part of the DemoThreadsPriority program output

USING THE RUNNABLE INTERFACE

You can create a Thread subclass by inheriting from the Thread class, but this approach won't work if you want your Thread subclass to inherit from another class as well. You have already learned that Java does not allow a class to inherit from more than one superclass. So, for example, if you want to create a JApplet that can run as a Thread, you cannot inherit from both JApplet and Thread; instead you must implement an interface. You can implement the Runnable interface as an alternative to inheriting from the Thread class.

Tip: You learned about implementing interfaces in Chapter 12.

You can write an applet that acts as a timer; the applet counts and displays seconds as they pass. You declare an integer variable—for example, `int secs = 0;`. The applet's paint() method contains a statement that adds one to the secs variable, and then displays the current count of seconds.

To let the applet run indefinitely, you can create an infinite loop that repaints the screen every 1,000 milliseconds (or every second), as in the following code:

```
while(true)  // execute forever
       repaint();
try
{
       clock.sleep(1000);
}
catch(InterruptedException e)
{
}
```

The problem with this approach is that as long as the infinite while loop is running, the CPU is occupied and cannot perform any other actions, such as carrying out the repaint() method. Instead of displaying a counter, the applet appears to freeze the screen.

The solution is to place the `while` loop in a Thread that can share time with the operating system's default Thread in which the applet runs. Next you will produce a JApplet that uses this approach as you create a JApplet that shares processor time with the clock Thread that hosts it.

To create a JApplet that implements the Runnable interface:

1. Open a new file in your text editor, and then type the following first few lines of the TimerApplet applet:

   ```
   import javax.swing.*;
   import java.awt.*;
   public class TimerApplet extends JApplet
     implements Runnable
   {
   ```

2. Define three JLabels. The first and third contain literal Strings. The second will display the changing time count.

   ```
   private JLabel label1 = new JLabel("Time is passing.");
   private JLabel label2 = new JLabel();
   private JLabel label3 = new
     JLabel("Plan your event today.");
   ```

3. Declare a Container, a variable to hold the seconds, and a Thread named clock:

   ```
   private Container c = getContentPane();
   private int secs = 0;
   private Thread clock;
   ```

4. The init() method establishes the layout manager, adds the three JLabels to the content pane, and declares a Thread object named clock. When the applet is initialized, the clock Thread will have value `null` because the constructor has not been called yet, so within the init() method, you instantiate the Thread. The run() method of the TimerApplet will execute when the Thread is created because the `this` in `clock = new Thread(this);` refers to "this" applet. Finally, add start() to the Thread.

   ```
   public void init()
   {
     c.setLayout(new FlowLayout());
     c.add(label1);
     c.add(label2);
     c.add(label3);
     if(clock == null)
          clock = new Thread(this);
     clock.start();
   }
   ```

5. The applet contains the following run() method because the applet implements Runnable. Within the run() method, you place the infinite loop that calls repaint() every second and provides for the InterruptedException thrown by the sleep() method.

```
public void run()
{
  while(true)
  {
      repaint();
      try
      {
             clock.sleep(1000);
      }
      catch(InterruptedException e)
      {
      }
  }
}
```

6. Add the following paint() method that adds one to the secs variable (because paint() is called once every second) and updates the displayed count in label2:

```
public void paint(Graphics gr)
{
  ++secs;
  label2.setText("Time: " + secs);
}
```

7. Add a closing curly brace for the applet.

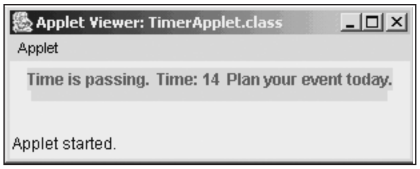
8. Save the applet as **TimerApplet.java** in the Chapter.17 folder on your Student Disk. Compile the applet using the `javac` command.

9. Open a new file in your text editor, and then type the following HTML document to host your applet:

```
<HTML>
<APPLET CODE = "TimerApplet.class"
    WIDTH = 300 HEIGHT = 50>
</APPLET>
</HTML>
```

10. Save the HTML file as **TestTimer.html** in the Chapter.17 folder, and then execute it using the `appletviewer` command. The output appears similar to Figure 17-13. As you watch the timer, note that the time value changes every second.

11. Click the **Close** button to close the Applet Viewer window.



```
Time is passing.  Time: 14  Plan your event today.
```

```
Applet started.
```

Figure 17-13 Output of the TimerApplet program

The TimerApplet that you created uses the default system Thread, and your clock object uses its own Thread. You could not have created this applet without using the power of multithreading.

CREATING AN ANIMATED FIGURE

Cartoonists create animated films by drawing a sequence of frames or cells. These individual drawings are shown to the audience in rapid succession to give the illusion of natural movement. You create computer animation using the same techniques. If you display computer images as fast as your CPU can process them, you might not be able to see anything. Most computer animation employs the Thread class sleep() method to pause for short periods of time between animation cells, so the human brain has time to absorb each image's content.

Artists often spend a great deal of time creating the exact images they want to use in an animation sequence. As a much simpler example, Event Handlers Incorporated wants you to create a stick figure whose arm moves up and down. The only difference between creating a stick figure and a more complex graphic image is in the amount of time and degree of artistic talent you have; the programming skills you use are the same.

You begin to create computer animation by using graph paper to sketch a figure, as shown in Figure 17-14.

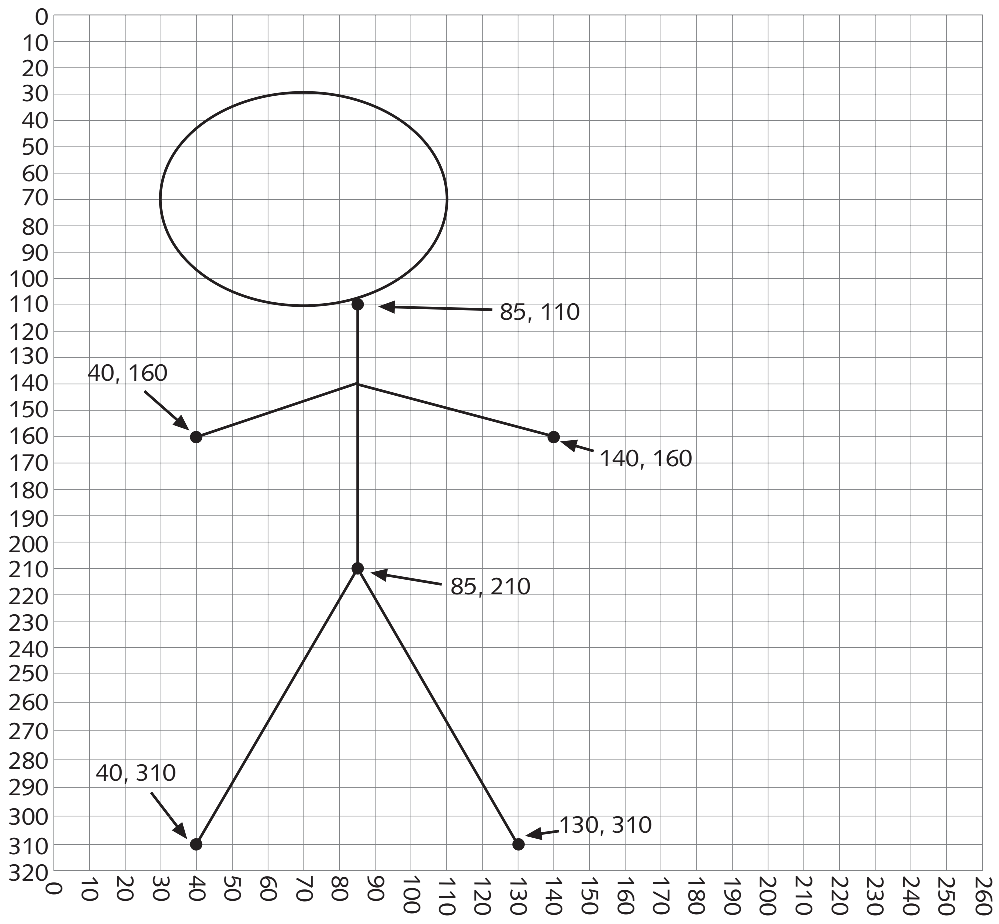


Figure 17-14 Sketch of a stick figure

To create the illusion that this figure's arm moves, you create a second sketch in which the arm is slightly raised from its position in the first sketch. In your third drawing the arm is slightly higher, and so on. Figure 17-15 shows four potential arm positions for the stick figure.

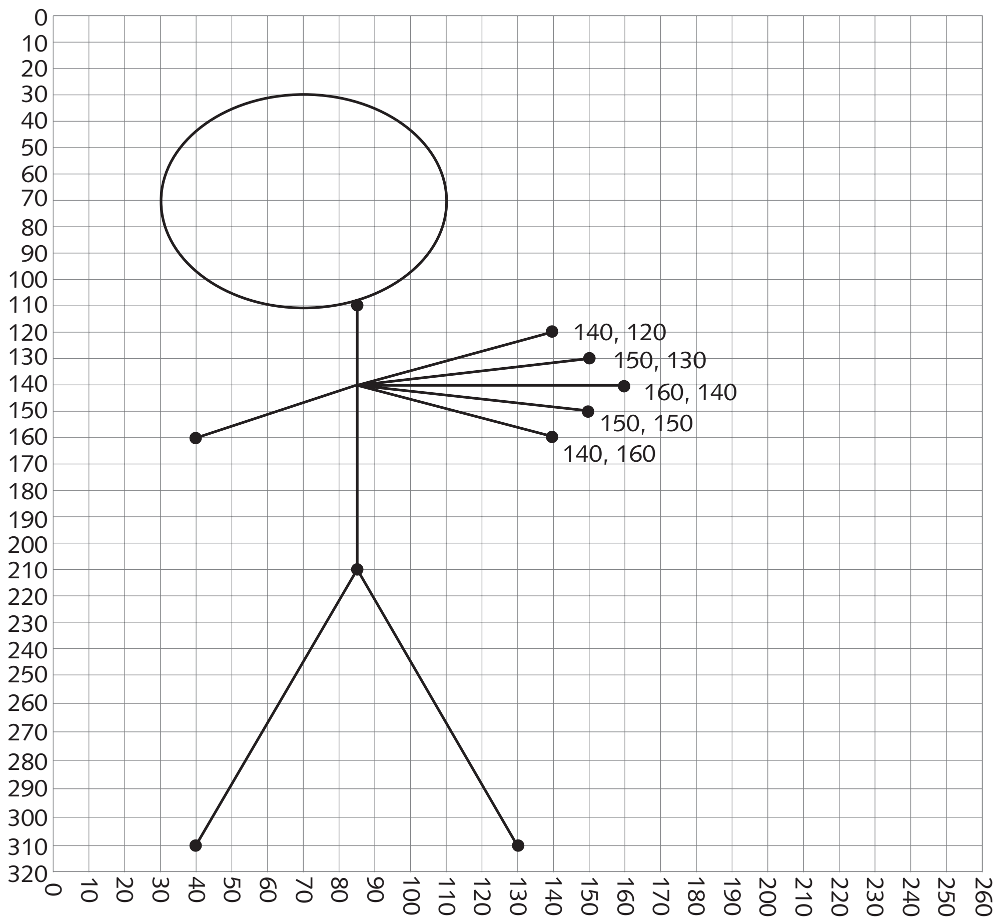


Figure 17-15 Sketch of four arm positions for the stick figure

Each time you draw the stick figure on the screen, the arm will appear slightly higher, giving the viewer the illusion of movement. The head, body, and legs of the stick figure never change. You can draw the stick figure's body using constant values for the drawing coordinates, but you need variables to store the horizontal and vertical positions of the end of the moving arm. One efficient approach is to store the x- and y-coordinates for the end location of each of the arm positions in parallel arrays, as in the following example:

```
int[] horiz = {140, 150, 160, 150, 140};
int[] vert =  {160, 150, 140, 130, 120};
```

Each time you redraw the stick figure, you can increase a subscript and use the next horizontal and vertical coordinate pair to indicate the end of the figure's arm. When you have used all five arm coordinates, you can reduce the subscript to a value of zero, and then begin again. Next you will create a stick figure and use an array of coordinates to animate its arm.

To create an animated stick figure for Event Handlers Incorporated:

1. Open a new file in your text editor, and then type the following first few lines of the AnimatedFigure class. (You will inherit from Applet instead of from JApplet because Applets' surfaces are automatically updated when repainted.)

   ```
   import java.applet.*;
   import java.awt.*;
   public class AnimatedFigure extends Applet
   {
   ```

2. Declare an integer index that you can use to access the horizontal and vertical position arrays. Also declare the arrays themselves, which are loaded with values taken from the preliminary sketches:

   ```
   private int index = 0;
   int[] horiz = {140,150,160,150,140};
   int[] vert =  {160,150,140,130,120};
   ```

3. Add the following variable to store the sleep time. If you want to speed up or slow down the figure later, it will be convenient to change the value in this variable. For now, set the sleep time to 100 milliseconds as follows:

   ```
   private int sleep = 100;
   ```

4. In the applet's start() method, initialize the index to 0:

   ```
   public void start()
   {
     index = 0;
   }
   ```

5. Within the applet's paint() method, add the following code to draw the figure's head, body, and two legs. Use the sketch as a reference for the coordinates:

   ```
   public void paint(Graphics gr)
   {
     gr.drawOval(30,30,80,80);
     gr.drawLine(85,110,85,210);
     gr.drawLine(85,210,40,310);
     gr.drawLine(85,210,130,310);
   ```

6. You can create the figure's left arm by using the coordinates in the sketch. However, creating the right arm is more complicated. Although the right arm always starts at the same position on the body, the hand end of the arm might be in any one of four positions taken from the horiz and vert arrays. After you draw the right arm using position 0 from each array, you want to increase the index, so you will use the next pair of coordinates the next time you draw the arm. When the index eventually exceeds the highest subscript allowed, reset the index to 0 so the process of drawing each of the arm positions can start over.

```
gr.drawLine(85,140,40,160);
gr.drawLine(85,140,horiz[index],vert[index]);
++index;
if(index == horiz.length)
     index = 0;
```

7. Add the following `try` block to make the thread sleep for the designated amount of time, and then add a `catch` block to handle the InterruptedException.

```
try
{
  Thread.sleep(sleep);
}
catch(InterruptedException e)
{
}
```

8. The last step in the paint() method is to call the repaint() method. This restarts paint() to draw the figure with a new arm position, update the index, and sleep. Add two curly braces—one to close the paint() method and one to close the class.

```
    repaint();
  }
}
```

9. Save the file as **AnimatedFigure.java** in the Chapter.17 folder on your Student Disk, and then compile it.

10. Open a new file in your text editor and create the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "AnimatedFigure.class"
  WIDTH = 350 HEIGHT = 400>
</APPLET>
</HTML>
```

11. Save the HTML file as **TestAnim.html**, and then use the **`appletviewer`** command to run the applet. The stick figure's arm waves up and down. (You might notice some flickering on the screen; you will learn to eliminate any flickering in the next section.) One frame of the output appears in Figure 17-16.

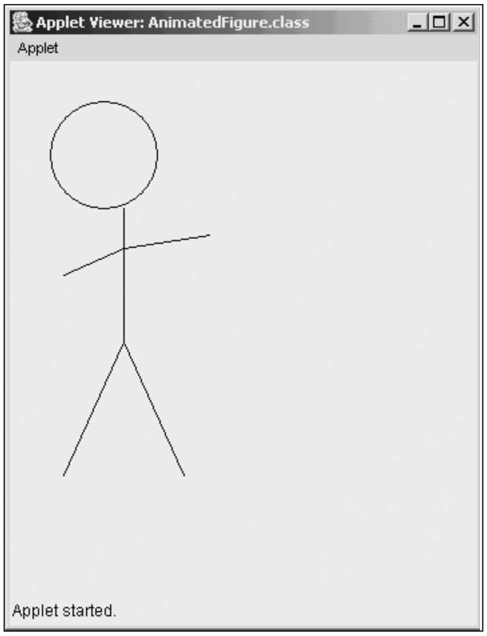


Figure 17-16 Output of the AnimatedFigure applet

12. Close the Applet Viewer by clicking its **Close** button.

Next you will add some text to the applet. Event Handlers Incorporated wants the company name to move up and down on the screen, as though the stick figure is using the name as a yo-yo. You will also add text which will appear in a fixed position in the applet.

To add moving text to the applet:

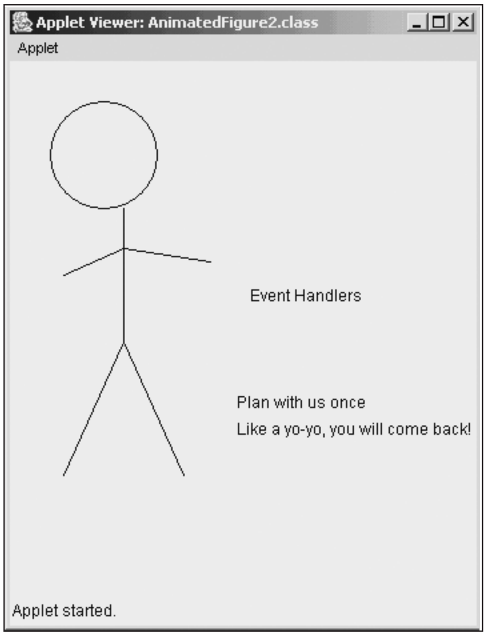
1. Open the **AnimatedFigure.java** file in your text editor if it is not still open, and then immediately save it as **AnimatedFigure2.java** in the Chapter.17 folder on your Student Disk.
2. Change the class name to **`AnimatedFigure2`**.
3. Within the paint() method, position your insertion point at the end of the last drawLine() method call (the statement that draws the moving arm), and then press **[Enter]** to start a new line.
4. To draw the "Event Handlers" name as though it is moving along with the stick figure's arm, indicate its vertical position as 30 pixels below the arm's current vertical position as follows:

```
gr.drawString("Event Handlers",180, vert[index] + 30);
```

5. Add the following text lines at lower positions on the next two lines:

```
gr.drawString("Plan with us once",170,260);
gr.drawString("Like a yo-yo,
  you will come back!",170,280);
```

6. Save the file, and then compile it using the **`javac`** command.
7. Open the **TestAnim.html** file in your text editor. Change the APPLET CODE reference to **`"AnimatedFigure2.class"`**, and then save the HTML document as **TestAnim2.html** in the Chapter.17 folder on your Student Disk.
8. Use the **`appletviewer TestAnim2.html`**command to run the applet. The stick figure appears to use the Event Handlers company name as a yo-yo. One snapshot of the output is shown in Figure 17-17.



Event Handlers

Plan with us once

Like a yo-yo, you will come back!

Applet started.

Figure 17-17 Output of the AnimatedFigure2 applet

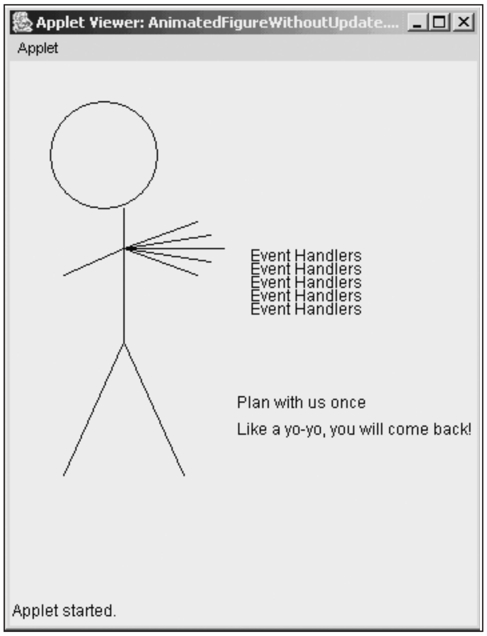
9. Close the AppletViewer window.

Reducing Flickering

When you create an applet with animation, your screen might flicker. To understand why screens flicker, recall the applet life cycle that you learned about in Chapter 9.

The faster your processor speed is, the less flickering you will see. The stick figure you created does not require a lot of drawing, so even with a processor of moderate speed you may not experience flickering. However, if you create a detailed drawing, you might experience flickering even with a fast processor, and you will want to employ the techniques you learn in this section.

When you change a drawing in an applet, such as when you reposition the stick figure's arm and the moving text String, you call the repaint() method to repaint the applet surface. The repaint() method calls the update() method, which clears the viewing area, and then calls the paint() method, which contains the instructions for drawing the figure and String in their new positions. If the repaint() method did not call update() to clear the screen, then all previous versions of the applet would remain visible. After the first five passes through the paint() method you would see all five of the figure's arms and all five messages, as shown in Figure 17-18.



```
Event Handlers
Event Handlers
Event Handlers
Event Handlers
Event Handlers

Plan with us once
Like a yo-yo, you will come back!
```

Figure 17-18 AnimatedFigure without the update() method

Your program must clear the screen so only one version of the arm and message appears at a time. However, clearing the screen takes enough time that your eye detects it; this results in the flickering you see on your screen. One way to reduce or eliminate flickering is to override the applet's update() method so that the viewing area is not cleared. After all, every time you call update() to clear the screen, the head, body, legs, and left arm of the stick figure and the two messages are immediately redrawn in the same positions from which they were just cleared. It is more efficient to draw the unchanging parts of the screen just once when the applet starts, and then constantly redraw only those portions of the screen that change.

The trick for erasing a portion of a drawing on the screen is to use the applet's background color to redraw the portion of the screen that you want to erase. When you create graphics using the background color, the graphics seem to disappear. In other words, to appear to move the stick figure's arm, you draw over its old arm in the background color, and then draw the new arm position in the previous drawing color.

For example, if the applet background color is white and the drawing color is black, one way to draw over the old black arm line is to save the arm coordinates after you draw the arm. Then before you draw a new black arm line, draw over the old line in white, using the saved coordinates.

To reduce flickering in the AnimatedFigure2 applet:

1. If it is not open, open the **AnimatedFigure2.java** file in your text editor. Immediately save the file as **AnimatedFigure3.java** in the Chapter.17 folder on your Student Disk.
2. Change the class name from AnimatedFigure2 to **`AnimatedFigure3`**.
3. Position the insertion point at the end of the statement that declares the vert array, and then press **[Enter]** to start a new line. Create the following two new integer variables that will hold the previous arm coordinates. Initialize these variables to the first pair of coordinates. Then create a variable to indicate whether the paint() method is on its first or a subsequent execution.

   ```
   int oldx = horiz[0], oldy = vert[0];
   boolean firstTime = true;
   ```

4. Position your insertion point to the right of the closing curly brace for the start() method, and then press **[Enter]** to start a new line. Type the following code to override the applet's automatic update() method. The replacement update() method calls paint(), passing along the Graphics object received from the repaint() method.

   ```
   public void update(Graphics gr)
   {
     paint(gr);
   }
   ```

5. Delete the current paint() method statements up to, but not including, the statement `++index;`. Begin the following replacement paint() method by drawing the head, body, legs, left arm, and constant text only if this is the first pass through the paint() method. Conclude the `if` block by setting the firstTime variable to `false`.

```
public void paint(Graphics gr)
{
       if(firstTime)
       {
          gr.drawOval(30,30,80,80);
          gr.drawLine(85,110,85,210);
          gr.drawLine(85,210,40,310);
          gr.drawLine(85,210,130,310);
          gr.drawLine(85,140,40,160);
          gr.drawString("Plan with us once",170,260);
          gr.drawString("Like a yo-yo, you will come
            back!",170,280);
          firstTime = false;
       }
```

6. Continue adding the following paint() method statements that execute every time the paint() method executes—that is, whether or not it is the first execution. First, set the drawing color to the background color, and redraw the arms and moving String in the background color to make them invisible.

```
gr.setColor(getBackground());
gr.drawLine(85,140,oldx, oldy);
gr.drawString("Event Handlers",180, oldy + 30);
```

7. Add the following code to change the drawing color back to the usual drawing color, and then to draw the new arm and the new moving message in their new positions:

```
gr.setColor(getForeground());
gr.drawLine(85,140,horiz[index],vert[index]);
gr.drawString("Event Handlers",180, vert[index] + 30);
```

8. Before increasing the index for the next execution of the paint() method, use the following code to save the current arm coordinates by storing them in the oldx and oldy variables:

```
oldx = horiz[index];
oldy = vert[index];
```

9. Save the file, and then compile it using the `javac` command.
10. Open the **TestAnim2.html** file in your text editor. Change the APPLET CODE reference to `"AnimatedFigure3.class"`, and then save the HTML document as **TestAnim3.html** in the Chapter.17 folder on your Student Disk.
11. Run the applet using the `appletviewer` command. When the applet runs, you will not see any flickering. (If your animation appears to run too fast or too slowly, change the value of the sleep variable.)
12. Minimize the Applet Viewer window, and then restore it. You can see only a disembodied moving arm and message. Close the Applet Viewer window.

The flickering disappears in the AnimatedFigure3 applet because the update() method no longer redraws the entire screen. However, a problem occurs when you minimize, restore, or move the applet—the figure no longer is visible and only the moving arm and message remain. When you start the applet, firstTime is set to `true`, the figure is drawn, and firstTime is set to `false`. When you minimize the applet, it is still running. When you restore the applet, the applet is redrawn, but because the applet is still running, firstTime is already set to `false`. The head and other body parts are drawn only when firstTime is true, so the applet now shows a disembodied arm. Next you will correct this problematic development.

To correct the applet problem:

1. Open the **AnimatedFigure3.java** file in your text editor, and immediately save it as **AnimatedFigure4.java**.
2. Change the class name to **`AnimatedFigure4`**.
3. Within the start() method, position your insertion point to the right of the statement that sets the index equal to 0, and then press **[Enter]** to start a new line.
4. Add the statement **`firstTime = true;`**. Now, every time the applet restarts, the applet will redraw the head and body.
5. Save the file, and then compile it using the **`javac`** command.
6. Open the **TestAnim3.html** file in your text editor. Change the APPLET CODE reference to **`"AnimatedFigure4.class"`**, and then save the HTML document as **TestAnim4.html** in the Chapter.17 folder on your Student Disk.
7. Run the applet using the **`appletviewer`** command. The applet runs the same as before. Minimize the Applet Viewer window and then restore it. The entire figure and complete set of messages are visible, because they are redrawn every time the start() method executes.
8. Close the Applet Viewer window.

USING PRE-DRAWN ANIMATED IMAGE OBJECTS

If your artistic talent is limited to drawing stick figures, you can substitute a variety of more sophisticated, pre-drawn animated images to achieve the graphic effects you want within your applets. In Chapter 10, you used the getImage() method to load a stored image into an applet. You can also use getImage() to load animated files in your programs. When you call getImage(), an image is loaded in a separate thread of execution; this allows program execution to continue while the image loads. When you used getImage() in Chapter 10, you did not create a thread—Java created one for you. Because loading images can be a time-consuming task, it is a great advantage that Java automatically creates a separate thread for them. Recall that the getImage() method requires two arguments: a URL object and the name of an image file.

Next you will load a pre-drawn animated .gif file into an applet and execute it. The .gif file includes 16 frames that display a falling egg that Event Handlers Incorporated is using in an advertising slogan to give Event Handlers a "crack" at planning the customer's next party.

To demonstrate loading a pre-drawn animated image into an applet:

1. Open a new file in your text editor, and then enter the following first few lines of an applet that will load a pre-drawn animated image:

```
import java.applet.*;
import java.awt.*;
public class LoadImage extends Applet
{
```

2. Declare an Image object that will represent the falling egg as follows:

```
private Image egg;
```

3. The Chapter.17 folder on your Student Disk contains an animated .gif file named egg.gif. Type the following init() method, which loads the egg.gif file in the applet:

```
public void init()
{
  egg = getImage(getDocumentBase(),"egg.gif");
}
```

4. Type the following paint() method, which draws the egg Image at location 1,1 within the applet, and also draws two Strings:

```
public void paint(Graphics g)
{
  g.drawImage(egg,1,1,this);
  g.drawString("Planning your next party?",100,50);
  g.drawString
    ("Give Event Handlers a crack at it.",100,   300);
}
```

5. Type a closing curly brace for the applet.

6. Save the file as **LoadImage.java** in the Chapter.17 folder on your Student Disk, and then compile the applet using the `javac LoadImage.java` command.

7. Open a new file in your text editor, and then create the following HTML document:

```
<HTML>
<APPLET CODE = "LoadImage.class"
  WIDTH = 300 HEIGHT = 350>
</APPLET>
</HTML>
```

8. Save the HTML document as **TestImage.html** in the Chapter.17 folder of your Student Disk.
9. Execute the HTML file using the `appletviewer` command. The animation shows an egg that falls and cracks open at the bottom of the applet viewing area. Figure 17-19 shows the final frame of the animation.

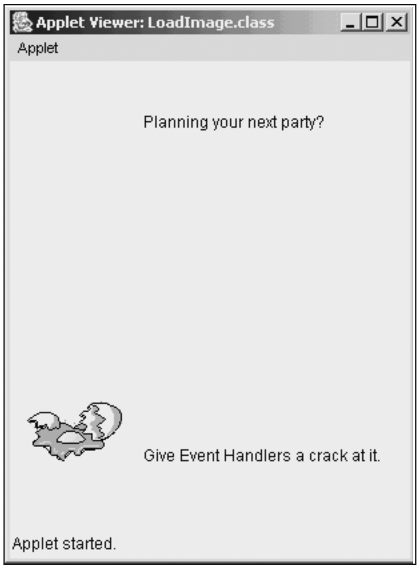

Applet Viewer: LoadImage.class
Applet
Planning your next party?
Give Event Handlers a crack at it.
Applet started.

Figure 17-19 LoadImage applet

10. Watch the egg fall and crack open repeatedly. When you are ready, close the Applet Viewer window.

Tip

Many animated images are available on the Web for you to use freely. Use your search engine to search for keywords such as gif files, jpeg files, and animation to find sources for shareware files. The egg file was found at *www.mediabuilder.com*.

Understanding Garbage Collection

The Java programming language supports a garbage collection feature that you seldom find in other programming languages. The **garbage collector** provides for the automatic cleanup of unnecessarily reserved memory.

With many programming languages, when you allocate memory from the memory that is available (often known as the **heap**), you must purposely deallocate it or your system's performance begins to slow as less memory is available. The Java garbage collector automatically sweeps through memory looking for unneeded objects and destroys them. Out-of-scope objects or objects with `null` references are available for collection. Partially constructed objects that throw an Exception during construction are also available for collection.

Tip

You first learned about garbage collection in Chapter 7 when String memory addresses were introduced. Recall that when you change a String's contents, the previous contents still exist in memory. These unused characters would needlessly occupy memory if they were not collected and disposed of.

Garbage collection works by running as a very low-priority Thread. Typically, this Thread runs only if available memory is very low. That is, all other Threads must be delayed by memory limitations before the garbage collector gets the opportunity to run.

You cannot prevent the garbage collector from running, but you can request that it run by using the statement `System.gc();`. Suppose you write a program that counts seconds and uses a String named counter to hold the values "one", "two", and so on. When you assign "two" to counter, the memory that holds the character string "one" becomes garbage. Instead of allowing the useless strings to accumulate in memory, you might want to use the `System.gc();` statement. Using this statement does not force the garbage collector to run; it is only a request to the system to schedule the garbage collector. The garbage collector will run when all other Threads are delayed.

Putting Animation in a Web Browser Page

If you browse the Internet, you probably have seen examples of animation on Web pages. Web page creators use animation to attract your attention and to make their pages more appealing. Next you will create an applet that displays a moving word. As a stand-alone applet it is interesting, but using the applet becomes much more interesting when you run several versions of it in a Web browser at the same time.

Each version of the applet displays the single String "Party" at x- and y-coordinates. After each display, you use the Thread class sleep() method to pause the animation. You then erase the String by drawing it in the background color; you then redraw it in a new position. The new position is three pixels to the right and down from the previous String, so the illusion is that the String is moving down and to the right. When the String reaches the right edge of the viewing area, you begin to subtract three pixels from the horizontal and vertical coordinates, so the String appears to reverse course and move up and to the left.

To create the first of three applets containing a moving word that you will run in the browser:

1. Open a new file in your text editor, and then type the following opening lines for the BouncingParty1 applet:

```
import java.applet.*;
import java.awt.*;
public class BouncingParty1 extends Applet
{
```

2. Establish variables for the step value increase for each drawString() and for the x- and y-coordinates as follows:

```
private int step = 3;
private int x = 10, y = 10;
```

3. Also establish variables to hold the maximum screen positions, the previous screen positions, and the sleep interval as follows:

```
private int maxX = 100, maxY = 100;
int oldx, oldy;
private int sleep = 50;
```

4. Type the following update() method that calls the paint() method (so Java doesn't call its own update() method, which would clear the viewing area and increase flickering):

```
public void update(Graphics gr)
{
  paint(gr);
}
```

5. Begin the following paint() method, which draws "Party" using the background color at the previous x- and y-coordinates, and then draws the String using the foreground color at the new coordinates:

```
public void paint(Graphics gr)
{
  gr.setColor(getBackground());
  gr.drawString("Party", oldx, oldy);
  gr.setColor(getForeground());
  gr.drawString("Party", x, y);
```

6. Save the x and y values in the oldx and oldy variables. Then increase x and y by the step value as follows:

```
oldx = x;
oldy = y;
x += step;
y += step;
```

7. When the String approaches the edge of the applet viewing area, you want to reverse the direction of movement. You can accomplish this by changing the step value to its negative equivalent as follows:

```
if(x < 10 || x > 90)
   step = -step;
try
{
   Thread.sleep(sleep);
}
catch(InterruptedException e)
{
}
repaint();
```

8. Add the closing curly brace for the paint() method and the closing curly brace for the class.
9. Save the file as **BouncingParty1.java** in the Chapter.17 folder on your Student Disk, and then compile the class using the `javac` command.
10. In a new file in your text editor, create the following HTML document to test the class:

```
<HTML>
<APPLET CODE = "BouncingParty1.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
</HTML>
```

11. Save this file as **TestParty.html** in the Chapter.17 folder, and then use the `appletviewer` command to test the class. The word "Party" moves up and down the screen. Close the Applet Viewer window.

The simple BouncingParty1 applet is interesting, but it won't hold your attention for very long. If you create several similar applets and run them at the same time, the output will be more noteworthy. Next you will create two more applets that display a bouncing "Party" message. You will alter each applet so that the displayed String starts in a slightly different position within each applet.

To create applets in which the moving String starts in a different position:

1. Open the **BouncingParty1.java** applet in your text editor, and then immediately save it as **BouncingParty2.java**.
2. Change the class name to `BouncingParty2`.
3. Change the beginning x and y variable values from 10 to **40** so the statement becomes `private int x = 40, y = 40;`.
4. Save the file, and then compile it using the `javac` command.
5. To create the third applet, save the BouncingParty2.java file as **BouncingParty3.java**. Change the class name to `BouncingParty3`. Change the values for both the x and y variables to **70**. Save the file and compile it.

You now have three similar applets that each display a bouncing "Party" String; the only difference is that each initially places the String in a different position. In the next set of steps you will incorporate these three applets into an HTML document and view it in your Web browser. To make the HTML document more interesting, you will add two headings to the document.

HTML provides six levels of headings named `H1` through `H6`. `H1` headings are the largest, and `H6` are the smallest. Just as with the `<APPLET>` and `</APPLET>` tags you have used in your HTML documents, the heading tags come in pairs. You place text you want to display between a pair, such as `<H1>` and `</H1>`.

Tip

HTML document authors seldom use H6 headings; they are quite small and difficult to read. You first learned about HTML tags in Chapter 9.

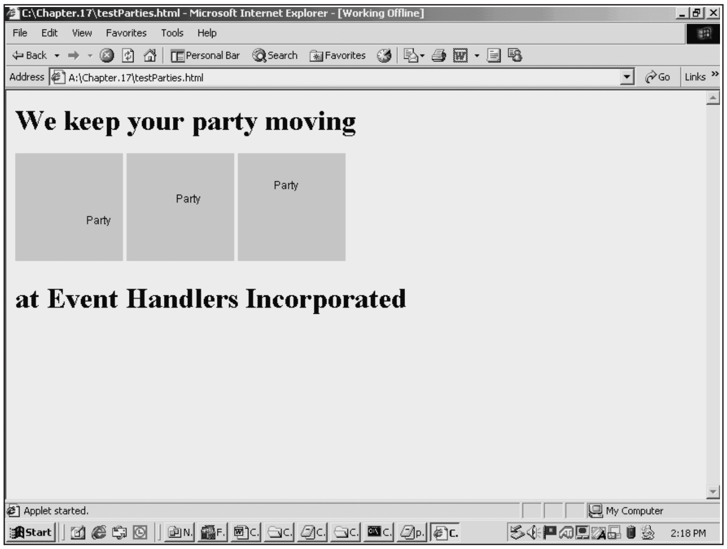
To create an HTML document to use with your browser to view the three applets:

1. Open a new file in your text editor, and then enter the following HTML document, which consists of a large heading, three applets, and another heading:

```
<HTML>
<H1>We keep your party moving</H1>
<APPLET CODE = "BouncingParty1.class"
    WIDTH = 120 HEIGHT = 120>
</APPLET>
<APPLET CODE = "BouncingParty2.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
<APPLET CODE = "BouncingParty3.class"
  WIDTH = 120 HEIGHT = 120>
</APPLET>
<H1>at Event Handlers Incorporated</H1>
</HTML>
```

2. Save the file as **TestParties.html** in the Chapter.17 folder on your Student Disk.
3. Open a Web browser such as Netscape Navigator or Microsoft Internet Explorer. (You do not need to connect to the Internet to complete this step.) From your browser's main menu, select **File**, click **Open**, and either choose the Browse button to locate your HTML document, or type its complete path; for example, **A:\Chapter.17\TestParties.html**, and then press **[Enter]**. Alternately, you can locate the TestParties.html file using Explorer or My Computer, and then double-click the file icon to open it in your default browser. The three applets run within the HTML document, as shown in Figure 17-20. Depending on your browser, the background colors for the three applets might differ.

Tip: Instead of opening the HTML document using the File menu, you can type the path to your HTML document in your browser's Location field (where you type URLs to visit Web sites), and then press **[Enter]**.


C:\Chapter.17\testParties.html - Microsoft Internet Explorer - [Working Offline]
File Edit View Favorites Tools Help
Address A:\Chapter.17\testParties.html
Applet started.
My Computer

We keep your party moving

Party Party Party

at Event Handlers Incorporated

Figure 17-20 Three applets running in the browser

4. Close your Web browser.
5. Experiment with modifying the step variable value and the sleep time in the BouncingParty applets and observe the results in your browser.
6. Try to modify one of the applets so that the "Party" message moves from upper right to lower left by altering the x- and y-coordinates so that as x gets bigger, y gets smaller.
7. Experiment with modifying the HTML document to hold additional applets.

Chapter Summary

- A thread is the flow of execution of one set of program statements. The Java programming language allows you to launch, or start, multiple threads. You use multithreading to make your programs perform better.
- You can create threads by extending the Thread class, which is defined in the java.lang package. The Thread class contains a method named run() that you override to tell the system how to execute the Thread.

- A Thread can exist in one of five states during its life: new, ready, running, inactive, or finished.
- You use the sleep() method to pause a Thread for a specified number of milliseconds. When you use the sleep() method, you must `catch` an InterruptedException.
- Every Thread in Java has a priority, or rank, in terms of preferential access to the operating system's resources. The Thread class contains three constants: MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY.
- You can implement the Runnable interface as an alternative to inheriting from the Thread class.
- You create computer animation by displaying Images in rapid sequence. Most computer animation employs the Thread class sleep() method to pause for short periods of time between animation cells; then the viewer has time to absorb each image's content.
- Clearing the screen before new images are drawn or added takes enough time that your eye detects it. As a result, the screen appears to flicker. One way to reduce or eliminate flickering is to override the applet's update() method so that the viewing area is not cleared. To "erase" a portion of a drawing on the screen, you use the applet's background color to redraw the portion of the screen that you want to erase.
- The applet method getImage() loads a stored image into an applet. The getImage() method requires two arguments: a URL object and the name of an image file.
- Garbage collection runs as a low-priority Thread and provides for the automatic cleanup of unnecessarily reserved memory.
- Web page creators use animation to attract your attention and to make their pages interesting. You can use HTML heading tags to display text in Web documents.

Review Questions

1. A thread is the ________________ one set of program statements.
   a. amount of memory occupied by
   b. flow of execution of
   c. machine language code for
   d. area of memory occupied by
2. A modern computer with a single CPU can execute ________________ statement(s) at a time.
   a. one
   b. two
   c. at least several dozen
   d. at least several thousand

3. If you use a computer with a single processor, you can execute ________________ concurrently.
   a. only one thread
   b. several threads
   c. any number of threads
   d. You cannot execute multiple threads on a single-processor computer.
4. You can create threads by ________________ the Thread class.
   a. making a copy of
   b. instantiating
   c. extending
   d. overriding
5. You override the ________________ method to tell the system how to execute a Thread.
   a. thread()
   b. execute()
   c. run()
   d. start()
6. You achieve ________________ by starting more than one Thread object.
   a. polythreading
   b. bithreading
   c. multithreading
   d. buffered threading
7. Which of the following states is not possible for a Thread?
   a. ready
   b. finished
   c. altered
   d. new
8. A Thread's rank in terms of preferential access to the operating system's resources is its ________________.
   a. priority
   b. prerogative
   c. supremacy
   d. license

9. If you do not assign a priority to a Thread object, it assumes a priority of ________ by default.
   a. 0 (zero)
   b. 1
   c. 5
   d. 10
10. Which of the following Threads would most likely run first?
    a. a finished Thread with priority 10
    b. a suspended Thread with priority 5
    c. a runnable Thread with priority NORM_PRIORITY
    d. a runnable Thread with priority 2
11. Java sometimes chooses to run a low-priority Thread to avoid ________.
    a. construction
    b. death
    c. starvation
    d. sedation
12. You can use the Runnable ________ to inherit Thread methods.
    a. interface
    b. method
    c. mode
    d. formula
13. When you define a Thread as `private Thread someThread;`, the value of someThread is ________.
    a. zero
    b. `null`
    c. `false`
    d. unknown
14. You would see all versions of an animated drawing at the same time if you did not call an applet's ________ method.
    a. clear()
    b. paint()
    c. draw()
    d. update()

15. A technique to reduce screen flickering is to ________________.
    a. redraw the entire image with each call to the paint() method
    b. redraw only those portions of the screen that actually change
    c. call the paint() method directly instead of repaint()
    d. buy a more expensive monitor
16. To create the illusion that an object moves up and to the left, you would ________________.
    a. increase its x-coordinate and decrease its y-coordinate
    b. decrease its x-coordinate and increase its y-coordinate
    c. increase both its x- and y-coordinates
    d. decrease both its x- and y-coordinates
17. The applet method getImage() ________________.
    a. allows you to draw an image on the screen
    b. loads a stored image into an applet
    c. produces a copy of a displayed image
    d. returns the name of an image when you point to it with your mouse
18. When you use getImage(), ________________.
    a. you must create a thread in which it can run
    b. you must not create a thread in the same file
    c. Java automatically creates a thread for you
    d. as many threads are launched as there are frames within the image
19. HTML provides ________________ levels of headings.
    a. two
    b. six
    c. 12
    d. an unlimited number of
20. Image files usually use ________________ as filename extensions.
    a. .jpeg or .gif
    b. .exe or .doc
    c. .www or .url
    d. .com or .net

Exercises

1. a. Create a Thread class named Friend whose constructor accepts a friend's name. The Friend class run() method displays a single space 499 times before displaying the friend's name once. Write a program that instantiates three Threads to wihich you pass your first name and the first names of two friends. Start the Threads and observe which Thread wins the race. (When you run the program several times, a different Friend will win by different margins represented by the spaces between the names.) Save the Friend class as **Friend.java** and the program as **NameRace.java** in the Chapter.17 folder on your Student Disk.

   b. Set different priorities for the three Thread objects you created in Exercise 1a, and then run the program again. Save the new program as **NameRaceWithPriorities.java** in the Chapter.17 folder of your Student Disk.

2. Create two classes that extend Thread—LovesMeThread and LovesMeNotThread. Each Thread displays the phrase its name implies 1000 times. Write a program to start the two Threads; the final message is the answer to your question! Save the classes as **LovesMeThread.java** and **LovesMeNotThread.java** and the program as **LoveQuestion.java** in the Chapter.17 folder on your Student Disk.

3. Create a class named RaceHorse that extends Thread. Each RaceHorse has a name and a run() method that displays the name 5000 times. Write a program that instantiates two RaceHorse objects. The last RaceHorse to finish is the loser. Save the class as **RaceHorse.java** and the program that creates the RaceHorse objects as **Race.java** in the Chapter.17 folder on your Student Disk.

4. Create a CharacterThread class that displays a single character 500 times. Write a program that creates five CharacterThreads, each of which displays a different character. Give two Threads the minimum priority, two Threads the maximum priority, and one Thread the default priority. Run the program and observe the results. Save the class as **CharacterThread.java** and the program as **FiveThreads.java** in the Chapter.17 folder on your Student Disk.

5. a. Write an applet that displays a stick figure doing jumping jacks. Save the program as **Exercise.java** in the Chapter.17 folder on your Student Disk.

   b. Add text to the Exercise applet so it serves as an advertisement for a health club. Save the new program as **ExerciseAd.java** in the Chapter.17 folder on your Student Disk.

6. Write an applet that shows a bouncing ball by drawing a circle in a foreground color, redrawing it in the background color, and then drawing a new foreground-colored ball in a new position. The ball reverses direction when it nears the edge of the viewing area. Use the Thread.sleep() method so there is enough time for the user to absorb the changes on the screen before it is redrawn. Save the program as **Pong.java** in the Chapter.17 folder on your Student Disk.

7. a. Write an applet that shows a yo-yo moving up and down on its string. Each time you redraw the yo-yo, let it sleep 100 milliseconds. Create an HTML file named **TestYoYo.html** that you can run in your browser to test the class. Save the program as **YoYo.java** in the Chapter.17 folder and save both the program and the HTML file on your Student Disk.

   b. Save the YoYo.java program as **YoYo1.java**. Then create a new **YoYo2.java** file in which the yo-yo sleeps for 150 milliseconds—slightly longer than YoYo1. Create a file named **TestYoYos.html** that you can run in your browser to watch both yo-yos at once. Save all the files in the Chapter.17 folder on your Student Disk.

8. Locate a shareware animated GIF file on the Web, and then include it in an applet. Save the program as **MyMovie.java** in the Chapter.17 folder on your Student Disk.

9. Write an applet that creates a straight line that constantly rotates in a circle. Save the program as **Baton.java** in the Chapter.17 folder on your Student Disk.

10. Write an applet that simulates a marquee by displaying a String of characters one-at-a-time from right-to-left across the screen. Use the Thread.sleep() method to pause momentarily before each new character displays. When the String message is fully displayed, start the message again. Save the program as **Marquee.java** in the Chapter.17 folder on your Student Disk.

11. Each of the following files in the Chapter.17 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugSeventeen1.java will become FixDebugSeventeen1.java. You can use the files testDebug3.html and testDebug4.html in the Chapter.17 folder of your Student Disk to test the applets in DebugSeventeen3.java and DebugSeventeen4.java, respectively. There are five additional files accompanying the Debug files. DebugSeventeen1.java uses DebugThreadA.java and DebugThreadB.java. DebugSeventeen2.java uses DebugSeventeen2Thread.java. You can use the html files TestDebug3.html and TestDebug4.html to run DebugSeventeen3.java and DebugSeventeen4.java, respectively.

    a. DebugSeventeen1.java

    b. DebugSeventeen2.java

    c. DebugSeventeen3.java

    d. DebugSeventeen4.java

CASE PROJECT

Case Project

The Healthy Heart Association is sponsoring a walk-a-thon to raise money and wants an animated figure to display on its Web site. Write an applet that displays a figure that walks from left-to-right across the screen. When the figure nears the right side of the screen, start the figure at the left again. Save the program as **Walking.java**. Write an HTML document named **TestWalkers.html** that displays three walking figures and includes text describing a fund-raising walk-a-thon. Save both files in the Chapter.17 folder on your Student Disk.

APPENDIX

A

WORKING WITH JAVA SDK 1.4

In this Appendix, you will:

- ♦ Configure the Java System Development Kit
- ♦ Use Notepad to save and edit source code
- ♦ Use the DOS prompt to compile source code
- ♦ Use TextPad to work with Java
- ♦ Use TextPad to save and edit source code

CONFIGURING WINDOWS TO WORK WITH THE SDK

To configure Windows with the SDK that accompanies this book, you must add the Java bin directory to the command path of your operating system (OS). That way, your OS will know where to look for the Java commands that you use.

One way to update the OS path for Windows 95 and 98 is to use a procedure to edit the OS Path or set the OS Path command in the autoexec.bat file. This is the file that is automatically executed every time you start your computer. After you edit or set the command in the file, to test whether the path is set correctly to include the bin directory, you can type autoexec.bat at the command prompt.

How to set the path for Windows 95 or 98:

1. Go to the Start menu and select the **Run** option.
2. In the Run dialog box, type **`sysedit`**, and then click **OK**. This should start the System Configuration Editor.
3. If necessary, use the Window menu to switch to the AUTOEXEC.BAT file.
4. If the file contains a PATH or SET PATH command, type a **;** (semicolon) at the end of the command, then type **`C:\j2sdk1.4.0\bin\`**, followed by another **;** (semicolon), then a **.** (period), and finally an ending **;** (semicolon). The command should look something like the following:

   ```
   PATH = C:\WINDOWS;C:\WINDOWS\COMMAND;C:\jdk1.4.1\bin\;.;
   ```

5. If the file does not contain a PATH command, enter:
   `PATH = C:\j2sdk1.4.0\bin\;.;` at the beginning of the file.

6. So that the OS can find your class files that are in the Java bin directory and also on your hard disk or floppy disk, add the CLASSPATH command to the AUTOEXEC.BAT file. Enter the command SET CLASSPATH on a separate line in the AUTOEXEC.BAT file: **`SET CLASSPATH = C:\j2sdk1.4.0\bin\;.;`**.
7. The previous steps assume you are using JDK1.4.0. If you are using JDK1.3.1 or JDK1.2, substitute that name in the previous steps.
8. Save the file and exit the System Configuration Editor.
9. Reboot your system to have the new path take effect.

How to set the path for Windows NT, Windows 2000, or Windows XP:

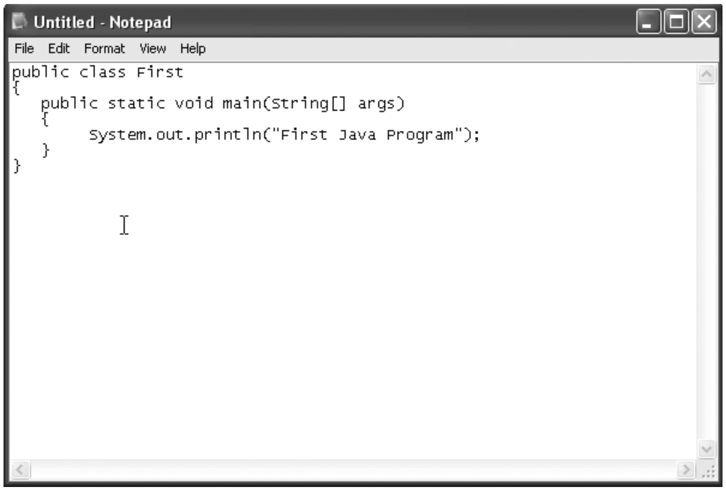
1. Open the Start menu and click **Control Panel**. For Windows 2000 and Windows NT users, the Control Panel is found under Settings on the Start menu.
2. On the Control Panel, double-click the **System** icon. On System Properties, click the **Advanced** tab. (If you are using Windows XP, you will need to switch to Classic View.)
3. Click the **Environment Variables** button.
4. In the bottom pane labeled System variables, if necessary scroll to find the Path statement.
5. If the file contains a Path or SET PATH command, click to select the **Path** variable, and then click the **Edit** button. A new Edit System Variable window opens and the existing Path command can be edited. At the end of the existing Path command, type a **;** (semicolon), and then type **`C:\j2sdk1.4.0\bin\`**, followed by another **;** (semicolon), then type a **.** (period), and finally type an ending **;** (semicolon). The command should look something like the following: `PATH` (in the top window specifying the command name), and `%SystemRoot%\system32;C:\j2sdk1.4.0\bin;.;` in the bottom Edit System variable window.
6. If no such PATH command exists, click the **New** button and enter **`PATH`** as the Variable Name and **`C:\j2sdk1.4.0\bin\;.;`** as the Variable Value.
7. So that the OS can find your class files that are in the Java bin directory and also on your hard disk or floppy disk, add the CLASSPATH command to the AUTOEXEC.BAT file. Enter the command **`SET CLASSPATH`** on a separate line in the AUTOEXEC.BAT file by clicking the **New** button, then enter **`CLASSPATH`** as the Variable Name, and **`SET CLASSPATH = C:\j2sdk1.4.0\bin\;.;`** as the Variable Value.
8. The previous steps assume you are using JDK1.4.0. If you are using JDK1.3.1 or JDK1.2, substitute that name in the previous steps.

9. Save the file and exit.
10. Reboot your system to have the new path take effect.

Using Notepad to Save and Edit Source Code

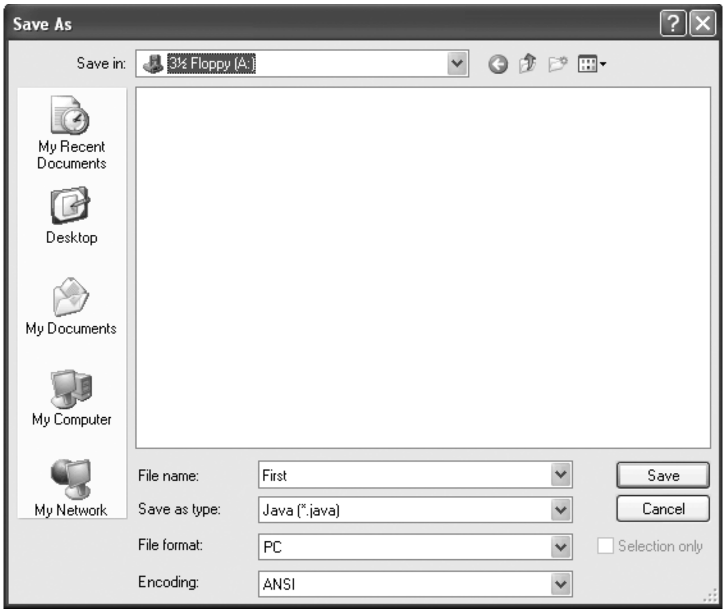
Figure A1-1 shows how to use the Windows Notepad text editor to save and edit the source code for Java Programs. To start Notepad using Windows 95/98/2000, click the Start menu, select the Program folder, select Accessories, and then select Notepad. To start Notepad using Windows XP, click the Start menu, click All Programs, select Accessories, and then select Notepad. After you start Notepad, you can enter and edit the code just as you would with any text editor.

Saving source code in Notepad requires that the Java source file be saved with a .java extension. Because Java is case sensitive, you must save a file with the proper capitalization. If the class name of the file and the filename do not match, in both spelling and case, you will receive an error when you attempt to compile the source code. Because the class name in Figure A1-1 is "First" you must save the file as "First.java". If you use TextPad to create your files, you can select Java as a Save As type as shown in Figure A1-2. If you use Notepad, one way to ensure that the Java file is saved in the proper format is to place double quotation marks around the filename, as in "First.java". This ensures that the file is not saved as "First.java.txt".



```
public class First
{
   public static void main(String[] args)
   {
        System.out.println("First Java Program");
   }
}
```

Figure A1-1 First.java as it appears in Notepad using Windows XP



File name: First
Save as type: Java (*.java)
File format: PC
Encoding: ANSI

Figure A1-2 First.java shown correctly in Save As dialog box

USING THE DOS PROMPT TO COMPILE SOURCE CODE

Figure A1-3 shows how to use the DOS prompt, or command prompt, to compile and run applications. Finding the command prompt varies depending on your version of Windows as follows:

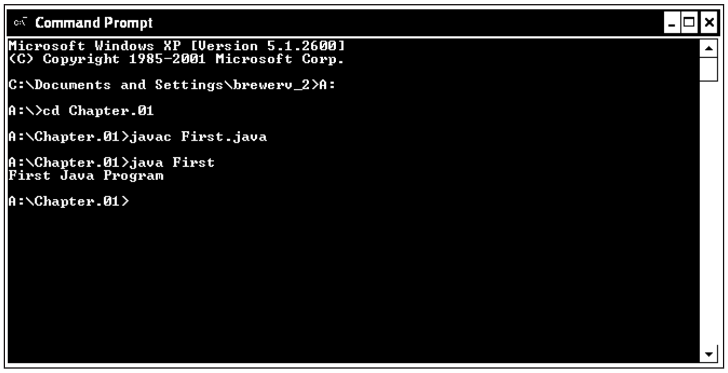
- Windows 95 and 98; click Start, click Programs, and then click MS-DOS Prompt
- Windows 2000: Click Start, click Programs, click Accessories, and then click Command Prompt
- Windows XP: Click Start, click All Programs, click Accessories, and then click Command prompt

At the command prompt, change from the default drive prompt to the drive where your application is stored. For example you might type `A:` and press [Enter] to change to the A prompt as shown in Figure A1-3, then change the directory (or folder) to the directory that holds your application using the `cd` (change directory) command. For example, you might type `cd Chapter.01` as shown in Figure A1-3.

To compile an application, you type the `javac` command to start the Java compiler. The command `javac` is entered, followed by a space and the complete name of the .java file—in this case First.java. If the application doesn't compile successfully, the PATH might not be set correctly to the Java SDK bin directory where the javac.exe file is located. Also, you might have failed to use the same capitalization as the name of the Java class name.

When you compile a .java file correctly, the Java compiler creates a .class file that has the same filename as the .java file. Thus, a successful compilation of the First.java file creates a First.class.

To run a Java program, you use the `java` command and the class name without the .class extension. For example, after a program named First.java is compiled producing First.class, you execute the program using the command `java First` as shown in Figure A1-3. When the program ends, control is returned to the command prompt. If a program does not end on its own, or you want to end it prematurely, you can press `[Ctrl]+C` to return to the command prompt.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\brewerv_2>A:

A:\>cd Chapter.01

A:\Chapter.01>javac First.java

A:\Chapter.01>java First
First Java Program

A:\Chapter.01>
```

Figure A1-3 Commands for compiling and running the First.java program

Using TextPad to Work with Java

Now that you've learned how to use Notepad and the DOS prompt for working with Java, you're ready to learn how to use TextPad. Because this text editor is designed for working with Java, many programmers prefer it to Notepad. You can download a trial version from *http://www.textpad.com*.

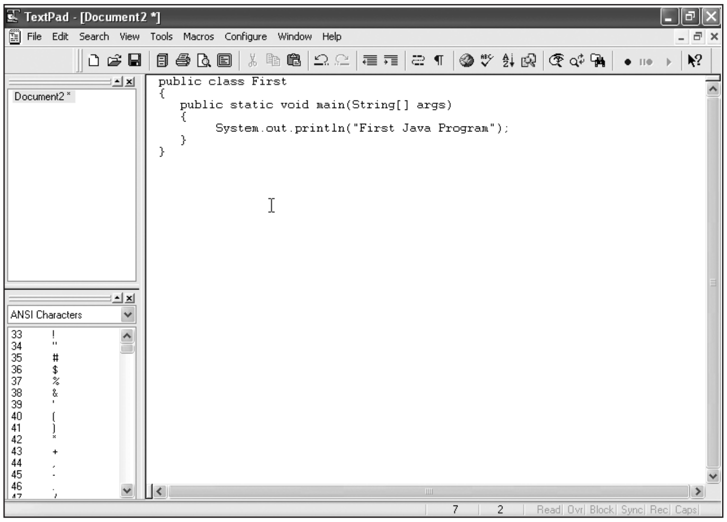
Unlike Notepad, TextPad is not included with Windows, and you must install it before you can use it. To do so, run the setup file that you can download from the TextPad Web site. Then respond to the resulting dialog boxes. Because this is a trial version of TextPad, you should purchase TextPad if you decide to use it beyond the initial trial period. TextPad is relatively inexpensive, especially when you consider how much time and effort it can save you.

Note that TextPad runs only under the Windows operating system. If you are not using Windows, you can use the text editor that comes with your operating system or you can search the Web to find a text editor that better suits your needs.

Using TextPad to Save and Edit Source Code

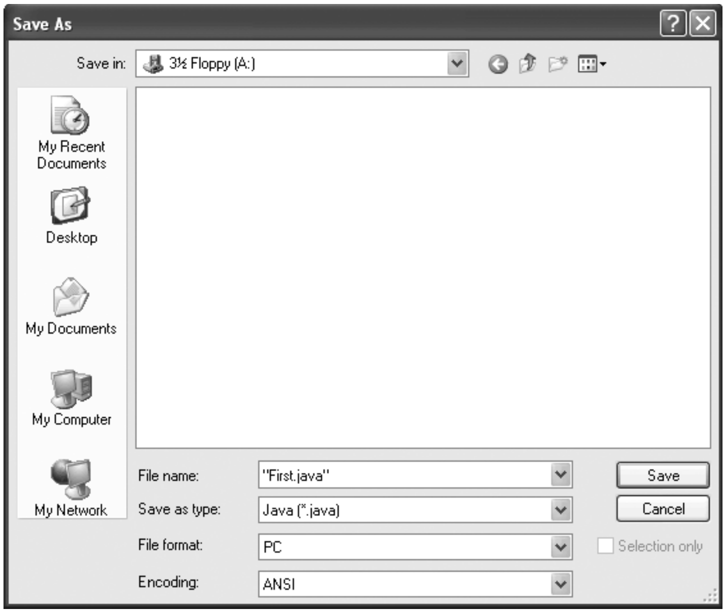
Figures A1-4 and A1-5 show how to use TextPad to create, edit, and save source code. To enter and edit source code, you can use the same techniques that you use when working with any other Windows text editor. In short, you can use the standard Windows shortcut keystrokes and menus to enter, edit, and save your code. You can use the File menu to open and close files. You can use the Edit menu to cut, copy, and paste text. And you can use the Search menu to find and replace text. In addition, TextPad color codes the source files so it is easier to recognize the Java syntax. TextPad also makes it easier to save *.java files with the proper capitalization and extension.

To save the source code, select the Save command from the File menu ([Ctrl]+S). Enter the filename so it is exactly the same as the class name, and then select the Java option from the Save As type list so TextPad adds the four-letter Java extension to the filename. (If you are using earlier versions of Windows, you may need to enter the four-letter extension with the filename, as in First.java. Otherwise, the extension will be truncated to jav.)


TextPad - [Document2 *]
File Edit Search View Tools Macros Configure Window Help
Document2 *
ANSI Characters
7 2 Read Ovr Block Sync Rec Caps

```
public class First
{
   public static void main(String[] args)
   {
        System.out.println("First Java Program");
   }
}
```

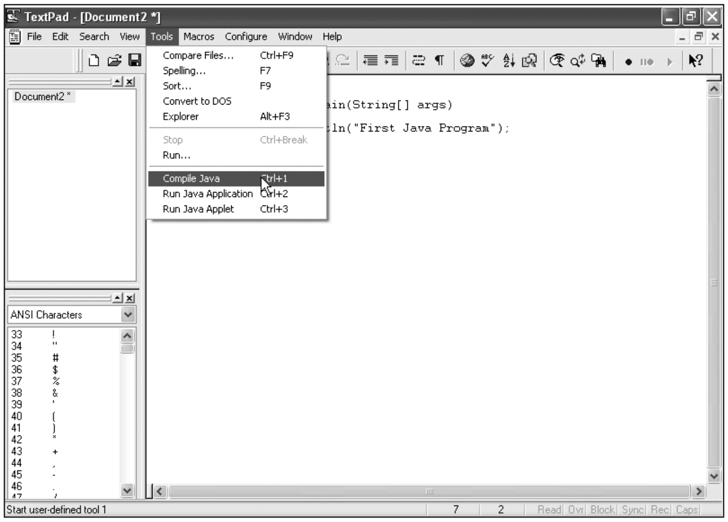
Figure A1-4 Creating and Editing First.java with TextPad



File name: "First.java"
Save as type: Java (*.java)
File format: PC
Encoding: ANSI

Figure A1-5 The TextPad Save As dialog box

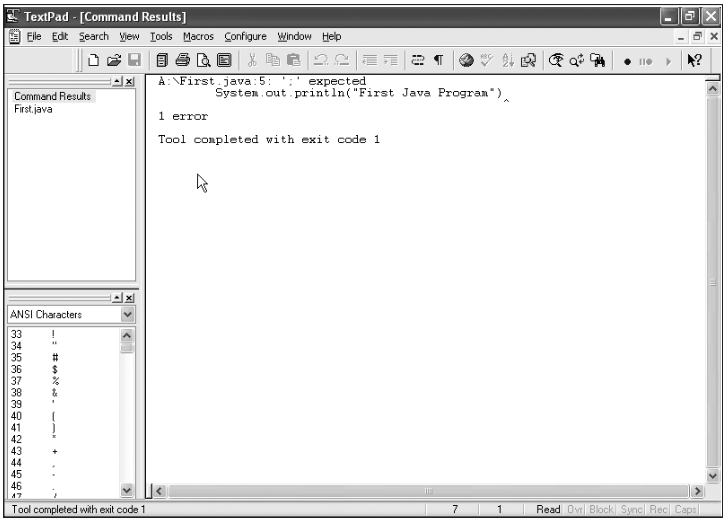
Figure A1-6 shows how to use TextPad to compile the source code for a Java application. To compile the current source code, press **[Ctrl]+1** or select the Compile Java command from the Tools menu.



```
ain(String[] args)
tln("First Java Program");
```

Figure A1-6 Compiling source code using the Tools Menu

If the source code does not compile cleanly, TextPad will leave you at a Command Results window like the one shown in Figure A1-7. In this case, you can read the error message, switch to the source code window, correct the error, and compile the source code again. Because each error message identifies the line number of the error, it is often easier to find the error by selecting the Line Number option from the View menu. That way, TextPad will display line numbers.


TextPad - [Command Results]
File Edit Search View Tools Macros Configure Window Help
Command Results
First.java
ANSI Characters
Tool completed with exit code 1
7
1
Read Ovr Block Sync Rec Caps

```
A:\First.java:5: ';' expected
        System.out.println("First Java Program")
                                                ^
1 error

Tool completed with exit code 1
```

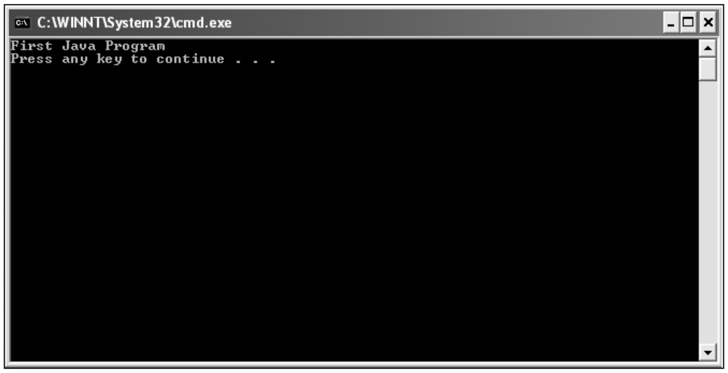
Figure A1-7 A compile time error

When you have several Java files open at once, you can use the document selector pane to switch between files. In Figure A1-7, only two documents are open (First and Command Results), but you can open as many Java files as you like. You can also use the Windows menu and standard Windows keystrokes ([Ctrl]+F6 and [Ctrl]+Shift+F6) to switch between windows.

To edit as efficiently as possible, you can use the Document Properties command in the View menu to set formatting options. In particular, you should set the tab settings so you can easily align the code in a program.

How to Use TextPad to Run an Application

Once you've completed the source code for an application, you can run that application by pressing **[Ctrl]+2** or by clicking **Run Java Application** from the Tools menu. If the application that you run prints text to the console, TextPad will start a DOS Prompt window like the one shown in Figure A1-8. Then you can press any key to end the application. If necessary, you can also click the **Close** button or press **[Alt]+F4** to close the DOS Prompt window.



```
First Java Program
Press any key to continue . . .
```

Figure A1-8 Text printed to the console

Index

Java Programming

Java is a popular programming language among professional programmers because it can be used to build visually interesting GUI and Web-based applications. This book, Java Programming, provides the student with a guide to developing applications and applets using Java.

The textbook assumes little or no programming experience. It is written in a non-technical style and emphasises good programming practice. It also provides a solid background in good object-oriented programming techniques and introduces the student to object-oriented terminology using clear, familiar language.

Particular features of the book include:

- Each chapter begins with a list of objectives so that the student knows the topics that will be covered
- Tips are included that provide additional information, such as an alternative method of performing a procedure or a common error to avoid
- Each chapter includes a summary that recaps the programming concepts and techniques that have been covered
- Review questions are included in an end-of-chapter assessment in order to reinforce the main ideas introduced in each chapter
- Each chapter concludes with a programming exercise that provides additional practice of the skills and concepts learned in that chapter

The author, Joyce Farrell, is Assistant Professor of Computer Information Systems at Harper College in Palatine, Illinois, USA. She is the author of three other programming texts.

Internet
http://www.nccedu.com

ISBN 0-9543071-1-9

9 780954 307110