

6

Methods

Objectives

- To understand how to construct programs modularly from small pieces called *methods*.
- To introduce the common math methods available in the Java API.
- To be able to create new methods.
- To understand the mechanisms for passing information between methods.
- To introduce simulation techniques that use random-number generation.
- To understand how the visibility of identifiers is limited to specific regions of programs.
- To understand how to write and use methods that call themselves.

Form ever follows function.

Louis Henri Sullivan

E pluribus unum.

(One composed of many.)

Virgil

O! call back yesterday, bid time return.

William Shakespeare, *Richard II*

Call me Ishmael.

Herman Melville, *Moby Dick*

When you call me that, smile.

Owen Wister



Outline

- 6.1 Introduction
- 6.2 Program Modules in Java
- 6.3 `Math` Class Methods
- 6.4 Methods
- 6.5 Method Definitions
- 6.6 Argument Promotion
- 6.7 Java API Packages
- 6.8 Random-Number Generation
- 6.9 Example: A Game of Chance
- 6.10 Duration of Identifiers
- 6.11 Scope Rules
- 6.12 Recursion
- 6.13 Example Using Recursion: The Fibonacci Series
- 6.14 Recursion vs. Iteration
- 6.15 Method Overloading
- 6.16 Methods of Class `JApplet`
- 6.17 (Optional Case Study) Thinking About Objects: Identifying Class Operations

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

6.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this text. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or *modules*. This technique is called *divide and conquer*. This chapter describes many key features of the Java language that facilitate the design, implementation, operation and maintenance of large programs.

6.2 Program Modules in Java

Modules in Java are called *methods* and *classes*. Java programs are written by combining new methods and classes the programmer writes with “prepackaged” methods and classes available in the *Java API* (also referred to as the *Java class library*) and in various other method and class libraries. In this chapter, we concentrate on methods; we begin to discuss classes in detail in Chapter 8.

The Java API provides a rich collection of classes and methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, error checking and many other useful operations. This set of modules makes the programmer’s job easier, because the modules provide many of the capabilities

programmers need. The Java API methods are provided as part of the Java 2 Software Development Kit (J2SDK).



Good Programming Practice 6.1

Familiarize yourself with the rich collection of classes and methods in the Java API and with the rich collections of classes available in various class libraries.



Software Engineering Observation 6.1

Avoid reinventing the wheel. When possible, use Java API classes and methods instead of writing new classes and methods. This reduces program development time and avoids introducing programming errors.



Performance Tip 6.1

Do not try to rewrite existing Java API classes and methods to make them more efficient. You usually will not be able to increase the performance of these classes and methods.

The programmer can write methods to define specific tasks that a program may use many times during its execution. These are sometimes referred to as *programmer-defined methods*. The actual statements defining the methods are written only once and are hidden from other methods.

A method is *invoked* (i.e., made to perform its designated task) by a *method call*. The method call specifies the name of the method and provides information (as *arguments*) that the called method requires to perform its task. When the method call completes, the method either returns a result to the *calling method* (or *caller*) or simply returns control to the calling method. A common analogy for this program structure is the hierarchical form of management. A boss (the calling method, or caller) asks a worker (the *called method*) to perform a task and report back (i.e., *return*) the results after completing the task. The boss method does not know *how* the worker method performs its designated tasks. The worker may also call other worker methods, and the boss will be unaware of this occurrence. We will soon see how this “hiding” of implementation details promotes good software engineering. Figure 6.1 shows the **boss** method communicating with several worker methods in a hierarchical manner. Note that **worker1** acts as a “boss method” to **worker4** and **worker5**. Relationships among methods may also be different than the hierarchical structure shown in this figure.

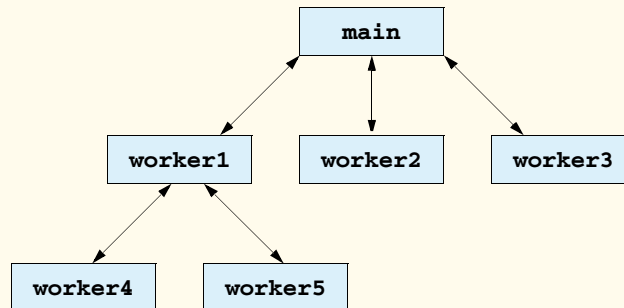


Fig. 6.1 Hierarchical boss-method/worker-method relationship.

6.3 Math Class Methods

Math class methods allow the programmer to perform certain common mathematical calculations. We use various **Math** class methods here to introduce the concept of methods. Throughout the book, we discuss many other methods from the classes of the Java API.

Methods are called by writing the name of the method, followed by a left parenthesis, followed by the *argument* (or a comma-separated list of arguments) of the method, followed by a right parenthesis. For example, a programmer desiring to calculate the square root of **900.0** might write

```
Math.sqrt( 900.0 )
```

When this statement executes, it calls **static Math** method **sqrt** to calculate the square root of the number contained in the parentheses (**900.0**). The number **900.0** is the *argument* of method **sqrt**. The preceding expression evaluates to **30.0**. Method **sqrt** method takes an argument of type **double** and returns a result of type **double**. Note that all **Math** class methods are **static**; therefore, they are invoked by preceding the name of the method with the class name **Math** and a dot (.) operator. To output the value of the preceding method call in the command window, a programmer might write

```
System.out.println( Math.sqrt( 900.0 ) );
```

In this statement, the value that **sqrt** returns becomes the argument to method **println**.



Software Engineering Observation 6.2

*It is not necessary to import class **Math** to use its methods. **Math** is part of the **java.lang** package, which is automatically imported by the compiler.*



Common Programming Error 6.1

*Forgetting to invoke a **Math** class method by preceding the name of the method with the class name **Math** and a dot operator (.) results in a syntax error.*

Method arguments may be constants, variables or expressions. If **c1 = 13.0**, **d = 3.0** and **f = 4.0**, then the statement

```
System.out.println( Math.sqrt( c1 + d * f ) );
```

calculates and prints the square root of **13.0 + 3.0 * 4.0 = 25.0**, namely **5.0**.

Some **Math** class methods are summarized in Fig. 6.2. In the figure, the variables **x** and **y** are of type **double**. The **Math** class also defines two commonly used mathematical constants: **Math.PI** and **Math.E**. The constant **Math.PI** (3.14159265358979323846) of class **Math** is the ratio of a circle's circumference to its diameter. The constant **Math.E** (2.7182818284590452354) is the base value for natural logarithms (calculated with static **Math** method **log**).

6.4 Methods

Methods allow the programmer to modularize a program. Variables declared in method definitions are *local variables*—only the method that defines them knows they exist. Most methods have a list of *parameters* that provide the means for communicating information between methods via method calls. A method's parameters are also local variables.

Method	Description	Example
abs(x)	absolute value of x (this method also has versions for float , int and long values)	abs(23.7) is 23.7 abs(0.0) is 0.0 abs(-23.7) is 23.7
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of x (x is in radians)	cos(0.0) is 1.0
exp(x)	exponential method e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is -10.0
log(x)	natural logarithm of x (base e)	log(2.718282) is 1.0 log(7.389056) is 2.0
max(x, y)	larger value of x and y (this method also has versions for float , int and long values)	max(2.3, 12.7) is 12.7 max(-2.3, -12.7) is -2.3
min(x, y)	smaller value of x and y (this method also has versions for float , int and long values)	min(2.3, 12.7) is 2.3 min(-2.3, -12.7) is -12.7
pow(x, y)	x raised to power y (x^y)	pow(2.0, 7.0) is 128.0 pow(9.0, .5) is 3.0
sin(x)	trigonometric sine of x (x is in radians)	sin(0.0) is 0.0
sqrt(x)	square root of x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	trigonometric tangent of x (x is in radians)	tan(0.0) is 0.0

Fig. 6.2 **Math** class methods.

There are several motivations for modularizing a program with methods. The divide-and-conquer approach makes program development more manageable. Another motivation is *software reusability*—using existing methods as building blocks to create new programs. With good method naming and definition, you can create programs from standardized methods rather than by building customized code. For example, we did not have to define how to convert **Strings** to integers and floating-point numbers; Java already provides such methods for us in class **Integer** (**parseInt**) and class **Double** (**parseDouble**). A third motivation is to avoid repeating code in a program. Packaging code as a method allows a program to execute that code from several locations in a program simply by calling the method.



Software Engineering Observation 6.3

To promote software reusability, each method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.



Software Engineering Observation 6.4

If you cannot choose a concise name that expresses a method's task, it is possible that your method is attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller method definitions.

6.5 Method Definitions

The programs presented up to this point each consisted of a class definition containing at least one method definition that called Java API methods to accomplish its tasks. We now consider how programmers write their own customized methods. Until we discuss more of the details of class definitions in Chapter 8, we use applets for all programs that contain two or more method definitions, for simplicity.

Consider an applet that uses a method **square** (invoked from the applet's **init** method) to calculate the squares of the integers from 1 to 10 (Fig. 6.3).

When the applet begins execution, the applet container calls the applet's **init** method. Line 16 declares **JTextArea** reference **outputArea** and initializes it with a new **JTextArea**. This **JTextArea** will display the program's results.

This program is the first in which we display a GUI component on an applet. The on-screen display area for a **JApplet** has a *content pane*, to which the GUI components must be attached so they can be displayed at execution time. The content pane is an object of class **Container** from the *java.awt* package. This class was **imported** on line 5 for use in the applet. Line 19 declares **Container** reference **container** and assigns to it the result of a call to method **getContentPane**—one of the many methods that our class **SquareInt** inherits from class **JApplet**. Method **getContentPane** returns a reference to the applet's content pane. The program uses that reference to attach GUI components, like a **JTextArea**, to the applet's user interface.

Line 22 places the **JTextArea** GUI component object to which **outputArea** refers on the applet. When the applet executes, any GUI components attached to it are displayed. **Container** method **add** attaches a GUI component to a container. For the moment, we can attach only one GUI component to the applet's content pane, and that GUI component will occupy the applet's entire drawing area on the screen (as defined by the **width** and **height** of the applet, in pixels, in the applet's HTML document). Later, we will discuss how to attach many GUI components to an applet by changing the applet's *layout*. The layout controls how the applet positions GUI components in its area on the screen.

Line 24 declares **int** variable **result** to store the result of each square calculation. Line 25 declares **String** reference **output** and initializes it with the empty string. This **String** will contain the results of squaring the values from 1 to 10. Lines 28–37 define a **for** repetition structure. Each iteration of this loop calculates the **square** of the current value of control variable **x**, stores the value in **result** and concatenates the **result** to the end of **output**.

The applet invokes (or calls) its **square** method on line 31 with the statement

```
result = square( counter );
```

The **()** after **square** represent the *method-call operator*, which has high precedence. At this point, the program makes a copy of the value of **counter** (the *argument* to the method call) and transfers program control to the first line of method **square** (defined at lines 44–

48). Method **square** receives the copy of the value of **counter** in the *parameter y*. Then, **square** calculates **y * y** (line 46). Method **square** uses a **return** statement to return (i.e., give back) the result of the calculation to the statement in **init** that invoked **square**. In method **init**, the return value is assigned to variable **result**. Lines 34–35 concatenate "The square of ", the value of **counter**, " is ", the value of **result** and a newline character to the end of **output**. This process repeats for each iteration of the **for** repetition structure. Line 39 uses method **setText** to set **outputArea**'s text to the **String output**.

```

1  // Fig. 6.3: SquareIntegers.java
2  // A programmer-defined square method
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class SquareIntegers extends JApplet {
11
12     // set up GUI and calculate squares of integers from 1 to 10
13     public void init()
14     {
15         // JTextArea to display results
16         JTextArea outputArea = new JTextArea();
17
18         // get applet's content pane (GUI component display area)
19         Container container = getContentPane();
20
21         // attach outputArea to container
22         container.add( outputArea );
23
24         int result; // store result of call to method square
25         String output = ""; // String containing results
26
27         // loop 10 times
28         for ( int counter = 1; counter <= 10; counter++ ) {
29
30             // calculate square of counter and store in result
31             result = square( counter );
32
33             // append result to String output
34             output += "The square of " + counter +
35                     " is " + result + "\n";
36
37         } // end for structure
38
39         outputArea.setText( output ); // place results in JTextArea
40
41     } // end method init
42

```

Fig. 6.3 Using programmer-defined method **square** (part 1 of 2).

```

43 // square method definition
44 public int square( int y )
45 {
46     return y * y; // return square of y
47
48 } // end method square
49
50 } // end class SquareIntegers

```

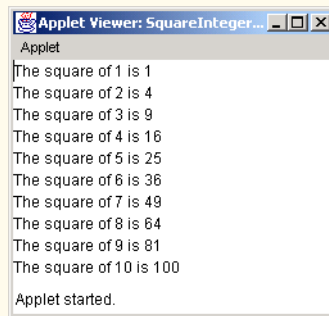


Fig. 6.3 Using programmer-defined method **square** (part 2 of 2).

Note that we declared references **output**, **outputArea** and **container** and variable **result** as local variables in **init**, because they are used only in **init**. Variables should be declared as instance variables only if they are required for use in more than one method of the class or if the program should save their values between calls to the class's methods. Also, note that method **init** calls method **square** directly without preceding the method name with a class name and a dot operator or a reference name and a dot operator. Each method in a class is able to call the class's other methods directly. However, there is an exception to this rule. A class's **static** methods can call only other **static** methods of the class directly. Chapter 8 discusses **static** methods in detail.

The definition of method **square** (line 44) shows that **square** expects an integer parameter **y**; **square** uses this name to manipulate the value it receives. Keyword **int** preceding the name of the method indicates that **square** returns an integer result. The **return** statement in **square** passes the result of the calculation **y * y** back to the calling method. Note that the entire method definition appears between the braces of the class **SquareInt**. All methods must be defined inside a class definition.



Good Programming Practice 6.2

Place a blank line between method definitions to separate the methods and enhance program readability.



Common Programming Error 6.2

Defining a method outside the braces of a class definition is a syntax error.

The general format of a method definition is

```
return-value-type method-name( parameter-list )
{
    declarations and statements
}
```

The *method-name* is any valid identifier. The *return-value-type* is the data type of the result returned from the method to the caller. The *return-value-type* **void** indicates that a method does not return a value. Methods can return at most one value.

The *parameter-list* is a comma-separated list in which the method declares each parameter's type and name. There must be one argument in the method call for each parameter in the method definition. Each argument also must be compatible with the type of the corresponding parameter in the method definition. For example, a parameter of type **double** can receive values of 7.35, 22 or -0.03546, but not **"hello"** (because a **String** cannot be assigned to a **double** variable). If a method does not receive any values, the *parameter-list* is empty (i.e., the name of the method is followed by an empty set of parentheses). Each parameter in the parameter list of a method must be declared with a data type; otherwise, a syntax error occurs.

Following the first line of the method definition (also known as the *method header*), *declarations and statements* in braces form the *method body*. The method body is also referred to as a block. Variables can be declared in any block, and blocks can be nested. A method cannot be defined inside another method.

There are three ways to return control to the statement that invoked a method. If the method does not return a result, control returns when the program flow reaches the method-ending right brace or when the statement

```
return;
```

is executed. If the method returns a result, the statement

```
return expression;
```

evaluates the *expression*, then returns the resulting value to the caller. When a **return** statement executes, control returns immediately to the statement that invoked the method.

Note that the example in Fig. 6.3 actually contains two method definitions—**init** (lines 13–41) and **square** (line 44–48). Remember that the applet container calls method **init** to initialize the applet. In this example, method **init** repeatedly invokes the **square** method to perform a calculation, then places the results in the **JTextArea** that is attached to the applet's content pane. When the applet appears on the screen, the results are displayed in the **JTextArea**.

Notice the syntax used to invoke method **square**—we use just the name of the method, followed by the arguments to the method in parentheses. Methods in a class definition are allowed to invoke other methods in the same class definition by using this syntax. (There is an exception to this rule, discussed in Chapter 8.) Methods in the same class definition are both the methods defined in that class and the inherited methods (the methods from the class that the current class **extends**—**JApplet** in Fig. 6.3). We have now seen three ways to call a method: A method name by itself (as shown with **square(x)** in this example), a reference to an object followed by the dot (.) operator and the method name

(such as `g.drawLine( x1, y1, x2, y2 )`), and a class name followed by a method name (such as `Integer.parseInt( stringToConvert )`). The last syntax is only for **static** methods of a class (discussed in detail in Chapter 8).



Common Programming Error 6.3

Omitting the return-value-type in a method definition is a syntax error.



Common Programming Error 6.4

*Forgetting to return a value from a method that should return a value is a syntax error. If a return value type other than **void** is specified, the method must contain a **return** statement.*



Common Programming Error 6.5

*Returning a value from a method whose return type has been declared **void** is a syntax error.*



Common Programming Error 6.6

*Declaring method parameters of the same type as **float x, y** instead of **float x, float y** is a syntax error, because types are required for each parameter in the parameter list.*



Common Programming Error 6.7

Placing a semicolon after the right parenthesis enclosing the parameter list of a method definition is a syntax error.



Common Programming Error 6.8

Redefining a method parameter in the method's body is a syntax error.



Common Programming Error 6.9

Passing to a method an argument that is not compatible with the corresponding parameter's type is a syntax error.



Common Programming Error 6.10

Defining a method inside another method is a syntax error.



Good Programming Practice 6.3

Avoid using the same names for instance variables and local variables. This helps readers of your program distinguish variables used in different parts of a class definition.



Good Programming Practice 6.4

Choosing meaningful method names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.



Software Engineering Observation 6.5

A method should usually be no longer than one page. Better yet, a method should usually be no longer than half a page. Regardless of how long a method is, it should perform one task well. Small methods promote software reusability.



Software Engineering Observation 6.6

Programs should be written as collections of small methods. This makes programs easier to write, debug, maintain and modify.

**Software Engineering Observation 6.7**

A method requiring a large number of parameters may be performing too many tasks. Consider dividing the method into smaller methods that perform the separate tasks. The method header should fit on one line if possible.

**Software Engineering Observation 6.8**

The method header and method calls must all agree in the number, type and order of parameters and arguments.

**Testing and Debugging Tip 6.1**

Small methods are easier to test, debug and understand than large ones.

The applet in our next example (Fig. 6.4) uses a programmer-defined method called **maximum** to determine and return the largest of three floating-point values.

```

1  // Fig. 6.4: Maximum.java
2  // Finding the maximum of three doubles
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class Maximum extends JApplet {
11
12     // initialize applet by obtaining user input and creating GUI
13     public void init()
14     {
15         // obtain user input
16         String s1 = JOptionPane.showInputDialog(
17             "Enter first floating-point value" );
18         String s2 = JOptionPane.showInputDialog(
19             "Enter second floating-point value" );
20         String s3 = JOptionPane.showInputDialog(
21             "Enter third floating-point value" );
22
23         // convert user input to double values
24         double number1 = Double.parseDouble( s1 );
25         double number2 = Double.parseDouble( s2 );
26         double number3 = Double.parseDouble( s3 );
27
28         // call method maximum to determine largest value
29         double max = maximum( number1, number2, number3 );
30
31         // create JTextArea to display results
32         JTextArea outputArea = new JTextArea();
33

```

Fig. 6.4 Programmer-defined **maximum** method (part 1 of 2).

```

34      // display numbers and maximum value
35      outputArea.setText( "number1: " + number1 +
36                          "\nnumber2: " + number2 + "\nnumber3: " + number3 +
37                          "\nmaximum is: " + max );
38
39      // get the applet's GUI component display area
40      Container container = getContentPane();
41
42      // attach outputArea to Container c
43      container.add( outputArea );
44
45  } // end method init
46
47      // maximum method uses Math class method max to help
48      // determine maximum value
49      public double maximum( double x, double y, double z )
50      {
51          return Math.max( x, Math.max( y, z ) );
52
53      } // end method maximum
54
55  } // end class Maximum

```

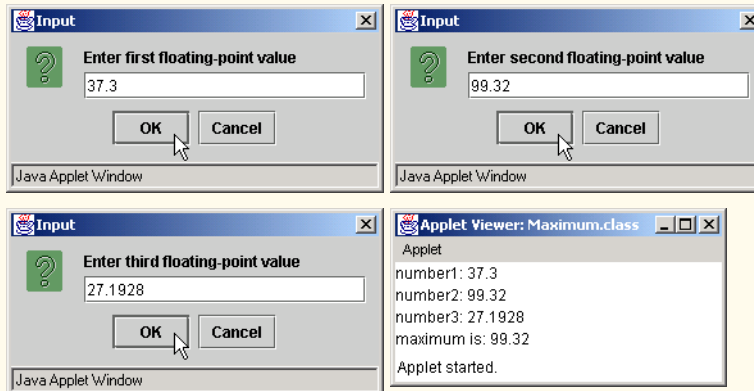


Fig. 6.4 Programmer-defined `maximum` method (part 2 of 2).

The three floating-point values are input by the user via input dialogs (lines 16–21 of `init`). Lines 24–26 use method `Double.parseDouble` to convert the `Strings` input by the user to `double` values. Line 29 calls method `maximum` (defined on lines 49–53) to determine the largest `double` value of the three `double` values passed as arguments to the method. Method `maximum` returns the result to method `init`, using a `return` statement. The program assigns the result to variable `max`. Lines 35–37 use `String` concatenation to form a `String` containing the three `double` values input by the user and the `max` value and place the result in `JTextArea` `outputArea`.

Notice the implementation of the method `maximum` (lines 49–53). The first line indicates that the method returns a `double` floating-point value, that the method's name is `maximum` and that the method takes three `double` parameters (`x`, `y` and `z`) to accomplish its task. Also, the statement (line 51) in the body of the method returns the largest of the

three floating-point values, using two calls to the **Math.max** method. First, the statement invokes method **Math.max** with the values of variables **y** and **z** to determine the larger of the two values. Next, the statement passes the value of variable **x** and the result of the first call to **Math.max** to method **Math.max**. Finally, the statement returns the result of the second call to **Math.max** to line 29 (the point at which method **init** invoked method **maximum**).

6.6 Argument Promotion

Another important feature of method definitions is the *coercion of arguments* (i.e., the forcing of arguments to the appropriate type to pass to a method). For example, a program can call **Math** method **sqrt** with an integer argument even though the method expects to receive a **double** argument. For example, the statement

```
System.out.println( Math.sqrt( 4 ) );
```

correctly evaluates **Math.sqrt(4)** and prints the value **2**. The method definition's parameter list causes Java to convert the integer value **4** to the **double** value **4.0** before passing the value to **sqrt**. In some cases, attempting these conversions leads to compiler errors if Java's *promotion rules* are not satisfied. The promotion rules specify how to convert types to other types without losing data. In our **sqrt** example above, an **int** is converted to a **double** without changing its value. However, converting a **double** to an **int** truncates the fractional part of the **double** value. Converting large integer types to small integer types (e.g., **long** to **int**) may also result in changed values.

The promotion rules apply to expressions containing values of two or more data types (also referred to as *mixed-type expressions*) and to primitive-data-type values passed as arguments to methods. The type of each value in a mixed-type expression is promoted to the "highest" type in the expression (actually, the expression uses a temporary copy of each value; the original values remain unchanged). The type of a method argument can be promoted to any "higher" type. Figure 6.5 lists the primitive data types and the types to which each is allowed to be promoted automatically.

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

Fig. 6.5 Allowed promotions for primitive data types.

Converting values to lower types can result in different values. Therefore, in cases where information may be lost due to conversion, the Java compiler requires the programmer to use a cast operator to force the conversion to occur. To invoke our **square** method, which uses an integer parameter with the **double** variable **y** (Fig. 6.3), we write the method call as **square((int) y)**. This method call explicitly casts (converts) the value of **y** to an integer for use in method **square**. Thus, if **y**'s value is **4.5**, method **square** returns **16**, not **20.25**.



Common Programming Error 6.11

Converting a primitive-data-type value to another primitive data type may change the value if the new data type is not an allowed promotion (e.g., **double** to **int**). Also, converting any integral value to a floating-point value and back to an integral value may introduce rounding errors into the result.

6.7 Java API Packages

As we have seen, Java contains many predefined classes that are grouped into categories of related classes, called *packages*. Together, we refer to these packages as the *Java applications programming interface (Java API)*, or the *Java class library*.

Throughout the text, **import** statements specify the classes required to compile a Java program. For example, a program uses the statement

```
import javax.swing.JApplet;
```

to tell the compiler to load the **JApplet** class from the **javax.swing** package. One of the great strengths of Java is the large number of classes in the packages of the Java API that programmers can reuse rather than “reinventing the wheel.” We exercise a large number of these classes in this book. Figure 6.6 lists a subset of the many packages in the Java API and provides a brief description of each package. We use classes from these packages and others throughout this book. We provide this table to begin introducing you the variety of reusable components available in the Java API. When learning Java, you should spend time reading the descriptions of the packages and classes in the Java API documentation (java.sun.com/j2se/1.3/docs/api).

Package	Description
java.applet	<i>The Java Applet Package.</i> This package contains the Applet class and several interfaces that enable the creation of applets, interaction of applets with the browser and playing audio clips. In Java 2, class javax.swing.JApplet is used to define an applet that uses the <i>Swing GUI components</i> .
java.awt	<i>The Java Abstract Windowing Toolkit Package.</i> This package contains the classes and interfaces required to create and manipulate graphical user interfaces in Java 1.0 and 1.1. In Java 2, these classes can still be used, but the <i>Swing GUI components</i> of the javax.swing packages are often used instead.

Fig. 6.6 Packages of the Java API (part 1 of 2).

Package	Description
java.awt.event	<i>The Java Abstract Windowing Toolkit Event Package.</i> This package contains classes and interfaces that enable event handling for GUI components in both the java.awt and javax.swing packages.
java.io	<i>The Java Input/Output Package.</i> This package contains classes that enable programs to input and output data (see Chapter 16, Files and Streams).
java.lang	<i>The Java Language Package.</i> This package contains classes and interfaces required by many Java programs (many are discussed throughout this text) and is automatically imported by the compiler into all programs.
java.net	<i>The Java Networking Package.</i> This package contains classes that enable programs to communicate via networks (see Chapter 17, Networking).
java.text	<i>The Java Text Package.</i> This package contains classes and interfaces that enable a Java program to manipulate numbers, dates, characters and strings. It provides many of Java's internationalization capabilities i.e., features that enable a program to be customized to a specific locale (e.g., an applet may display strings in different languages, based on the user's country).
java.util	<i>The Java Utilities Package.</i> This package contains utility classes and interfaces, such as: date and time manipulations, random-number processing capabilities (Random), storing and processing large amounts of data, breaking strings into smaller pieces called <i>tokens</i> (StringTokenizer) and other capabilities (see Chapter 19, Data Structures, Chapter 20, Java Utilities Package and Bit Manipulation, and Chapter 21, The Collections API).
javax.swing	<i>The Java Swing GUI Components Package.</i> This package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.
javax.swing.event	<i>The Java Swing Event Package.</i> This package contains classes and interfaces that enable event handling for GUI components in the javax.swing package.

Fig. 6.6 Packages of the Java API (part 2 of 2).

The set of packages available in the Java 2 Software Development Kit (J2SDK) is quite large. In addition to the packages summarized in Fig. 6.6, the J2SDK includes packages for complex graphics, advanced graphical user interfaces, printing, advanced networking, security, database processing, multimedia, accessibility (for people with disabilities) and many other functions. For an overview of the packages in the J2SDK version 1.3, visit

java.sun.com/j2se/1.3/docs/api/overview-summary.html

Also, many other packages are available for download at **java.sun.com**.

6.8 Random-Number Generation

We now take a brief and, hopefully, entertaining diversion into a popular programming application, namely, simulation and game playing. In this section and the next section, we will develop a nicely structured game-playing program that includes multiple methods. The program uses most of the control structures we have studied to this point in the book and introduces several new concepts.

There is something in the air of a gambling casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the *element of chance*, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced through the **random** method from the **Math** class. [Note: Java also provides a **Random** class in package `java.util`. Class **Random** is covered in Chapter 20.]

Consider the following statement:

```
double randomValue = Math.random();
```

The **random** method of class **Math** generates a random **double** value from 0.0 up to, but not including, 1.0. If method **random** truly produces values at random, then every value from 0.0 up to, but not including, 1.0 should have an equal *chance* (or *probability*) of being chosen each time method **random** is called. Note that the values returned by **random** are actually *pseudo-random numbers*—a sequence of values produced by a complex mathematical calculation. That calculation uses the current time of day to *seed* the random number generator, such that each execution of a program yields a different sequence of random values.

The range of values produced directly by method **random** often is different from the range of values required in a particular Java application. For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.” A program that simulates rolling a six-sided die would require random integers in the range from 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game would require random integers in the range from 1 to 4.

To demonstrate method **random**, let us develop a program that simulates 20 rolls of a six-sided die and displays the value of each roll. We use the multiplication operator (*****) in conjunction with method **random** as follows to produce integers in the range from 0 to 5:

```
(int) ( Math.random() * 6 )
```

This manipulation is called *scaling* the range of values produced by **Math** method **random**. The number **6** in the preceding expression is called the *scaling factor*. The integer cast operator truncates the floating-point part (the part after the decimal point) of each value produced by the expression. Next, we *shift* the range of numbers produced by adding 1 to our previous result, as in

```
1 + (int) ( Math.random() * 6 )
```

Figure 6.7 confirms that the results of the preceding calculation are integers in the range from 1 to 6.

```

1  // Fig. 6.7: RandomIntegers.java
2  // Shifted, scaled random integers.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class RandomIntegers {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         int value;
13         String output = "";
14
15         // loop 20 times
16         for ( int counter = 1; counter <= 20; counter++ ) {
17
18             // pick random integer between 1 and 6
19             value = 1 + ( int ) ( Math.random() * 6 );
20
21             output += value + " "; // append value to output
22
23             // if counter divisible by 5,
24             // append newline to String output
25             if ( counter % 5 == 0 )
26                 output += "\n";
27
28         } // end for structure
29
30         JOptionPane.showMessageDialog( null, output,
31             "20 Random Numbers from 1 to 6",
32             JOptionPane.INFORMATION_MESSAGE );
33
34         System.exit( 0 ); // terminate application
35
36     } // end method main
37
38 } // end class RandomIntegers

```

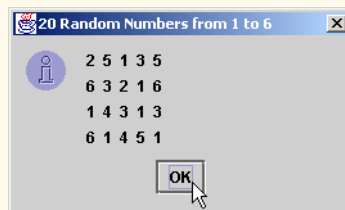


Fig. 6.7 Shifted and scaled random integers .

To show that these numbers occur with approximately equal likelihood, let us simulate 6000 rolls of a die with the program in Fig. 6.8. Each integer from 1 to 6 should appear approximately 1000 times.

```
1 // Fig. 6.8: RollDie.java
2 // Roll a six-sided die 6000 times.
3
4 // Java extension packages
5 import javax.swing.*;
6
7 public class RollDie {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12        int frequency1 = 0, frequency2 = 0, frequency3 = 0,
13            frequency4 = 0, frequency5 = 0, frequency6 = 0, face;
14
15        // summarize results
16        for ( int roll = 1; roll <= 6000; roll++ ) {
17            face = 1 + ( int ) ( Math.random() * 6 );
18
19            // determine roll value and increment appropriate counter
20            switch ( face ) {
21
22                case 1:
23                    ++frequency1;
24                    break;
25
26                case 2:
27                    ++frequency2;
28                    break;
29
30                case 3:
31                    ++frequency3;
32                    break;
33
34                case 4:
35                    ++frequency4;
36                    break;
37
38                case 5:
39                    ++frequency5;
40                    break;
41
42                case 6:
43                    ++frequency6;
44                    break;
45
46            } // end switch structure
47
48        } // end for structure
```

Fig. 6.8 Rolling a six-sided die 6000 times (part 1 of 2).

```

49
50     JTextArea outputArea = new JTextArea();
51
52     outputArea.setText( "Face\tFrequency" +
53         "\n1\t" + frequency1 + "\n2\t" + frequency2 +
54         "\n3\t" + frequency3 + "\n4\t" + frequency4 +
55         "\n5\t" + frequency5 + "\n6\t" + frequency6 );
56
57     JOptionPane.showMessageDialog( null, outputArea,
58         "Rolling a Die 6000 Times",
59         JOptionPane.INFORMATION_MESSAGE );
60
61     System.exit( 0 ); // terminate application
62
63 } // end method main
64
65 } // end class RollDie

```

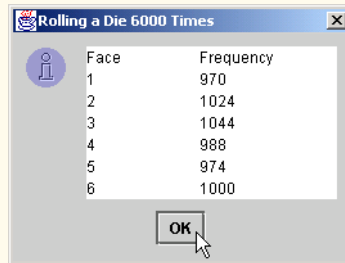


Fig. 6.8 Rolling a six-sided die 6000 times (part 2 of 2).

As the program output shows, scaling and shifting the values produced by method **random** enables the program to simulate realistically the rolling of a six-sided die. Note that the use of nested control structures in the program to determine the number of times each side of the six-sided die occurred. The **for** structure at lines 16–48 iterates 6000 times. During each iteration of the loop, line 17 produces a value from 1 to 6. The nested **switch** structure at lines 20–46 uses as its controlling expression the **face** value that was randomly chosen. Based on the value of **face**, the **switch** structure increments one of the six counter variables during each iteration of the loop. Note that the **switch** structure has no **default** case. When we study arrays in Chapter 7, we show how to replace the entire **switch** structure in this program with a single statement. Run the program several times, and observe the results. Notice that the program produces different results each time the program executes.

The values produced directly by **random** are always in the range

$$0.0 \leq \text{Math.random()} < 1.0$$

Previously, we demonstrated how to write a single statement to simulate the rolling of a six-sided die with the statement

```
face = 1 + (int) ( Math.random() * 6 );
```

which always assigns an integer (at random) to variable **face** in the range $1 \leq \text{face} \leq 6$. Note that the width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale **random** with the multiplication operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) added to **(int) (Math.random() * 6)**. We can generalize this result as

$$n = a + (\text{int}) (\text{Math.random}() * b);$$

where **a** is the *shifting value* (which is equal to the first number in the desired range of consecutive integers) and **b** is the *scaling factor* (which is equal to the width of the desired range of consecutive integers). In the exercises, we will see that it is possible to choose integers at random from sets of values other than ranges of consecutive integers.

6.9 Example: A Game of Chance

One of the most popular games of chance is a dice game known as “craps,” which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

A player rolls two dice. Each die has six faces. These faces contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll your point value). The player loses by rolling a 7 before making the point.

The applet in Fig. 6.9 simulates the game of craps.

```

1  // Fig. 6.9: Craps.java
2  // Craps
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class Craps extends JApplet implements ActionListener {
12
13     // constant variables for game status
14     final int WON = 0, LOST = 1, CONTINUE = 2;
15
16     // other variables used
17     boolean firstRoll = true;    // true if first roll of dice
18     int sumOfDice = 0;           // sum of the dice
19     int myPoint = 0;             // point if no win/loss on first roll
20     int gameStatus = CONTINUE;  // game not over yet
21

```

Fig. 6.9 Program to simulate the game of craps (part 1 of 5).

```

22 // graphical user interface components
23 JLabel die1Label, die2Label, sumLabel, pointLabel;
24 JTextField die1Field, die2Field, sumField, pointField;
25 JButton rollButton;
26
27 // set up GUI components
28 public void init()
29 {
30     // obtain content pane and change its layout to
31     // a FlowLayout
32     Container container = getContentPane();
33     container.setLayout( new FlowLayout() );
34
35     // create label and text field for die 1
36     die1Label = new JLabel( "Die 1" );
37     container.add( die1Label );
38     die1Field = new JTextField( 10 );
39     die1Field.setEditable( false );
40     container.add( die1Field );
41
42     // create label and text field for die 2
43     die2Label = new JLabel( "Die 2" );
44     container.add( die2Label );
45     die2Field = new JTextField( 10 );
46     die2Field.setEditable( false );
47     container.add( die2Field );
48
49     // create label and text field for sum
50     sumLabel = new JLabel( "Sum is" );
51     container.add( sumLabel );
52     sumField = new JTextField( 10 );
53     sumField.setEditable( false );
54     container.add( sumField );
55
56     // create label and text field for point
57     pointLabel = new JLabel( "Point is" );
58     container.add( pointLabel );
59     pointField = new JTextField( 10 );
60     pointField.setEditable( false );
61     container.add( pointField );
62
63     // create button user clicks to roll dice
64     rollButton = new JButton( "Roll Dice" );
65     rollButton.addActionListener( this );
66     container.add( rollButton );
67 }
68
69 // process one roll of dice
70 public void actionPerformed((ActionEvent actionEvent) )
71 {
72     // first roll of dice
73     if ( firstRoll ) {
74         sumOfDice = rollDice(); // roll dice

```

Fig. 6.9 Program to simulate the game of craps (part 2 of 5).

```

75
76         switch ( sumOfDice ) {
77
78             // win on first roll
79             case 7: case 11:
80                 gameStatus = WON;
81                 pointField.setText( "" );    // clear point field
82                 break;
83
84             // lose on first roll
85             case 2: case 3: case 12:
86                 gameStatus = LOST;
87                 pointField.setText( "" );    // clear point field
88                 break;
89
90             // remember point
91             default:
92                 gameStatus = CONTINUE;
93                 myPoint = sumOfDice;
94                 pointField.setText( Integer.toString( myPoint ) );
95                 firstRoll = false;
96                 break;
97
98         } // end switch structure
99
100     } // end if structure body
101
102     // subsequent roll of dice
103     else {
104         sumOfDice = rollDice();    // roll dice
105
106         // determine game status
107         if ( sumOfDice == myPoint ) // win by making point
108             gameStatus = WON;
109         else
110             if ( sumOfDice == 7 )    // lose by rolling 7
111                 gameStatus = LOST;
112     }
113
114     // display message indicating game status
115     displayMessage();
116
117 } // end method actionPerformed
118
119 // roll dice, calculate sum and display results
120 public int rollDice()
121 {
122     int die1, die2, sum;
123
124     // pick random die values
125     die1 = 1 + ( int ) ( Math.random() * 6 );
126     die2 = 1 + ( int ) ( Math.random() * 6 );
127

```

Fig. 6.9 Program to simulate the game of craps (part 3 of 5).

```

128     sum = die1 + die2;    // sum die values
129
130     // display results
131     die1Field.setText( Integer.toString( die1 ) );
132     die2Field.setText( Integer.toString( die2 ) );
133     sumField.setText( Integer.toString( sum ) );
134
135     return sum;    // return sum of dice
136
137 }    // end method rollDice
138
139 // determine game status and display appropriate message
140 // in status bar
141 public void displayMessage()
142 {
143     // game should continue
144     if ( gameStatus == CONTINUE )
145         showStatus( "Roll again." );
146
147     // game won or lost
148     else {
149
150         if ( gameStatus == WON )
151             showStatus( "Player wins. " +
152                         "Click Roll Dice to play again." );
153         else
154             showStatus( "Player loses. " +
155                         "Click Roll Dice to play again." );
156
157         // next roll is first roll of new game
158         firstRoll = true;
159     }
160
161 }    // end method displayMessage
162
163 }    // end class Craps

```

A JLabel object A JTextField object A JButton object

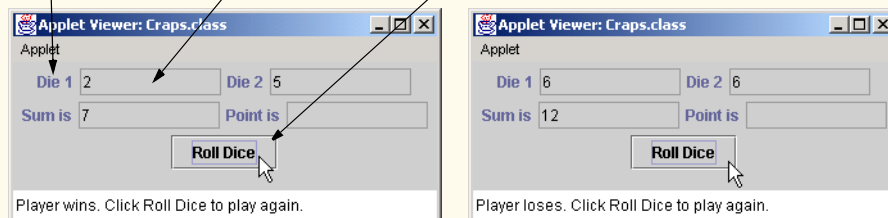


Fig. 6.9 Program to simulate the game of craps (part 4 of 5).

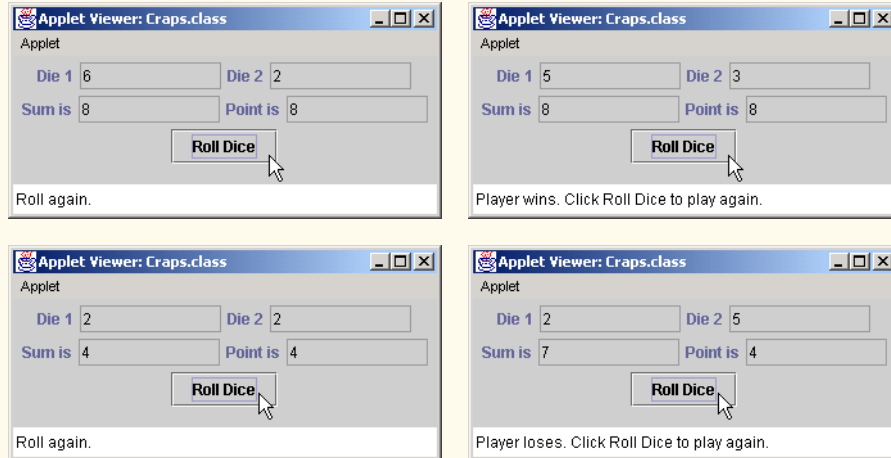


Fig. 6.9 Program to simulate the game of craps (part 5 of 5).

Notice that the player must roll two dice on the first and all subsequent rolls. When you execute the applet, click the **Roll Dice** button to play the game. The lower left corner of the **appletviewer** window displays the result of each roll. The screen captures show four separate executions of the applet (a win and a loss on the first roll, and a win and a loss after the first roll).

Until now, most user interactions in our programs have been through either an input dialog (in which the user could type an input value for the program) or a message dialog (in which a message was displayed to the user, and the user could click **OK** to dismiss the dialog). Although these dialogs are valid ways to receive input from a user and display output in a Java program, their capabilities are fairly limited—an input dialog can obtain only one value at a time from the user and a message dialog can display only one message. It is much more common to receive multiple inputs from the user at once (such as the user entering name and address information) or display many pieces of data at once (such as the values of the dice, the sum of the dice and the point, in this example). To begin our introduction to more elaborate user interfaces, this program illustrates two new graphical user interface concepts: Attaching several GUI components to an applet and GUI *event handling*. We discuss each of the new issues as they are encountered in the program.

The **import** statements in lines 5–9 enable the compiler to load the classes used in this applet. Line 5 specifies that the program uses classes from package **java.awt** (specifically, classes **Container** and **FlowLayout**). Line 6 specifies that the program uses classes from package **java.awt.event**. This package contains many data types that enable a program to process a user's interactions with a program's GUI. In this program, we use the **ActionListener** and **ActionEvent** data types from package **java.awt.event**. Line 9 specifies that the program uses classes from package **javax.swing** (specifically, **JApplet**, **JLabel**, **TextField** and **Button**).

As stated earlier, every Java program is based on at least one class definition that extends and enhances an existing class definition via inheritance. Remember that applets inherit from class **JApplet**. Line 11 indicates that class **Craps** inherits from **JApplet**

and *implements* **ActionListener**. A class can inherit existing attributes and behaviors (data and methods) from another class specified to the right of keyword **extends** in the class definition. In addition, a class can implement one or more *interfaces*. An interface specifies one or more behaviors (i.e., methods), which you must define in your class definition. Implementing interface **ActionListener** forces us to *define a method* with the first line

```
public void actionPerformed((ActionEvent actionEvent )
```

in our **Craps** class. This method's task is to process a user's interaction with the **JButton** (called **Roll Dice** on the user interface). When the user presses the button, this method will be called automatically in response to the user interaction. This process is called *event handling*. The *event* is the user interaction (i.e., pressing the button). The *event handler* is method **actionPerformed**. We discuss the details of this interaction and method **actionPerformed** shortly. Chapter 9, Object-Oriented Programming, discusses interfaces in detail. For now, as you develop your own applets that have graphical user interfaces, mimic the features that support event handling of the GUI components we present.

The game of craps is reasonably involved. The player may win or lose on the first roll, or may win or lose on any roll. Line 14 creates variables that define the three states of a game of craps: Game won (**WON**), game lost (**LOST**) or continue rolling the dice (**CONTINUE**). Keyword **final** at the beginning of the declaration indicates that these are *constant variables*. When a program declares a **final** variable, the program must initialize the variable before using the variable and cannot modify the variable thereafter. If the variable is an instance variable, this initialization normally occurs in the variable's declaration. The initialization also can occur in a special method of a class called a *constructor* (discussed in Chapter 8). Constant variables are often called *named constants* or *read-only variables*. We provide more details on keyword **final** in Chapter 7 and Chapter 8.



Common Programming Error 6.12

After declaring and initializing a **final** variable, attempting to assign another value to that variable is a syntax error.



Good Programming Practice 6.5

Use only uppercase letters (with underscores between words) in the names of **final** variables. This format makes these constants stand out in a program.



Good Programming Practice 6.6

Using meaningfully named **final** variables rather than integer constants (such as 2) makes programs more readable.

Lines 17–20 declare several instance variables that are used throughout the **Craps** applet. Variable **firstRoll** is a **boolean** variable that indicates whether the next roll of the dice is the first roll in the current game. Variable **sumOfDice** maintains the sum of the dice for the last roll. Variable **myPoint** stores the “point” if the player does not win or lose on the first roll. Variable **gameStatus** keeps track of the current state of the game (**WON**, **LOST** or **CONTINUE**).

Lines 23–25 declare references to the GUI components used in this applet's graphical user interface. References **die1Label**, **die2Label**, **sumLabel** and **pointLabel** all refer to **JLabel** objects. A **JLabel** contains a string of characters to be displayed on the

screen. Normally, a **JLabel** indicates the purpose of another GUI component on the screen. In the screen captures of Fig. 6.9, the **JLabel** objects are the text to the left of each rectangle in the first two rows of the user interface. References **die1Field**, **die2Field**, **sumField** and **pointField** all refer to **JTextField** objects. **JTextFields** are used to get a single line of information from the user at the keyboard or to display information on the screen. The **JTextField** objects are the rectangles to the right of each **JLabel** in the first two rows of the user interface. Reference **rollButton** refers to a **JButton** object. When the user presses a **JButton**, the program normally responds by performing a task (rolling the dice, in this example). The **JButton** object is the rectangle containing the words **Roll Dice** at the bottom of the user interface shown in Fig. 6.9. We have seen **JButtons** in prior programs—every message dialog and every input dialog contained an **OK** button to dismiss the message dialog or send the user's input to the program. We also have seen **JTextFields** in prior programs—every input dialog contains a **JTextField** in which the user types an input value.

Method **init** (lines 28–67) creates the GUI component objects and attaches them to the user interface. Line 32 declares **Container** reference **container** and assigns to it the result method **getContentPane**. Remember, method **getContentPane** returns a reference to the applet's content pane that can be used to attach GUI components to the applet's user interface.

Line 33 uses **Container** method **setLayout** to specify the *layout manager* for the applet's user interface. Layout managers arrange GUI components on a **Container** for presentation purposes. The layout managers determine the position and size of every GUI component attached to the container, thereby processing most of the layout details and enabling the programmer to concentrate on the basic look and feel of the programs.

FlowLayout is the simplest layout manager. GUI components are placed from left to right in the order in which they are attached to the **Container** (the applet's content pane in this example) with method **add**. When the layout manager reaches the edge of the container, it begins a new row of components and continues laying out the components on that row. Line 33 creates a new object of class **FlowLayout** and passes it as the argument to method **setLayout**. Normally, the layout is set before any GUI components are added to a **Container**.



Common Programming Error 6.13

*If a **Container** is not large enough to display the GUI component attached to it, some or all of the GUI components simply will not display.*

[*Note: Each **Container** can have only one layout manager at a time. Separate **Containers** in the same program can have different layout managers. Most Java programming environments provide GUI design tools that help a programmer graphically design a GUI; then the tools write Java code to create the GUI. Some of these GUI design tools also allow the programmer to use layout managers. Chapter 12 and Chapter 13 discuss several layout managers that allow more precise control over the layout of the GUI components.*]

Lines 36–40, 43–47, 50–54 and 57–61 each create a **JLabel** and **JTextField** pair and attach them to the user interface. Since these sets of lines are all quite similar, we concentrate on lines 36–40. Line 36 creates a new **JLabel** object, initializes it with the string **"Die 1"** and assigns the object to reference **die1Label**. This procedure labels the corresponding **JTextField** (named **die1Field**) in the user interface, so the user can deter-

mine the purpose of the value displayed in **die1Field**. Line 37 attaches the **JLabel** to which **die1Label** refers to the applet's content pane. Line 38 creates a new **JTextField** object, initializes it to be 10 characters wide and assigns the object to reference **die1Field**. This **JTextField** displays the value of the first die after each roll of the dice. Line 39 uses **JTextField** method **setEditable** with the argument **false** to indicate that the user should not be able to type in the **JTextField**. This setting makes the **JTextField** *uneditable* and causes it to be displayed with a gray background by default. An editable **JTextField** has a white background (as seen in input dialogs). Line 32 attaches the **JTextField** to which **die1Field** refers to the applet's content pane.

Line 64 creates a new **JButton** object, initializes it with the string "Roll Dice" (this string will appear on the button) and assigns the object to reference **rollButton**.

Line 65 specifies that **this** applet should *listen* for events from the **rollButton**. The **this** keyword enables the applet to refer to itself. (We discuss **this** in detail in Chapter 8.) When the user interacts with a GUI component an *event* is sent to the applet. GUI events are messages (method calls) indicating that the user of the program has interacted with one of the program's GUI components. For example, when you press **rollButton** in this program, a message indicating the event that occurred is sent to the applet to notify the applet that you pressed the button. For a **JButton**, the message indicates to the applet that *an action was performed* by the user on the **JButton** and automatically calls method **actionPerformed** to process the user's interaction.

This style of programming is known as *event-driven programming*—the user interacts with a GUI component, the program is notified of the event and the program processes the event. The user's interaction with the GUI "drives" the program. The methods that are called when an event occurs are also known as *event-handling methods*. When a GUI event occurs in a program, Java creates an object containing information about the event that occurred and *calls* an appropriate event-handling method. Before any event can be processed, each GUI component must know which object in the program defines the event-handling method that will be called when an event occurs. In line 65, **JButton** method **addActionListener** is used to tell **rollButton** that the applet (**this**) can *listen* for *action events* and defines method **actionPerformed**. This procedure is called *registering the event handler* with the GUI component. (We also like to call it the *start-listening line*, because the applet is now listening for events from the button.) To respond to an action event, we must define a class that implements **ActionListener** (this requires that the class also define method **actionPerformed**), and we must register the event handler with the GUI component. Finally, the last line in **init** attaches the **JButton** to which **roll** refers to the applet's content pane, thus completing the user interface.

Method **actionPerformed** (lines 70–117) is one of several methods that process interactions between the user and GUI components. The first line of the method indicates that **actionPerformed** is a **public** method that returns nothing (**void**) when it completes its task. Method **actionPerformed** receives one argument—an **ActionEvent**—when it is called in response to an action performed on a GUI component by the user (in this case, pressing the **JButton**). The **ActionEvent** argument contains information about the action that occurred.

We define method **rollDice** (lines 120–137) to roll the dice and compute and display their sum. Method **rollDice** is defined once, but it is called from two places in the program (lines 74 and 104). Method **rollDice** takes no arguments, so it has an empty

parameter list. Method **rollDice** returns the sum of the two dice, so a return type of **int** is indicated in the method's header.

The user clicks **Roll Dice** to roll the dice. This action invokes method **actionPerformed** (line 70) of the applet. Method **actionPerformed** checks the **boolean** variable **firstRoll** (line 73) to determine if it is **true** or **false**. If it is **true**, this roll is the first roll of the game. Line 74 calls **rollDice**, which picks two random values from 1 to 6, displays the value of the first die, second die and the sum of the dice in the first three **JTextFields**, respectively, and returns the sum of the dice. Note that the integer values are converted to **Strings** (lines 131–133) with **static** method **Integer.toString**, because **JTextFields** can display only **Strings**. After the first roll, the nested **switch** structure at line 76 in **actionPerformed** determines if the game has been won or lost, or if the game should continue with another roll. After the first roll, if the game is not over, **sumOfDice** is saved in **myPoint** and displayed in **pointField**.

Line 115 calls method **displayMessage** (defined at lines 141–161) to display the current status of the game. The **if/else** structure at line 144 uses applet method **showStatus** to display a **String** in the applet container's status bar. Line 145 displays

Roll again.

if **gameStatus** is equal to **CONTINUE**. Lines 150–151 display

Player wins. Click Roll Dice to play again.

if **gameStatus** is equal to **WON**. Lines 153–154 display

Player loses. Click Roll Dice to play again.

if **gameStatus** is equal to **LOST**. Method **showStatus** receives a **String** argument and displays it in the status bar of the applet container. If the game is over (i.e., it has been won or lost) line 158 sets **firstRoll** to **true** to indicate that the next roll of the dice is the first roll of the next game.

The program then waits for the user to click the **Roll Dice** button again. Each time the user presses **Roll Dice** button, method **actionPerformed** invokes method **rollDice** to produce a new **sumOfDice**. If the current roll is a continuation of an incomplete game, the code in lines 103–112 executes. In line 107, if **sumOfDice** matches **myPoint**, line 108 sets **gameStatus** to **WON**, and the game is complete. In line 110, if **sumOfDice** is equal to 7, line 111 sets **gameStatus** to **LOST**, and the game is complete. When the game completes, **displayMessage** displays an appropriate message, and the user can click the **Roll Dice** button to begin a new game. Throughout the program, the four **JTextFields** are updated with the new values of the dice and the sum on each roll, and the **pointField** is updated each time a new game begins.

Note the interesting use of the various program control mechanisms we have discussed. The craps program uses four methods—**init**, **actionPerformed**, **rollDice** and **displayMessage**—and the **switch**, **if/else** and nested **if** structures. Note also the use of multiple **case** labels in the **switch** structure to execute the same statements (lines 79 and 85). Also, note that the event-handling mechanism acts as a form of program control. In this program, event-handling enables user-controlled repetition—each time the user clicks **Roll Dice**, the program rolls the dice again. In the exercises, we investigate various interesting characteristics of the game of craps.

6.10 Duration of Identifiers

Chapter 2 through Chapter 5 used identifiers for variable names and reference names. The attributes of variables and references include name, type, size and value. We also use identifiers as names for user-defined methods and classes. Actually, each identifier in a program has other attributes, including *duration* and *scope*.

An identifier's *duration* (also called its *lifetime*) is the period during which the identifier exists in memory. Some identifiers exist for brief periods of time and others exist for the entire execution of a program.

An identifier's *scope* defines where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program, while others can be referenced only from limited portions of a program. This section discusses duration of identifiers. Section 6.11 discusses the scope of identifiers.

Identifiers that represent local variables in a method (i.e., parameters and variables declared in the body of the method) have *automatic duration*. Automatic-duration variables are created when program control reaches their declaration; they exist while the block in which they are declared is active; and they are destroyed when the block in which they are declared is exited. We will continue to refer to variables of automatic duration as *local variables*.



Performance Tip 6.2

Automatic duration is a means of conserving memory, because automatic-duration variables are created when program control reaches their declaration and are destroyed when the block in which they are declared is exited.

The instance variables of a class are initialized automatically by the compiler if the programmer does not provide explicit initial values. Variables of the primitive data types are initialized to zero, except **boolean** variables, which are initialized to **false**. References are initialized to **null**. Unlike instance variables of a class, automatic variables must be initialized by the programmer before they can be used.



Testing and Debugging Tip 6.2

If an automatic variable is not initialized before it is used in a method, the compiler issues an error message.

Java also has identifiers of *static duration*. Variables and references of static duration exist from the point at which the class that defines them is loaded into memory for execution until the program terminates. Their storage is allocated and initialized when their classes are loaded into memory. Even though static-duration variables and reference names exist when their classes are loaded into memory, these identifiers cannot necessarily be used throughout a program. Duration (an identifier's lifetime) and scope (where an identifier can be used) are separate issues, as shown in Section 6.11.



Software Engineering Observation 6.9

Automatic duration is an example of the principle of least privilege. This principle states that each component of a system should have sufficient rights and privileges to accomplish its designated task, but no additional rights or privileges. This constraint helps prevent accidental and/or malicious errors from occurring in systems. Why have variables stored in memory and accessible when they are not needed?

6.11 Scope Rules

The *scope* of an identifier for a variable, reference or method is the portion of the program that can reference the identifier. A local variable or reference declared in a block can be used only in that block or in blocks nested within that block. The scopes for an identifier are *class scope* and *block scope*. There is also a special scope for labels used with the **break** and **continue** statements (introduced in Chapter 5, “Control Structures: Part 2”). A label is visible only in the body of the repetition structure that immediately follows the label.

Methods and instance variables of a class have *class scope*. Class scope begins at the opening left brace, **{**, of the class definition and terminates at the closing right brace, **}**, of the class definition. Class scope enables methods of a class to invoke directly all methods defined in that same class or inherited into that class (such as the methods inherited into our applets from class **JApplet**) and to access directly all instance variables defined in the class. In Chapter 8, we will see that **static** methods are an exception to this rule. In a sense, all instance variables and methods of a class are *global* to the methods of the class in which they are defined (i.e., the methods can modify the instance variables directly and invoke other methods of the class). [Note: One of the reasons we use mainly applets in this chapter is to simplify our discussions. We have not as yet introduced a true windowed application in which the methods of our application class will have access to all the other methods of the class and the instance variables of the class.]

Identifiers declared inside a block have *block scope*. Block scope begins at the identifier’s declaration and ends at the terminating right brace (**}**) of the block. Local variables of a method have block scope, as do method parameters, which are also local variables of the method. Any block may contain variable or reference declarations. When blocks are nested in a method’s body and an identifier declared in an outer block has the same name as an identifier declared in an inner block, the compiler generates a syntax error stating that the variable is already defined. If a local variable in a method has the same name as an instance variable, the instance variable is “hidden” until the block terminates execution. In Chapter 8, we discuss how to access such “hidden” instance variables.



Common Programming Error 6.14

Accidentally using the same name for an identifier in an inner block of a method as is used for an identifier in an outer block of the same method results in a syntax error from the compiler.



Good Programming Practice 6.7

Avoid local-variable names that hide instance-variable names. This can be accomplished by avoiding the use of duplicate identifiers in a class.

The applet of Fig. 6.10 demonstrates scoping issues with instance variables and local variables.

This example uses the applet’s **start** method (lines 27–40) for the first time. Remember, when an applet container loads an applet, the container first creates an instance of the applet class. It then calls the applet’s **init**, **start** and **paint** methods. Method **start** always is defined with the line shown on line 27 as its first line.

```

1  // Fig. 6.10: Scoping.java
2  // A scoping example.
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class Scoping extends JApplet {
11     JTextArea outputArea;
12
13     // instance variable accessible to all methods of this class
14     int x = 1;
15
16     // set up applet's GUI
17     public void init()
18     {
19         outputArea = new JTextArea();
20         Container container = getContentPane();
21         container.add( outputArea );
22     } // end method init
23
24     // method start called after init completes; start calls
25     // methods useLocal and useInstance
26     public void start()
27     {
28         int x = 5;    // variable local to method start
29
30         outputArea.append( "local x in start is " + x );
31
32         useLocal();    // useLocal has local x
33         useInstance(); // useInstance uses instance variable x
34         useLocal();    // useLocal reinitializes local x
35         useInstance(); // instance variable x retains its value
36
37         outputArea.append( "\n\nlocal x in start is " + x );
38     } // end method start
39
40     // useLocal reinitializes local variable x during each call
41     public void useLocal()
42     {
43         int x = 25; // initialized each time useLocal is called
44
45         outputArea.append( "\n\nlocal x in useLocal is " + x +
46             " after entering useLocal" );
47         ++x;
48         outputArea.append( "\n\nlocal x in useLocal is " + x +
49             " before exiting useLocal" );
50     } // end method useLocal
51
52
53

```

Fig. 6.10 A scoping example (part 1 of 2).

```

54
55 // useInstance modifies instance variable x during each call
56 public void useInstance()
57 {
58     outputArea.append( "\n\ninstance variable x is " + x +
59                       " on entering useInstance" );
60     x *= 10;
61     outputArea.append( "\n\ninstance variable x is " + x +
62                       " on exiting useInstance" );
63
64 } // end method useInstance
65
66 } // end class Scoping

```

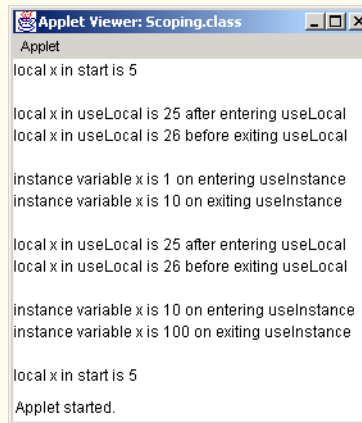


Fig. 6.10 A scoping example (part 2 of 2).

Line 14 declares and initializes instance variable **x** to **1**. This instance variable is hidden in any block (or method) that declares a variable named **x**. Method **start** declares a local variable **x** (line 29) and initializes it to **5**. This variable's value is displayed in the **JTextArea** **outputArea** to show that the instance variable **x** is hidden in **start**. The program defines two other methods—**useLocal** (lines 43–53) and **useInstance** (lines 56–64)—that each take no arguments and do not return results. Method **start** calls each of these methods twice. Method **useLocal** defines local (automatic) variable **x** (line 45). When **useLocal** is called (line 33), it creates local variable **x** and initializes **x** to **25**, displays the value of **x** in **outputArea**, increments **x** and displays the value of **x** again. When **useLocal** is called again (line 35), it recreates local variable **x** and initializes **x** to **25**. Method **useInstance** does not declare any variables. Therefore, when it refers to variable **x**, the instance variable **x** is used. When **useInstance** is called (line 34), it displays the instance variable **x** in **outputArea**, multiplies the instance variable **x** by **10** and displays instance variable **x** again before returning. The next time method **useInstance** is called (line 36), the instance variable has its modified value, **10**. Finally, the program displays the local variable **x** in **start** again to show that none of the method calls modified the value of **x**, because the methods all referred to variables in other scopes.

6.12 Recursion

The programs we have discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it is useful to have methods call themselves. A *recursive method* is a method that calls itself either directly or indirectly through another method. Recursion is an important topic discussed at length in upper-level computer science courses. In this and the next section, simple examples of recursion are presented. This book contains an extensive treatment of recursion. Figure 6.15 (at the end of Section 6.14) summarizes the recursion examples and exercises in this book.

We consider recursion conceptually first. Then we examine several programs containing recursive methods. Recursive problem-solving approaches have a number of elements in common. A recursive method is called to solve a problem. The method actually knows how to solve only the simplest case(s) or so-called *base case(s)*. If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two conceptual pieces: a piece that the method knows how to do (base case) and a piece that the method does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of the original problem. Because this new problem looks like the original problem, the method invokes (calls) a fresh copy of itself to go to work on the smaller problem; this procedure is referred to as a *recursive call* and is also called the *recursion step*. The recursion step also normally includes the keyword **return**, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still open (i.e., while it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. For the recursion eventually to terminate, each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the method recognizes the base case, and returns a result to the previous copy of the method, and a sequence of returns ensues up the line until the original method call eventually returns the final result to the caller. This process sounds exotic when compared with the conventional problem solving we have performed to this point. As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer n , written $n!$ (and pronounced “ n factorial”), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

where $1!$ is equal to 1 and $0!$ is defined to be 1. For example, $5!$ is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, **number**, greater than or equal to 0, can be calculated *iteratively* (nonrecursively) using the **for** structure as follows:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;
```

A recursive definition of the factorial method is arrived at by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

For example, $5!$ is clearly equal to $5 \cdot 4!$, as is shown by the following equations:

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

The evaluation of $5!$ would proceed as shown in Fig. 6.11. Figure 6.11 (a) shows how the succession of recursive calls proceeds until $1!$ is evaluated to be 1, which terminates the recursion. Figure 6.11 (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 6.12 uses recursion to calculate and print the factorials of the integers from 0 to 10. (The choice of the data type **long** will be explained momentarily.) The recursive method **factorial** (lines 29–39) first tests to determine whether a terminating condition (line 32) is **true**. If **number** is less than or equal to 1 (the base case), **factorial** returns 1, no further recursion is necessary and the method returns. If **number** is greater than 1, line 37 expresses the problem as the product of **number** and a recursive call to **factorial** evaluating the factorial of **number - 1**. Note that **factorial(number - 1)** is a slightly simpler problem than the original calculation **factorial(number)**.

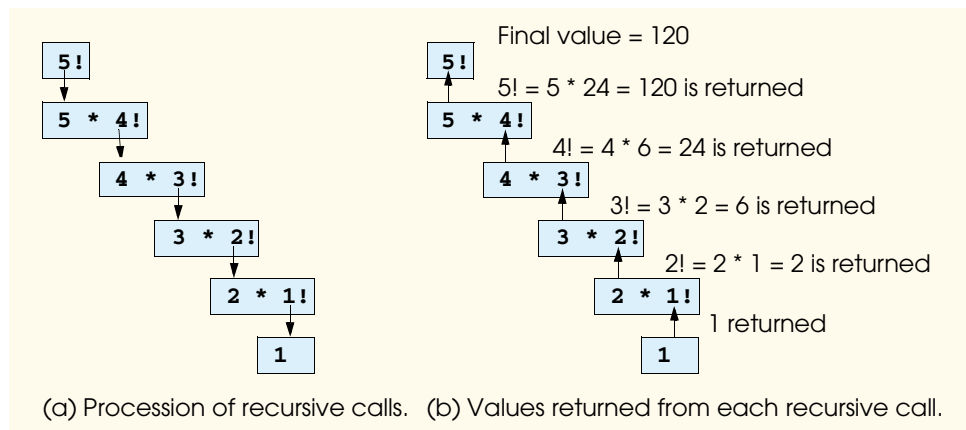


Fig. 6.11 Recursive evaluation of $5!$.

```

1 // Fig. 6.12: FactorialTest.java
2 // Recursive factorial method
3
4 // Java core packages
5 import java.awt.*;
6

```

Fig. 6.12 Calculating factorials with a recursive method (part 1 of 2).

```

7  // Java extension packages
8  import javax.swing.*;
9
10 public class FactorialTest extends JApplet {
11     JTextArea outputArea;
12
13     // initialize applet by creating GUI and calculating factorials
14     public void init()
15     {
16         outputArea = new JTextArea();
17
18         Container container = getContentPane();
19         container.add( outputArea );
20
21         // calculate the factorials of 0 through 10
22         for ( long counter = 0; counter <= 10; counter++ )
23             outputArea.append( counter + "! = " +
24                               factorial( counter ) + "\n" );
25
26     } // end method init
27
28     // Recursive definition of method factorial
29     public long factorial( long number )
30     {
31         // base case
32         if ( number <= 1 )
33             return 1;
34
35         // recursive step
36         else
37             return number * factorial( number - 1 );
38
39     } // end method factorial
40
41 } // end class FactorialTest

```

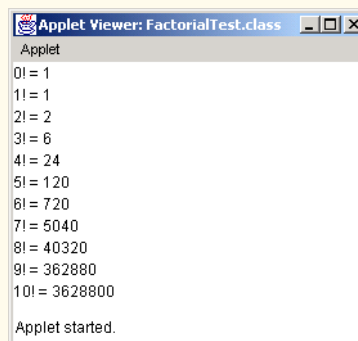


Fig. 6.12 Calculating factorials with a recursive method (part 2 of 2).

Method **factorial** (line 29) receives a parameter of type **long** and returns a result of type **long**. As can be seen in Fig. 6.12, factorial values become large quickly. We chose data type **long** so the program can calculate factorials greater than 20!. Unfortunately, the

factorial method produces large values so quickly that even **long** does not help us print many factorial values before the values exceed the size that can be stored in a **long** variable.

We explore in the exercises the fact that **float** and **double** may ultimately be needed by users desiring to calculate factorials of larger numbers. This situation points to a weakness in most programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications. As we will see in Chapter 9, Object-Oriented Programming, Java is an extensible language that allows us to create arbitrarily large integers if we wish. In fact, package **java.math** provides two classes—**BigInteger** and **BigDecimal**—explicitly for mathematical calculations of arbitrary precision that cannot be represented with Java's primitive data types.



Common Programming Error 6.15

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

6.13 Example Using Recursion: The Fibonacci Series

The Fibonacci series,

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618.... This number, too, repeatedly occurs in nature and has been called the *golden ratio*, or the *golden mean*. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden-mean length/width ratio.

The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

Note that there are two base cases for the Fibonacci calculation: **fibonacci(0)** is defined to be 0, and **fibonacci(1)** is defined to be 1. The applet of Fig. 6.13 calculates the i^{th} Fibonacci number recursively, using method **fibonacci** (lines 68–78). The applet enables the user to input an integer in a **JTextField**. The value input indicates the i^{th} Fibonacci number to calculate. When the user presses the *Enter* key, method **actionPerformed** executes in response to the user interface event and calls method **fibonacci** to calculate the specified Fibonacci number. Fibonacci numbers tend to become large quickly. Therefore, we use data type **long** as the parameter type and the return type of **fibonacci**. In Fig. 6.13, the screen captures show the results of calculating several Fibonacci numbers.

Once again, method **init** of this applet creates the GUI components and attaches them to the applet's content pane. The layout manager for the content pane is set to **FlowLayout** at line 22.

The event handling in this example is similar to the event handling of the **Craps** applet in Fig. 6.9. Line 34 specifies that **this** applet should listen for events from the **JTextField numberField**. Remember, the **this** keyword enables the applet to refer to itself. So, in line 34, the applet is telling **numberField** that the applet should be notified (with a call to the applet's **actionPerformed** method) when an action event occurs in the **numberField**. In this example, the user presses the *Enter* key while typing in the **numberField** to generate the action event. A message is then sent to the applet (i.e., a method—**actionPerformed**—is called on the applet) indicating that the user of the program has interacted with one of the program's GUI components (**numberField**). Remember that the statement to register the applet as the **numberField**'s listener will compile only if the applet class also implements **ActionListener** (line 12).

```

1  // Fig. 6.13: FibonacciTest.java
2  // Recursive fibonacci method
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class FibonacciTest extends JApplet
12     implements ActionListener {
13
14     JLabel numberLabel, resultLabel;
15     JTextField numberField, resultField;
16
17     // set up applet's GUI
18     public void init()
19     {
20         // obtain content pane and set its layout to FlowLayout
21         Container container = getContentPane();
22         container.setLayout( new FlowLayout() );
23
24         // create numberLabel and attach it to content pane
25         numberLabel =
26             new JLabel( "Enter an integer and press Enter" );
27         container.add( numberLabel );
28
29         // create numberField and attach it to content pane
30         numberField = new JTextField( 10 );
31         container.add( numberField );
32
33         // register this applet as numberField's ActionListener
34         numberField.addActionListener( this );
35
36         // create resultLabel and attach it to content pane
37         resultLabel = new JLabel( "Fibonacci value is" );
38         container.add( resultLabel );
39

```

Fig. 6.13 Recursively generating Fibonacci numbers (part 1 of 3).

```

40      // create numberField, make it uneditable
41      // and attach it to content pane
42      resultField = new JTextField( 15 );
43      resultField.setEditable( false );
44      container.add( resultField );
45
46  } // end method init
47
48  // obtain user input and call method fibonacci
49  public void actionPerformed((ActionEvent e)
50  {
51      long number, fibonacciValue;
52
53      // obtain user's input and conver to long
54      number = Long.parseLong( numberField.getText() );
55
56      showStatus( "Calculating ..." );
57
58      // calculate fibonacci value for number user input
59      fibonacciValue = fibonacci( number );
60
61      // indicate processing complete and display result
62      showStatus( "Done." );
63      resultField.setText( Long.toString( fibonacciValue ) );
64
65  } // end method actionPerformed
66
67  // Recursive definition of method fibonacci
68  public long fibonacci( long n )
69  {
70      // base case
71      if ( n == 0 || n == 1 )
72          return n;
73
74      // recursive step
75      else
76          return fibonacci( n - 1 ) + fibonacci( n - 2 );
77
78  } // end method fibonacci
79
80 } // end class FibonacciTest

```

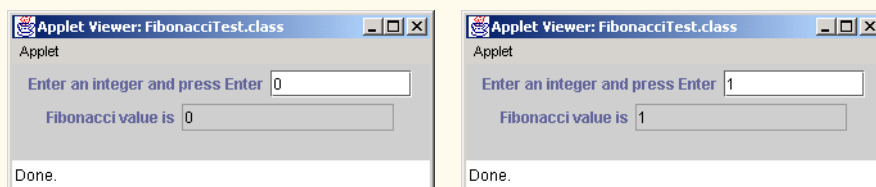


Fig. 6.13 Recursively generating Fibonacci numbers (part 2 of 3).

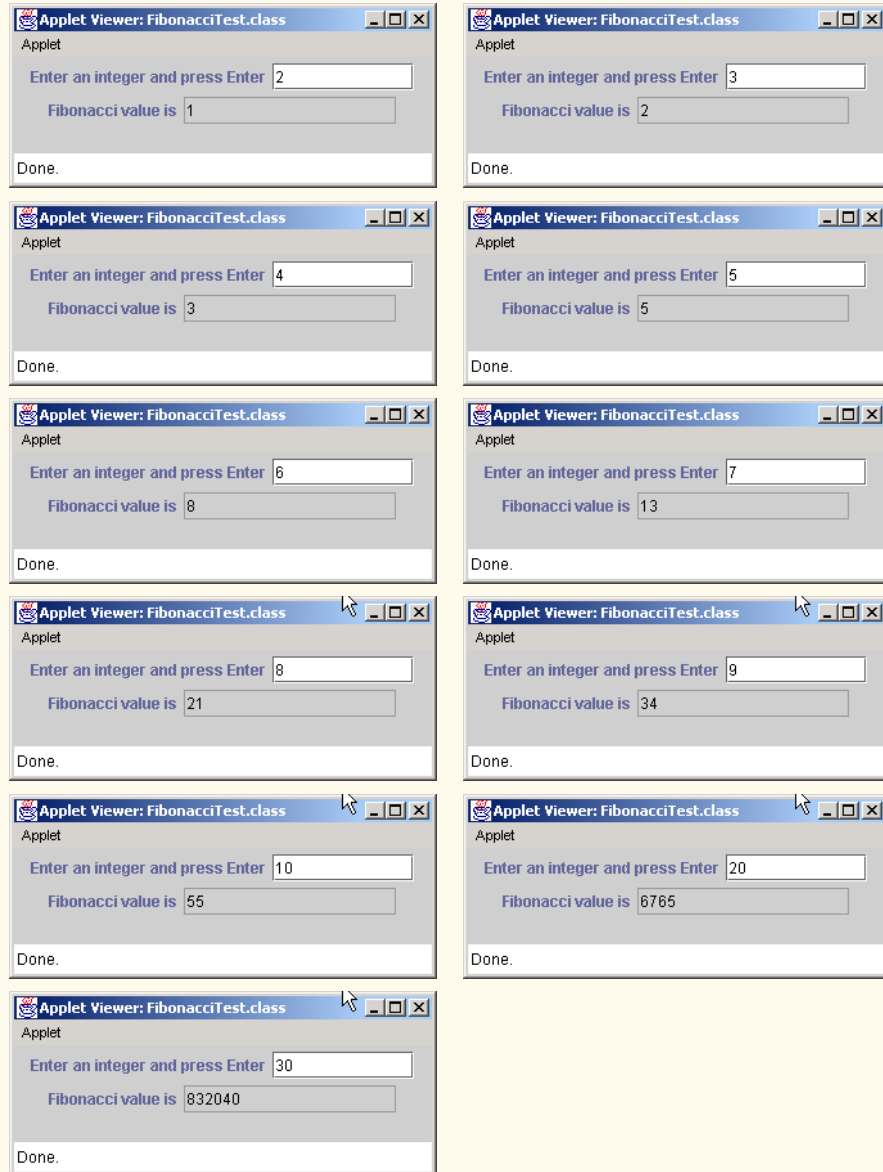


Fig. 6.13 Recursively generating Fibonacci numbers (part 3 of 3).

The call to **fibonacci** (line 59) from **actionPerformed** is not a recursive call, but all subsequent calls to **fibonacci** performed from the body of **fibonacci** are recursive. Each time **fibonacci** is invoked, it immediately tests for the base case—**n** equal to 0 or 1. If this condition is true, **fibonacci** returns **n** (fibonacci(0) is 0, and fibonacci(1) is 1). Interestingly, if **n** is greater than 1, the recursion step generates *two* recursive calls, each for a slightly simpler problem than the original call to **fibonacci**.

Figure 6.14 shows how method **fibonacci** evaluates **fibonacci(3)**. In the figure, **f** is an abbreviation for **fibonacci**.

Figure 6.14 raises some interesting issues about the order in which Java compilers evaluate the operands of operators. This issue is different than the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence. From Fig. 6.14, it appears that while **f(3)** is being evaluated, two recursive calls will be made, namely **f(2)** and **f(1)**. But in what order will these calls be made? Most programmers assume that the operands will be evaluated left to right. In Java, this assumption is true.

The C and C++ languages (on which many of Java's features are based) do not specify the order in which the operands of most operators (including **+**) are evaluated. Therefore, the programmer can make no assumption in those languages about the order in which the calls in this example execute. The calls could, in fact, execute **f(2)** first and **f(1)** second, or the calls could be executed in the reverse order: **f(1)**, then **f(2)**. In this program and in most other programs, it turns out that the final result would be the same for either case. But in some programs, the evaluation of an operand may have *side effects* that could affect the final result of the expression.

The Java language specifies that the order of evaluation of the operands is from left to right. Thus, the method calls are in fact **f(2)** first and **f(1)** second.



Good Programming Practice 6.8

Do not write expressions that depend on the order of evaluation of the operands of an operator. Use of such expressions often results in programs that are difficult to read, debug, modify and maintain.

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the **fibonacci** method that does not match one of the base cases (i.e., 0 or 1) results in two more recursive calls to the **fibonacci** method. This set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 using the program in Fig. 6.13 requires 21,891 calls to the **fibonacci** method; calculating the Fibonacci value of 30 requires 2,692,537 calls to the **fibonacci** method.

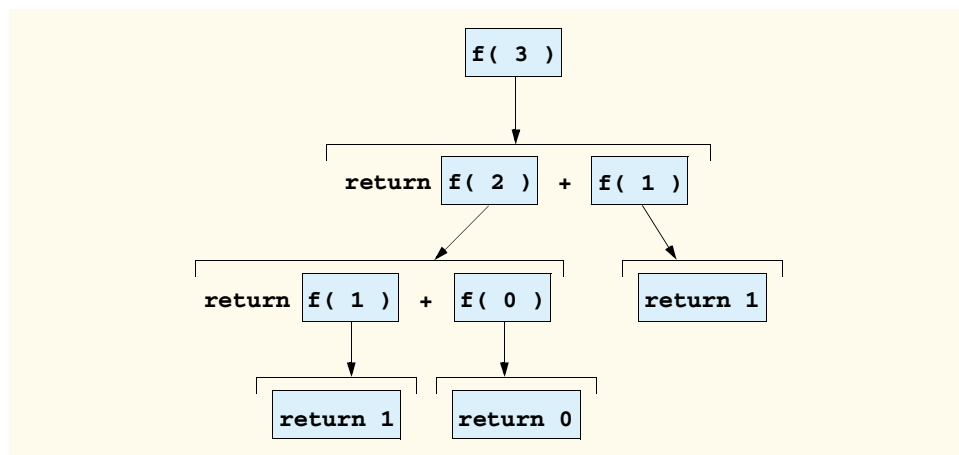


Fig. 6.14 Set of recursive calls to method **fibonacci** (**f** in this diagram).

As you try calculate larger Fibonacci values, you will notice that each consecutive Fibonacci number you ask the applet to calculate results in a substantial increase in calculation time and number of calls to the **fibonacci** method. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls. As you can see, the number of calls to **fibonacci** is increasing quickly—1,664,080 additional calls between Fibonacci values of 30 and 31 and 2,692,538 additional calls between Fibonacci values of 31 and 32. This difference in number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the number of calls for Fibonacci values between 30 and 31. Problems of this nature humble even the world's most powerful computers! In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks. Complexity issues are discussed in detail in the upper-level computer science curriculum course generally called “Algorithms.”

Performance Tip 6.3



Avoid Fibonacci-style recursive programs, which result in an exponential “explosion” of calls.



Testing and Debugging Tip 6.3

Try enhancing the Fibonacci program of Fig. 6.13 such that it calculates the approximate amount of time required to perform the calculation. For this purpose, call **static System** method **getCurrentTimeMillis**, which takes no arguments and returns the computer's current time in milliseconds. Call this method twice—once before the call to **fibonacci** and once after the call to **fibonacci**. Save each of these values and calculate the difference in the times to determine how many milliseconds were required to perform the calculation. Display this result.

6.14 Recursion vs. Iteration

In the previous sections, we studied two methods that can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why the programmer might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure (such as **for**, **while** or **do/while**); recursion uses a selection structure (such as **if**, **if/else** or **switch**). Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many negatives. It repeatedly invokes the mechanism and, consequently the overhead, of method calls. This repetition can be expensive in terms of both processor time and memory space. Each recursive call causes another copy of the method (actually, only the method's variables) to be created; this set of copies can consume considerable

memory space. Iteration normally occurs within a method, so the overhead of repeated method calls and extra memory assignment is omitted. So why choose recursion?



Software Engineering Observation 6.10

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Often, a recursive approach can be implemented with few lines of code and a corresponding iterative approach may take large amounts of code. Another reason to choose a recursive solution is that an iterative solution may not be apparent.



Performance Tip 6.4

Avoid using recursion in situations requiring performance. Recursive calls take time and consume additional memory.



Common Programming Error 6.16

Accidentally having a nonrecursive method call itself either directly or indirectly through another method can cause infinite recursion.

Most programming textbooks introduce recursion much later than we have done here. We feel that recursion is a sufficiently rich and complex topic that it is better to introduce it earlier and spread examples of it over the remainder of the text. Figure 6.15 summarizes the recursion examples and exercises in this text.

Chapter	Recursion examples and exercises
6	Factorial method Fibonacci method Greatest common divisor Sum of two integers Multiply two integers Raising an integer to an integer power Towers of Hanoi Visualizing recursion
7	Sum the elements of an array Print an array Print an array backward Check if a string is a palindrome Minimum value in an array Selection sort Eight Queens Linear search Binary search Quicksort Maze traversal
10	Printing a string input at the keyboard backward

Fig. 6.15 Summary of recursion examples and exercises in this text (part 1 of 2).

Chapter	Recursion examples and exercises
19	Linked-list insert Linked-list delete Search a linked list Print a linked list backward Binary-tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree

Fig. 6.15 Summary of recursion examples and exercises in this text (part 2 of 2).

Let us reconsider some observations we make repeatedly throughout this book. Good software engineering is important. High performance is often important. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing larger and more complex software systems. High performance in these systems is key to realizing the systems of the future, which will place ever greater computing demands on hardware. Where do methods fit in here?



Software Engineering Observation 6.11

Modularizing programs in a neat, hierarchical manner promotes good software engineering. But it has a price.



Performance Tip 6.5

A heavily modularized program—as compared with a monolithic (i.e., one-piece) program without methods—makes potentially large numbers of method calls, which consume execution time and space on a computer’s processor(s). But monolithic programs are difficult to program, test, debug, maintain and evolve.

So modularize your programs judiciously, always keeping in mind the delicate balance between performance and good software engineering.

6.15 Method Overloading

Java enables several methods of the same name to be defined, as long as the methods have different sets of parameters (based on the number of parameters, the types of the parameters and the order of the parameters). This characteristic is called *method overloading*. When an overloaded method is called, the Java compiler selects the proper method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the same name that perform similar tasks, but on different data types.



Good Programming Practice 6.9

Overloading methods that perform closely related tasks can make programs more readable and understandable.

Figure 6.16 uses overloaded method **square** to calculate the square of an **int** and the square of a **double**.

```
1 // Fig. 6.16: MethodOverload.java
2 // Using overloaded methods
3
4 // Java core packages
5 import java.awt.Container;
6
7 // Java extension packages
8 import javax.swing.*;
9
10 public class MethodOverload extends JApplet {
11
12     // set up GUI and call versions of method square
13     public void init()
14     {
15         JTextArea outputArea = new JTextArea();
16         Container container = getContentPane();
17         container.add( outputArea );
18
19         outputArea.setText(
20             "The square of integer 7 is " + square( 7 ) +
21             "\nThe square of double 7.5 is " + square( 7.5 ) );
22     }
23
24     // square method with int argument
25     public int square( int intValue )
26     {
27         System.out.println(
28             "Called square with int argument: " + intValue );
29
30         return intValue * intValue;
31     }
32     // end method square with int argument
33
34     // square method with double argument
35     public double square( double doubleValue )
36     {
37         System.out.println(
38             "Called square with double argument: " + doubleValue );
39
40         return doubleValue * doubleValue;
41     }
42     // end method square with double argument
43
44 } // end class MethodOverload
```

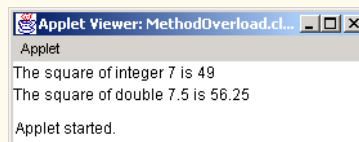


Fig. 6.16 Using overloaded methods (part 1 of 2).

```
Called square with int argument: 7
Called square with double argument: 7.5
```

Fig. 6.16 Using overloaded methods (part 2 of 2).

Overloaded methods are distinguished by their *signature*—a combination of the method’s name and its parameter types. If the Java compiler looked only at method names during compilation, the code in Fig. 6.16 would be ambiguous—the compiler would not know how to distinguish between the two **square** methods. Logically, the compiler uses longer “mangled” or “decorated” names that include the original method name, the types of each parameter and the exact order of the parameters to determine if the methods in a class are unique in that class.

For example, in Fig. 6.16, the compiler might use the logical name “square of int” for the **square** method that specifies an **int** parameter and “square of double” for the **square** method that specifies a **double** parameter. If a method **foo**’s definition begins as

```
void foo( int a, float b )
```

then the compiler might use the logical name “foo of int and float.” If the parameters are specified as

```
void foo( float a, int b )
```

then the compiler might use the logical name “foo of float and int.” Note that the order of the parameters is important to the compiler. The preceding two **foo** methods are considered to be distinct by the compiler.

The logical names of methods used by the compiler did not mention the return types of the methods, because methods cannot be distinguished by return type. The program in Fig. 6.17 illustrates the compiler errors generated when two methods have the same signature and different return types. Overloaded methods can have different return types, but must have different parameter lists. Also, overloaded methods need not have the same number of parameters.



Common Programming Error 6.17

Creating overloaded methods with identical parameter lists and different return types is a syntax error.

```
1 // Fig. 6.17: MethodOverload.java
2 // Overloaded methods with identical signatures and
3 // different return types.
4
5 // Java extension packages
6 import javax.swing.JApplet;
7
```

Fig. 6.17 Compiler error messages generated from overloaded methods with identical parameter lists and different return types (part 1 of 2).

```

8  public class MethodOverload extends JApplet {
9
10     // first definition of method square with double argument
11     public int square( double x )
12     {
13         return x * x;
14     }
15
16     // second definition of method square with double argument
17     // causes syntax error
18     public double square( double y )
19     {
20         return y * y;
21     }
22
23 } // end class MethodOverload

```

```

MethodOverload.java:18: square(double) is already defined in
MethodOverload
    public double square( double y )
                        ^
MethodOverload.java:13: possible loss of precision
found   : double
required: int
    return x * x;
        ^
2 errors

```

Fig. 6.17 Compiler error messages generated from overloaded methods with identical parameter lists and different return types (part 2 of 2).

6.16 Methods of Class JApplet

We have written many applets to this point in the text, but we have not yet discussed the key methods of class **JApplet** that the applet container calls during the execution of an applet. Figure 6.18 lists the key methods of class **JApplet**, specifies when they get called and explains the purpose of each method.

These **JApplet** methods are defined by the Java API to do nothing unless you provide a definition in your applet's class definition. If you would like to use one of these methods in an applet you are defining, you *must* define the first line of each method as shown in Fig. 6.18. Otherwise, the applet container will not call your versions of the methods during the applet's execution. Defining the methods as discussed here is known as *overriding* the original method definition. The applet container will call the overridden version of a method for your applet before it attempts to call the default versions inherited from **JApplet**. Overriding is discussed in detail in Chapter 9.



Common Programming Error 6.18

Providing a definition for one of the **JApplet** methods **init**, **start**, **paint**, **stop** or **destroy** that does not match the method headers shown in Figure 6.18 results in a method that will not be called automatically during execution of the applet.

Method	When the method is called and its purpose
--------	---

public void init()

This method is called once by the **appletviewer** or browser when an applet is loaded for execution. It performs initialization of an applet. Typical actions performed here are initialization of instance variables and GUI components of the applet, loading of sounds to play or images to display (see Chapter 18, Multimedia) and creation of threads (see Chapter 15, Multithreading).

public void start()

This method is called after the **init** method completes execution and every time the user of the browser returns to the HTML page on which the applet resides (after browsing another HTML page). This method performs any tasks that must be completed when the applet is loaded for the first time into the browser and that must be performed every time the HTML page on which the applet resides is revisited. Typical actions performed here include starting an animation see (Chapter 18, Multimedia) and starting other threads of execution (see Chapter 15, Multithreading).

public void paint(Graphics g)

This method is called after the **init** method completes execution and the **start** method has started executing to draw on the applet. It is also called automatically every time the applet needs to be repainted. For example, if the user covers the applet with another open window on the screen then uncovers the applet, the **paint** method is called. Typical actions performed here involve drawing with the **Graphics** object **g** that is automatically passed to the **paint** method for you.

public void stop()

This method is called when the applet should stop executing—normally, when the user of the browser leaves the HTML page on which the applet resides. This method performs any tasks that are required to suspend the applet's execution. Typical actions performed here are to stop execution of animations and threads.

public void destroy()

This method is called when the applet is being removed from memory—normally, when the user of the browser exits the browsing session. This method performs any tasks that are required to destroy resources allocated to the applet.

Fig. 6.18 JApplet methods that the applet container calls during an applet's execution .

Method **repaint** is also of interest to many applet programmers. The applet's **paint** method normally is called by the applet container. What if you would like to change the appearance of the applet in response to the user's interactions with the applet? In such situations, you may want to call **paint** directly. However, to call **paint**, we must pass it the **Graphics** parameter it expects. This requirement poses a problem for us. We do not have a **Graphics** object at our disposal to pass to **paint**. (We discuss this issue in Chapter 18, Multimedia.) For this reason, class **JApplet** provides method **repaint**. The statement

```
repaint();
```

obtains the **Graphics** object for you and invokes another method, called **update**. Method **update** invokes method **paint** and passes to it the **Graphics** object. The **repaint** method is discussed in detail in Chapter 18, “Multimedia.”

6.17 (Optional Case Study) Thinking About Objects: Identifying Class Operations

In the “Thinking About Objects” sections at the ends of Chapters 3, 4 and 5, we performed the first few steps in the object-oriented design for our elevator simulator. In Chapter 3, we identified the classes we need to implement. In Chapter 4, we created a class diagram that models the structure of our system. In Chapter 5, we examined objects’ states and modeled objects’ activities and state transitions.

In this section, we concentrate on determining the class *operations* (or *behaviors*) needed to implement the elevator simulator. In Chapter 7, we concentrate on the collaborations (interactions) between objects of our classes.

An operation of a class is a service that the class provides to “clients” (users) of that class. Consider the operations of some real-world classes. A radio’s operations include setting its station and volume (typically invoked by a person adjusting the radio’s controls). A car’s operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal and/or releasing the gas pedal), turning and shifting gears.

We can derive many of the operations of each class directly from the problem statement. To do so, we examine the verbs and verb phrases in the problem statement. We then relate each of these to particular classes in our system (Fig. 6.20). Many of the verb phrases in Fig. 6.20 help us determine the operations of our classes.

Class	Verb phrases
Elevator	moves to other floor, arrives at a floor, resets elevator button, rings elevator bell, signals its arrival, opens its door, closes its door
ElevatorShaft	turns off light, turns on light, resets floor button
Person	walks on floor, presses floor button, presses elevator button, rides elevator, enters elevator, exits elevator
Floor	[none in the problem statement]
FloorButton	requests elevator
ElevatorButton	closes elevator door, signals elevator to move to opposite floor
FloorDoor	signals person to enter elevator (by opening)
ElevatorDoor	signals person to exit elevator (by opening), opens floor door, closes floor door
Bell	[none in the problem statement]
Light	[none in the problem statement]
ElevatorModel	creates person

Fig. 6.20 Verb phrases for each class in simulator.

To create operations, we examine the verb phrases listed with each class. The phrase “moves to other floor” listed with class **Elevator** refers to the activity in which the elevator moves between floors. Should “moves” be an operation of class **Elevator**? The elevator decides to move in response to a button press. A button *signals* the elevator to move, but a button does not actually *move* the elevator—therefore, “moves to other floor” does not correspond to an operation. (We include the operations for informing the elevator to move to the other floor later in the discussion, when we discuss the verb phrases associated with the buttons.) The “arrives at a floor” phrase is also not an operation, because the elevator itself decides when to arrive on the floor after five seconds of travel.

The “resets elevator button” phrase associated with class **Elevator** implies that the elevator informs the elevator button to reset. Therefore, class **ElevatorButton** needs an operation to provide this service to the elevator. We place this operation (**resetButton**) in the bottom compartment of class **ElevatorButton** in our class diagram (Fig. 6.21).

We represent the names of the operations as method names (by following the names with a pair of parentheses) and include the return type after the colon:

resetButton() : void

The parentheses can contain a comma-separated list of the parameters that the operation takes—in this case, none. For the moment, most of our operations have no parameters and a **void** return type; this might change as our design and implementation processes proceed.

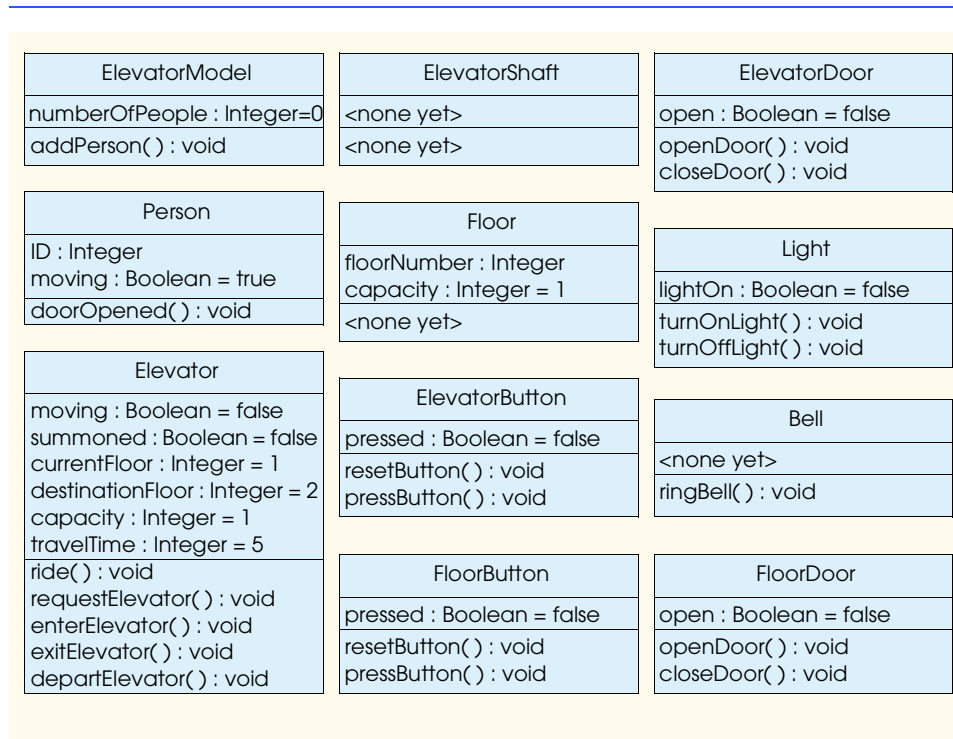


Fig. 6.21 Classes with attributes and operations.

From the “ring the elevator bell” phrase listed with class **Elevator**, we conclude that class **Bell** should have an operation that provides a service—namely, ringing. We list the **ringBell** operation under class **Bell**.

When arriving at a floor, the elevator “signals its arrival” to the doors. The elevator door responds by opening, as implied by the phrase “opens [the elevator’s] door” associated with class **Elevator**. Therefore, class **ElevatorDoor** needs an operation that opens its door. We place the **openDoor** operation in the bottom compartment of this class. The phrase “closes [the elevator’s] door” indicates that class **ElevatorDoor** needs an operation that closes its door, so we place the **closeDoor** operation in the same compartment.

Class **ElevatorShaft** lists “turns off light” and “turns on light” in its verb-phrases column, so we create the **turnOffLight** and **turnOnLight** operations and list them under class **Light**. The “resets floor button” phrase implies that the elevator instructs a floor button to reset. Therefore, class **FloorButton** needs a **resetButton** operation.

The phrase “walks on floor” listed by class **Person** is not an operation, because a person decides to walk across the floor in response to that person’s creation. However, the phrases “presses floor button” and “presses elevator button” are operations pertaining to the button classes. We therefore place the **pressButton** operation under classes **FloorButton** and **ElevatorButton** in our class diagram (Fig. 6.21). The phrase “rides elevator” implies that **Elevator** needs a method that allows a person to ride the elevator, so we place operation **ride** in the bottom compartment of **Elevator**. The “enters elevator” and “exits elevator” phrases listed with class **Person** suggest that class **Elevator** needs operations that correspond to these actions.¹ We place operations **enterElevator** and **exitElevator** in the bottom compartment of class **Elevator**.

The “requests elevator” phrase listed under class **FloorButton** implies that class **Elevator** needs a **requestElevator** operation. The phrase “signals elevator to move to opposite floor” listed with class **ElevatorButton** implies that **ElevatorButton** informs **Elevator** to depart. Therefore, the **Elevator** needs to provide a “departure” service; we place a **departElevator** operation in the bottom compartment of **Elevator**.

The phrases listed with classes **FloorDoor** and **ElevatorDoor** mention that the doors—by opening—signal a **Person** object to enter or exit the elevator. Specifically, a door informs a person that the door has opened. (The person then enters or exits the elevator, accordingly.) We place the **doorOpened** operation in the bottom compartment for class **Person**. In addition, the **ElevatorDoor** opens and closes the **FloorDoor**, so we assign **openDoor** and **closeDoor** to the bottom compartment of class **FloorDoor**.

Lastly, the “creates person” action associated with class **ElevatorModel** refers to creating a **Person** object and adding it to the simulation. Although we can require **ElevatorModel** to send a “create person” and an “add person” message, an object of class **Person** cannot respond to these messages, because that object does not yet exist. We discuss new objects when we consider implementation in Chapter 8. We place the operation **addPerson** in the bottom compartment of **ElevatorModel** in the class diagram of Fig. 6.21 and anticipate that the application user will invoke this operation.

1. At this point, we can only guess what these operations do. For example, perhaps these operations model real-world elevators, some of which have sensors that detect when passengers enter and exit. For now, we simply list these operations. We will discover what, if any, actions these operations perform as we continue our design process.

For now, we do not concern ourselves with operation parameters or return types; we attempt to gain only a basic understanding of the operations of each class. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary—and we might determine that some of our class operations need non-**void** return types.

SUMMARY

- The best way to develop and maintain a large program is to divide it into several smaller modules. Modules are written in Java as classes and methods.
- A method is invoked by a method call. The method call mentions the method by name and provides arguments in parentheses that the called method requires to perform its task. If the method is in another class, the call must be preceded by a reference name and a dot operator. If the method is **static**, it must be preceded by a class name and a dot operator.
- Each argument of a method may be a constant, a variable or an expression.
- A local variable is known only in a method definition. Methods are not allowed to know the implementation details of any other method (including its local variables).
- The on-screen display area for a **JApplet** has a content pane to which the GUI components must be attached so they can be displayed at execution time. The content pane is an object of class **Container** from the **java.awt** package.
- Method **getContentPane** of class **JApplet** returns a reference to the applet's content pane.
- The general format for a method definition is

```

return-value-type method-name ( parameter-list )
{
    declarations and statements
}

```

The *return-value-type* states the type of the value returned to the calling method. If a method does not return a value, the *return-value-type* is **void**. The *method-name* is any valid identifier. The *parameter-list* is a comma-separated list containing the declarations of the variables that will be passed to the method. If a method does not receive any values, *parameter-list* is empty. The method body is the set of *declarations and statements* that constitute the method.

- The arguments passed to a method should match in number, type and order with the parameters in the method definition.
- When a program encounters a method, control transfers from the point of invocation to the called method, the method executes and control returns to the caller.
- A called method can return control to the caller in one of three ways. If the method does not return a value, control returns at the method-ending right brace or by executing the statement

```
return;
```

If the method does return a value, the statement

```
return expression;
```

returns the value of *expression*.

- There are three ways to call a method—the method name by itself, a reference to an object followed by the dot (.) operator and the method name, and a class name followed by the dot (.) operator and a method name. The last syntax is for **static** methods of a class.

- An important feature of method definitions is the coercion of arguments. In many cases, argument values that do not correspond precisely to the parameter types in the method definition are converted to the proper type before the method is called. In some cases, these conversions can lead to compiler errors if Java's promotion rules are not followed.
- The promotion rules specify how types can be converted to other types without losing data. The promotion rules apply to mixed-type expressions. The type of each value in a mixed-type expression is promoted to the "highest" type in the expression.
- Method **Math.random** generates a double value from 0.0 up to, but not including, 1.0. Values produced by **Math.random** can be scaled and shifted to produce values in a range.
- The general equation for scaling and shifting a random number is

```
n = a + (int) ( Math.random() * b );
```

where **a** is the shifting value (the first number in the desired range of consecutive integers) and **b** is the scaling factor (the width of the desired range of consecutive integers).

- A class can inherit existing attributes and behaviors (data and methods) from another class specified to the right of keyword **extends** in the class definition. In addition, a class can implement one or more *interfaces*. An interface specifies one or more behaviors (i.e., methods) that you must define in your class definition.
- The interface **ActionListener** specifies that a class must define a method with the first line

```
public void actionPerformed((ActionEvent actionEvent )
```

- The task of method **actionPerformed** is to process a user's interaction with a GUI component that generates an action event. This method is called in response to the user interaction (the event). This process is called *event handling*. The event handler is the **actionPerformed** method, which is called in response to the event. This style of programming is known as *event-driven programming*.
- Keyword **final** declares constant variables. Constant variables must be initialized before they are used in a program. Constant variables are often called *named constants* or *read-only variables*.
- A **JLabel** contains a string of characters to be displayed on the screen. Normally, a **JLabel** indicates the purpose of another GUI element on the screen.
- **TextFields** get information from the user or displays information on the screen.
- When the user presses a **JButton**, the program normally responds by performing a task.
- **Container** method **setLayout** defines the layout manager for the applet's user interface. Layout managers are provided to arrange GUI components on a **Container** for presentation purposes.
- **FlowLayout** is the simplest layout manager. GUI components are placed on a **Container** from left to right in the order in which they are attached to the **Container** with method **add**. When the edge of the container is reached, components are continued on the next line.
- Before any event can be processed, each GUI component must know which object in the program defines the event-handling method that will be called when an event occurs. Method **addActionListener** is used to tell a **JButton** or **JTextField** that another object is listening for action events and defines method **actionPerformed**. This procedure is called *registering the event handler with the GUI component*. To respond to an action event, we must define a class that implements **ActionListener** and defines method **actionPerformed**. Also, we must register the event handler with the GUI component.
- Method **showStatus** displays a **String** in the applet container's status bar.

- Each variable identifier has the attributes duration (lifetime) and scope. An identifier's duration determines when that identifier exists in memory. An identifier's scope is where the identifier can be referenced in a program.
- Identifiers that represent local variables in a method have automatic duration. Automatic-duration variables are created when program control reaches their declaration; they exist while the block in which they are declared is active; and they are destroyed when the block in which they are declared is exited.
- Java also has identifiers of static duration. Variables and references of static duration exist from the point at which the class in which they are defined is loaded into memory for execution until the program terminates.
- The scopes for an identifier are class scope and block scope. An instance variable declared outside any method has class scope. Such an identifier is "known" in all methods of the class. Identifiers declared inside a block have block scope. Block scope ends at the terminating right brace (**}**) of the block.
- Local variables declared at the beginning of a method have block scope, as do method parameters, which are considered to be local variables of the method.
- Any block may contain variable declarations.
- A recursive method is a method that calls itself, either directly or indirectly.
- If a recursive method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two or more conceptual pieces: A piece that the method knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original problem, the method launches a recursive call to work on the smaller problem.
- For recursion to terminate, the sequence of smaller and smaller problems must converge to the base case. When the method recognizes the base case, the result is returned to the previous method call, and a sequence of returns ensues all the way up the line until the original call of the method returns the final result.
- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges to the base case.
- Recursion repeatedly invokes the mechanism and, consequently the overhead, of method calls. This repetition can be expensive in terms of both processor time and memory space.
- The user presses the *Enter* key while typing in a **JTextField** to generate an action event. The event handling for this GUI component is set up like a **JButton**: A class must be defined that implements **ActionListener** and defines method **actionPerformed**. Also, the **JTextField**'s **addActionListener** method must be called to register the event.
- It is possible to define methods with the same name, but different parameter lists. This feature is called method overloading. When an overloaded method is called, the compiler selects the proper method by examining the arguments in the call.
- Overloaded methods can have different return values and must have different parameter lists. Two methods differing only by return type will result in a syntax error.

- The applet's **init** method is called once by the applet container when an applet is loaded for execution. It performs initialization of an applet. The applet's **start** method is called after the **init** method completes execution and every time the user of the browser returns to the HTML page on which the applet resides (after browsing another HTML page).
- The applet's **paint** method is called after the **init** method completes execution and the **start** method has started executing to draw on the applet. It is also called every time the applet needs to be repainted.
- The applet's **stop** method is called when the applet should suspend execution—normally, when the user of the browser leaves the HTML page on which the applet resides.
- The applet's **destroy** method is called when the applet is being removed from memory—normally, when the user of the browser exits the browsing session.
- Method **repaint** can be called in an applet to cause a fresh call to **paint**. Method **repaint** invokes another method called **update** and passes it the **Graphics** object. The **update** method invokes the **paint** method and passes it the **Graphics** object.

TERMINOLOGY

ActionEvent class

ActionListener interface

actionPerformed method

argument in a method call

automatic duration

automatic variable

base case in recursion

block

block scope

call a method

called method

caller

calling method

class

class scope

coercion of arguments

constant variable

copy of a value

destroy method of **JApplet**

divide and conquer

duration

element of chance

factorial method

final

FlowLayout class

init method of **JApplet**

invoke a method

iteration

Java API (Java class library)

JButton class of package **javax.swing**

JLabel class of package **javax.swing**

TextField class of package **javax.swing** **setLayout** method of **JApplet**

local variable

Math class methods

Math.E

Math.PI

Math.random method

method

method call

method-call operator, **()**

method declaration

method definition

method overloading

mixed-type expression

modular program

named constant

overloading

paint method of **JApplet**

parameter in a method definition

programmer-defined method

promotion rules

random-number generation

read-only variable

recursion

recursion step

recursive call

recursive method

reference parameter

reference types

repaint method of **JApplet**

return

return-value type

scaling

scope

shifting

showStatus method of **JApplet**

signature

simulation

software engineering

software reusability

start method of **JApplet**

static storage duration

stop method of **JApplet****update** method of **JApplet****void****SELF-REVIEW EXERCISES****6.1** Fill in the blanks in each of the following statements:

- a) Program modules in Java are called _____ and _____.
- b) A method is invoked with a _____.
- c) A variable known only within the method in which it is defined is called a _____.
- d) The _____ statement in a called method can be used to pass the value of an expression back to the calling method.
- e) The keyword _____ indicates that a method does not return a value.
- f) The _____ of an identifier is the portion of the program in which the identifier can be used.
- g) The three ways to return control from a called method to a caller are _____, _____ and _____.
- h) The _____ method is invoked once when an applet begins execution.
- i) The _____ method produces random numbers.
- j) The _____ method is invoked each time the user of a browser revisits the HTML page on which an applet resides.
- k) The _____ method is invoked to draw on an applet.
- l) Variables declared in a block or in a method's parameter list are of _____ duration.
- m) The _____ method invokes the applet's **update** method, which in turn invokes the applet's **paint** method.
- n) The _____ method is invoked for an applet each time the user of a browser leaves an HTML page on which the applet resides.
- o) A method that calls itself either directly or indirectly is a _____ method.
- p) A recursive method typically has two components: one that provides a means for the recursion to terminate by testing for a _____ case and one that expresses the problem as a recursive call for a slightly simpler problem than does the original call.
- q) In Java, it is possible to have various methods with the same name that each operate on different types and/or numbers of arguments. This feature is called method _____.
- r) The _____ qualifier is used to declare read-only variables.

6.2 For the following program, state the scope (either class scope or block scope) of each of the following elements:

- a) the variable **x**.
- b) the variable **y**.
- c) the method **cube**.
- d) the method **paint**.
- e) the variable **yPos**.

```

1  public class CubeTest extends JApplet {
2      int x;
3
4      public void paint( Graphics g )
5      {
6          int yPos = 25;

```

```

7
8     for ( x = 1; x <= 10; x++ ) {
9         g.drawString( cube( x ), 25, yPos );
10        yPos += 15;
11    }
12 }
13
14 public int cube( int y )
15 {
16     return y * y * y;
17 }
18 }

```

6.3 Write an application that tests if the examples of the math-library method calls shown in Fig. 6.2 actually produce the indicated results.

6.4 Give the method header for each of the following methods:

- Method **hypotenuse**, which takes two double-precision, floating-point arguments **side1** and **side2** and returns a double-precision, floating-point result.
- Method **smallest**, which takes three integers **x**, **y** and **z** and returns an integer.
- Method **instructions**, which does not take any arguments and does not return a value. [Note: Such methods are commonly used to display instructions to a user.]
- Method **intToFloat**, which takes an integer argument **number** and returns a floating-point result.

6.5 Find the error in each of the following program segments. Explain how to correct the error.

- ```

int g() {
 System.out.println("Inside method g");
 int h() {
 System.out.println("Inside method h");
 }
}

```
- ```

int sum( int x, int y ) {
    int result;
    result = x + y;
}

```
- ```

int sum(int n) {
 if (n == 0)
 return 0;
 else
 n + sum(n - 1);
}

```
- ```

void f( float a ); {
    float a;
    System.out.println( a );
}

```
- ```

void product() {
 int a = 6, b = 5, c = 4, result;
 result = a * b * c;
 System.out.println("Result is " + result);
 return result;
}

```

**6.6** Write a complete Java applet to prompt the user for the **double** radius of a sphere, and call method **sphereVolume** to calculate and display the volume of that sphere using the assignment

```
volume = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3)
```

The user should input the radius through a **JTextField**.

### ANSWERS TO SELF-REVIEW EXERCISES

**6.1** a) methods and classes. b) method call. c) local variable. d) **return**. e) **void**. f) scope. g) **return**; or **return expression**; or encountering the closing right brace of a method. h) **init**. i) **Math.random**. j) **start**. k) **paint**. l) automatic. m) **repaint**. n) **stop**. o) recursive. p) base. q) overloading. r) **final**.

**6.2** a) Class scope. b) Block scope. c) Class scope. d) Class scope. e) Block scope.

**6.3** The following solution demonstrates the **Math** class methods in Fig. 6.2:

```
1 // Exercise 6.3: MathTest.java
2 // Testing the Math class methods
3
4 public class MathTest {
5 public static void main(String args[])
6 {
7 System.out.println("Math.abs(23.7) = " +
8 Math.abs(23.7));
9 System.out.println("Math.abs(0.0) = " +
10 Math.abs(0.0));
11 System.out.println("Math.abs(-23.7) = " +
12 Math.abs(-23.7));
13 System.out.println("Math.ceil(9.2) = " +
14 Math.ceil(9.2));
15 System.out.println("Math.ceil(-9.8) = " +
16 Math.ceil(-9.8));
17 System.out.println("Math.cos(0.0) = " +
18 Math.cos(0.0));
19 System.out.println("Math.exp(1.0) = " +
20 Math.exp(1.0));
21 System.out.println("Math.exp(2.0) = " +
22 Math.exp(2.0));
23 System.out.println("Math.floor(9.2) = " +
24 Math.floor(9.2));
25 System.out.println("Math.floor(-9.8) = " +
26 Math.floor(-9.8));
27 System.out.println("Math.log(2.718282) = " +
28 Math.log(2.718282));
29 System.out.println("Math.log(7.389056) = " +
30 Math.log(7.389056));
31 System.out.println("Math.max(2.3, 12.7) = " +
32 Math.max(2.3, 12.7));
33 System.out.println("Math.max(-2.3, -12.7) = " +
34 Math.max(-2.3, -12.7));
35 System.out.println("Math.min(2.3, 12.7) = " +
36 Math.min(2.3, 12.7));
37 System.out.println("Math.min(-2.3, -12.7) = " +
38 Math.min(-2.3, -12.7));
```

```

39 System.out.println("Math.pow(2, 7) = " +
40 Math.pow(2, 7));
41 System.out.println("Math.pow(9, .5) = " +
42 Math.pow(9, .5));
43 System.out.println("Math.sin(0.0) = " +
44 Math.sin(0.0));
45 System.out.println("Math.sqrt(25.0) = " +
46 Math.sqrt(25.0));
47 System.out.println("Math.tan(0.0) = " +
48 Math.tan(0.0));
49 }
50 }

```

```

Math.abs(23.7) = 23.7
Math.abs(0.0) = 0
Math.abs(-23.7) = 23.7
Math.ceil(9.2) = 10
Math.ceil(-9.8) = -9
Math.cos(0.0) = 1
Math.exp(1.0) = 2.71828
Math.exp(2.0) = 7.38906
Math.floor(9.2) = 9
Math.floor(-9.8) = -10
Math.log(2.718282) = 1
Math.log(7.389056) = 2
Math.max(2.3, 12.7) = 12.7
Math.max(-2.3, -12.7) = -2.3
Math.min(2.3, 12.7) = 2.3
Math.min(-2.3, -12.7) = -12.7
Math.pow(2, 7) = 128
Math.pow(9, .5) = 3
Math.sin(0.0) = 0
Math.sqrt(25.0) = 5
Math.tan(0.0) = 0

```

- 6.4 a) `double hypotenuse( double side1, double side2 )`  
 b) `int smallest( int x, int y, int z )`  
 c) `void instructions()`  
 d) `float intToFloat( int number )`

- 6.5 a) Error: Method **h** is defined in method **g**.  
 Correction: Move the definition of **h** outside the definition of **g**.  
 b) Error: The method is supposed to return an integer, but does not.  
 Correction: Delete variable **result**, and place the statement  
     `return x + y;`  
 in the method, or add the following statement at the end of the method body:  
     `return result;`  
 c) Error: The result of `n + sum( n - 1 )` is not returned by this recursive method, resulting in a syntax error.  
 Correction: Rewrite the statement in the **else** clause as  
     `return n + sum( n - 1 );`

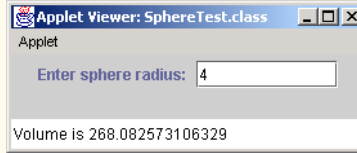
- d) Error: Both the semicolon after the right parenthesis that encloses the parameter list and redefining the parameter **a** in the method definition are incorrect.  
Correction: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration **float a;**.
- e) Error: The method returns a value when it is not supposed to.  
Correction: Change the return type to **int**.

**6.6** The following solution calculates the volume of a sphere using the radius entered by the user:

```

1 // Exercise 6.6: SphereTest.java
2
3 // Java core packages
4 import java.awt.*;
5 import java.awt.event.*;
6
7 // Java extension packages
8 import javax.swing.*;
9
10 public class SphereTest extends JApplet
11 implements ActionListener {
12
13 JLabel promptLabel;
14 JTextField inputField;
15
16 public void init()
17 {
18 Container container = getContentPane();
19 container.setLayout(new FlowLayout());
20
21 promptLabel = new JLabel("Enter sphere radius: ");
22 inputField = new JTextField(10);
23 inputField.addActionListener(this);
24 container.add(promptLabel);
25 container.add(inputField);
26 }
27
28 public void actionPerformed((ActionEvent actionEvent))
29 {
30 double radius =
31 Double.parseDouble(actionEvent.getActionCommand());
32
33 showStatus("Volume is " + sphereVolume(radius));
34 }
35
36 public double sphereVolume(double radius)
37 {
38 double volume =
39 (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);
40
41 return volume;
42 }
43 }

```



## EXERCISES

**6.7** What is the value of **x** after each of the following statements is performed?

- a) `x = Math.abs( 7.5 );`
- b) `x = Math.floor( 7.5 );`
- c) `x = Math.abs( 0.0 );`
- d) `x = Math.ceil( 0.0 );`
- e) `x = Math.abs( -6.4 );`
- f) `x = Math.ceil( -6.4 );`
- g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

**6.8** A parking garage charges a \$2.00 minimum fee to park for up to three hours. The garage charges an additional \$0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is \$10.00. Assume that no car parks for longer than 24 hours at a time. Write an applet that calculates and displays the parking charges for each customer who parked a car in this garage yesterday. You should enter in a **JTextField** the hours parked for each customer. The program should display the charge for the current customer and should calculate and display the running total of yesterday's receipts. The program should use the method **calculateCharges** to determine the charge for each customer.

**6.9** An application of method **Math.floor** is rounding a value to the nearest integer. The statement

```
y = Math.floor(x + .5);
```

will round the number **x** to the nearest integer and assign the result to **y**. Write an applet that reads **double** values and uses the preceding statement to round each of the numbers to the nearest integer. For each number processed, display both the original number and the rounded number.

**6.10** **Math.floor** may be used to round a number to a specific decimal place. The statement

```
y = Math.floor(x * 10 + .5) / 10;
```

rounds **x** to the tenths position (i.e., the first position to the right of the decimal point). The statement

```
y = Math.floor(x * 100 + .5) / 100;
```

rounds **x** to the hundredths position (i.e., the second position to the right of the decimal point). Write an applet that defines four methods to round a number **x** in various ways:

- a) `roundToInteger( number )`
- b) `roundToTenths( number )`
- c) `roundToHundredths( number )`
- d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

**6.11** Answer each of the following questions:

- What does it mean to choose numbers “at random?”
- Why is the **Math.random** method useful for simulating games of chance?
- Why is it often necessary to scale and/or shift the values produced by **Math.random**?
- Why is computerized simulation of real-world situations a useful technique?

**6.12** Write statements that assign random integers to the variable  $n$  in the following ranges:

- $1 \leq n \leq 2$
- $1 \leq n \leq 100$
- $0 \leq n \leq 9$
- $1000 \leq n \leq 1112$
- $-1 \leq n \leq 1$
- $-3 \leq n \leq 11$

**6.13** For each of the following sets of integers, write a single statement that will print a number at random from the set:

- 2, 4, 6, 8, 10.
- 3, 5, 7, 9, 11.
- 6, 10, 14, 18, 22.

**6.14** Write a method **integerPower( base, exponent )** that returns the value of

*base*<sup>*exponent*</sup>

For example, **integerPower( 3, 4 )** calculates  $3^4$  (or  $3 * 3 * 3 * 3$ ). Assume that **exponent** is a positive, nonzero integer and that **base** is an integer. Method **integerPower** should use **for** or **while** to control the calculation. Do not use any math library methods. Incorporate this method into an applet that reads integer values from **JTextField**s for **base** and **exponent** from the user and performs the calculation with the **integerPower** method. [Note: Register for event handling on only the second **JTextField**. The user should interact with the program by typing numbers in both **JTextField**s and pressing *Enter* only in the second **JTextField**.]

**6.15** Define a method **hypotenuse** that calculates the length of the hypotenuse of a right triangle when the other two sides are given (sample data appear in Fig. 6.22). The method should take two arguments of type **double** and return the hypotenuse as a **double**. Incorporate this method into an applet that reads values for **side1** and **side2** from **JTextField**s and performs the calculation with the **hypotenuse** method. Determine the length of the hypotenuse for each of the following triangles. [Note: Register for event handling on only the second **JTextField**. The user should interact with the program by typing numbers in both **JTextField**s and pressing *Enter* only in the second **JTextField**.]

| Triangle | Side 1 | Side 2 |
|----------|--------|--------|
| 1        | 3.0    | 4.0    |
| 2        | 5.0    | 12.0   |
| 3        | 8.0    | 15.0   |

**Fig. 6.22** Values for the sides of triangles in Exercise 6.15.

**6.16** Write a method **multiple** that determines for a pair of integers whether the second integer is a multiple of the first. The method should take two integer arguments and return **true** if the second is a multiple of the first and **false** otherwise. Incorporate this method into an applet that inputs a series of pairs of integers (one pair at a time using **JTextField**s). [Note: Register for event handling on only the second **JTextField**. The user should interact with the program by typing numbers in both **JTextField**s and pressing *Enter* only in the second **JTextField**.]

**6.17** Write a method **isEven** that uses the modulus operator to determine if an integer is even. The method should take an integer argument and return **true** if the integer is even and **false** otherwise. Incorporate this method into an applet that inputs a series integers (one at a time using a **JTextField**).

**6.18** Write a method **squareOfAsterisks** that displays a solid square of asterisks whose side is specified in integer parameter **side**. For example, if **side** is **4**, the method displays

```



```

Incorporate this method into an applet that reads an integer value for **side** from the user at the keyboard and performs the drawing with the **squareOfAsterisks** method. Note that this method should be called from the applet's **paint** method and should be passed the **Graphics** object from **paint**.

**6.19** Modify the method created in Exercise 6.18 to form the square out of whatever character is contained in character parameter **fillCharacter**. Thus, if **side** is **5** and **fillCharacter** is **"#"**, the method should print

```
#####
#####
#####
#####
#####
```

**6.20** Use techniques similar to those developed in Exercise 6.18 and Exercise 6.19 to produce a program that graphs a wide range of shapes.

**6.21** Modify the program of Exercise 6.18 to draw a solid square with the **fillRect** method of the **Graphics** class. Method **fillRect** receives four arguments: *x*-coordinate, *y*-coordinate, width and height. Allow the user to input the coordinates at which the square should appear.

**6.22** Write program segments that accomplish each of the following tasks:

- Calculate the integer part of the quotient when integer **a** is divided by integer **b**.
- Calculate the integer remainder when integer **a** is divided by integer **b**.
- Use the program pieces developed in parts a) and b) to write a method **displayDigits** that receives an integer between **1** and **99999** and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer **4562** should be printed as

```
4 5 6 2
```

- Incorporate the method developed in part c) into an applet that inputs an integer from an input dialog and invokes **displayDigits** by passing the method the integer entered. Display the results in a message dialog.

**6.23** Implement the following integer methods:

- a) Method **celsius** returns the Celsius equivalent of a Fahrenheit temperature, using the calculation

$$C = 5.0 / 9.0 * ( F - 32 );$$

- b) Method **fahrenheit** returns the Fahrenheit equivalent of a Celsius temperature, using the calculation

$$F = 9.0 / 5.0 * C + 32;$$

- c) Use these methods to write an applet that enables the user to enter either a Fahrenheit temperature and display the Celsius equivalent or enter a Celsius temperature and display the Fahrenheit equivalent.

[Note: This applet will require two **JTextField** objects that have registered action events. When **actionPerformed** is invoked, the **ActionEvent** parameter has method **getSource()** to determine the GUI component with which the user interacted. Your **actionPerformed** method should contain an **if/else** structure of the form

```
if (actionEvent.getSource() == input1) {
 // process input1 interaction here
}
else { // e.getSource() == input2
 // process input2 interaction here
}
```

where **input1** and **input2** are **JTextField** references.]

**6.24** Write a method **minimum3** that returns the smallest of three floating-point numbers. Use the **Math.min** method to implement **minimum3**. Incorporate the method into an applet that reads three values from the user and determines the smallest value. Display the result in the status bar.

**6.25** An integer number is said to be a *perfect number* if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number, because  $6 = 1 + 2 + 3$ . Write a method **perfect** that determines if parameter **number** is a perfect number. Use this method in an applet that determines and displays all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the computing power of your computer by testing numbers much larger than 1000. Display the results in a **JTextArea** that has scrolling functionality.

**6.26** An integer is said to be *prime* if it is divisible only by 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

- Write a method that determines if a number is prime.
- Use this method in an applet that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Display the results in a **JTextArea** that has scrolling functionality.
- Initially, you might think that  $n/2$  is the upper limit for which you must test to see if a number is prime, but you need only go as high as the square root of  $n$ . Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

**6.27** Write a method that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the method should return 1367. Incorporate the method into an applet that reads a value from the user. Display the result of the method in the status bar.

**6.28** The *greatest common divisor (GCD)* of two integers is the largest integer that evenly divides each of the two numbers. Write a method **gcd** that returns the greatest common divisor of two integers. Incorporate the method into an applet that reads two values from the user. Display the result of the method in the status bar.

**6.29** Write a method **qualityPoints** that inputs a student's average and returns **4** if a student's average is 90–100, **3** if the average is 80–89, **2** if the average is 70–79, **1** if the average is 60–69 and **0** if the average is lower than 60. Incorporate the method into an applet that reads a value from the user. Display the result of the method in the status bar.

**6.30** Write an applet that simulates coin tossing. Let the program toss a coin each time the user presses the **"Toss"** button. Count the number of times each side of the coin appears. Display the results. The program should call a separate method **flip** that takes no arguments and returns **false** for tails and **true** for heads. [Note: If the program realistically simulates coin tossing, each side of the coin should appear approximately half the time.]

**6.31** Computers are playing an increasing role in education. Write a program that will help an elementary school student learn multiplication. Use **Math.random** to produce two positive one-digit integers. The program should then display a question in the status bar, such as

**How much is 6 times 7?**

The student then types the answer into a **JTextField**. Next, the program checks the student's answer. If it is correct, draw the string **"Very good!"** on the applet and ask another multiplication question. If the answer is wrong, draw the string **"No. Please try again."** on the applet and let the student try the same question again repeatedly until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the applet begins execution and each time the user answers the question correctly. All drawing on the applet should be performed by the **paint** method.

**6.32** The use of computers in education is referred to as *computer-assisted instruction (CAI)*. One problem that develops in CAI environments is student fatigue. This problem can be eliminated by varying the computer's dialogue to hold the student's attention. Modify the program of Exercise 6.31 so the various comments are printed for each correct answer and each incorrect answer as follows:

Responses to a correct answer

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Responses to an incorrect answer

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random-number generation to choose a number from 1 to 4 that will be used to select an appropriate response to each answer. Use a **switch** structure in the **paint** method to issue the responses.

**6.33** More sophisticated computer-aided instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 6.32 to count the number of correct and incorrect

responses typed by the student. After the student types 10 answers, your program should calculate the percentage of correct responses. If the percentage is lower than 75%, print **Please ask your instructor for extra help** and reset the program so another student can try the program.

**6.34** Write an applet that plays the “guess the number” game as follows: Your program chooses the number to be guessed by selecting a random integer in the range 1 to 1000. The applet displays the prompt **Guess a number between 1 and 1000** next to a **TextField**. The player types a first guess into the **TextField** and presses the *Enter* key. If the player's guess is incorrect, your program should display **Too high. Try again.** or **Too low. Try again.** in the status bar to help the player “zero in” on the correct answer and should clear the **TextField** so the user can enter the next guess. When the user enters the correct answer, display **Congratulations. You guessed the number!** in the status bar and clear the **TextField** so the user can play again. [Note: The guessing technique employed in this problem is similar to a *binary search*.]

**6.35** Modify the program of Exercise 6.34 to count the number of guesses the player makes. If the number is 10 or fewer, print **Either you know the secret or you got lucky!** If the player guesses the number in 10 tries, print **Aha! You know the secret!** If the player makes more than 10 guesses, print **You should be able to do better!** Why should it take no more than 10 guesses? Well, with each “good guess” the player should be able to eliminate half of the numbers. Now show why any number from 1 to 1000 can be guessed in 10 or fewer tries.

**6.36** Write a recursive method **power ( base, exponent )** that when invoked returns

$$\text{base}^{\text{exponent}}$$

For example, **power ( 3, 4 ) = 3 \* 3 \* 3 \* 3**. Assume that **exponent** is an integer greater than or equal to 1. (Hint: The recursion step should use the relationship

$$\text{base}^{\text{exponent}} = \text{base} \cdot \text{base}^{\text{exponent} - 1}$$

and the terminating condition occurs when **exponent** is equal to 1, because

$$\text{base}^1 = \text{base}$$

Incorporate this method into an applet that enables the user to enter the **base** and **exponent**.)

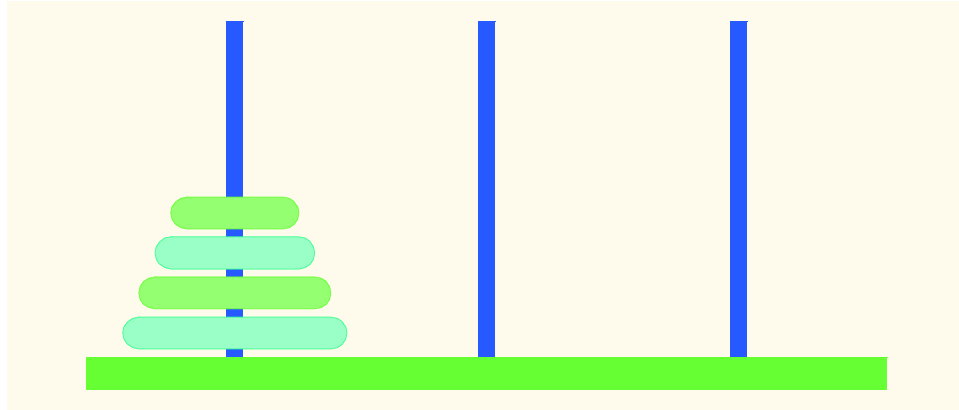
**6.37** (*Towers of Hanoi*) Every budding computer scientist must grapple with certain classic problems, and the *Towers of Hanoi* (see Fig. 6.23) is one of the most famous. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving  $n$  disks can be viewed in terms of moving only  $n - 1$  disks (and hence the recursion) as follows:

- Move  $n - 1$  disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
- Move the last disk (the largest) from peg 1 to peg 3.
- Move the  $n - 1$  disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving  $n = 1$  disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.



**Fig. 6.23** The Towers of Hanoi for the case with four disks.

Write an applet to solve the Towers of Hanoi problem. Allow the user to enter the number of disks in a **JTextField**. Use a recursive **tower** method with four parameters:

- a) the number of disks to be moved,
- b) the peg on which these disks are initially threaded,
- c) the peg to which this stack of disks is to be moved, and
- d) the peg to be used as a temporary holding area.

Your program should display in a **JTextArea** with scrolling functionality the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

```
1 → 3 (This notation means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3
```

**6.38** Any program that can be implemented recursively can be implemented iteratively, although sometimes with more difficulty and less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 6.37. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

**6.39** (*Visualizing Recursion*) It is interesting to watch recursion “in action.” Modify the factorial method of Fig. 6.12 to print its local variable and recursive-call parameter. For each recursive call, display the outputs on a separate line, and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

**6.40** The greatest common divisor of integers **x** and **y** is the largest integer that evenly divides into both **x** and **y**. Write a recursive method **gcd** that returns the greatest common divisor of **x** and **y**. The **gcd** of **x** and **y** is defined recursively as follows: If **y** is equal to 0, then **gcd(x, y)** is **x**; otherwise, **gcd(x, y)** is **gcd(y, x % y)**, where % is the modulus operator. Use this method to replace the one you wrote in the applet of Exercise 6.28.

**6.41** Exercise 6.31 through Exercise 6.33 developed a computer-assisted instruction program to teach an elementary school student multiplication. This exercise suggests enhancements to that program.

- Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, etc.
- Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of **1** means addition problems only, **2** means subtraction problems only, **3** means multiplication problems only, **4** means division problems only and **5** means to intermix randomly problems of all these types.

**6.42** Write method **distance**, to calculate the distance between two points ( $x1, y1$ ) and ( $x2, y2$ ). All numbers and return values should be of type **double**. Incorporate this method into an applet that enables the user to enter the coordinates of the points.

**6.43** What does the following method do?

```
// Parameter b must be a positive
// integer to prevent infinite recursion
public int mystery(int a, int b)
{
 if (b == 1)
 return a;
 else
 return a + mystery(a, b - 1);
}
```

**6.44** After you determine what the program in Exercise 6.43 does, modify the method to operate properly after removing the restriction of the second argument being nonnegative. Also, incorporate the method into an applet that enables the user to enter two integers, and test the method.

**6.45** Write an application that tests as many of the math-library methods in Fig. 6.2 as you can. Exercise each of the methods by having your program print out tables of return values for a diversity of argument values.

**6.46** Find the error in the following recursive method, and explain how to correct it:

```
public int sum(int n)
{
 if (n == 0)
 return 0;
 else
 return n + sum(n);
}
```

**6.47** Modify the craps program of Fig. 6.9 to allow wagering. Initialize variable **bankBalance** to 1000 dollars. Prompt the player to enter a **wager**. Check that **wager** is less than or equal to **bankBalance**, and if not, have the user reenter **wager** until a valid **wager** is entered. After a correct **wager** is entered, run one game of craps. If the player wins, increase **bankBalance** by **wager**, and print the new **bankBalance**. If the player loses, decrease **bankBalance** by **wager**, print the new **bankBalance**, check if **bankBalance** has become zero, and if so, print the message "Sorry. You busted!" As the game progresses, print various messages to create some "chatter," such as "Oh, you're going for broke, huh?" or "Aw c'mon, take a chance!" or "You're up big. Now's the time to cash in your chips!". Implement the "chatter" as a separate method that randomly chooses the string to display.

**6.48** Write an applet that uses a method **circleArea** to prompt the user for the radius of a circle and to calculate and print the area of that circle.