CHAPTER 6

Methods

"Though this be madness, yet there is method in 't."
—From Hamlet (II, ii, 206)

"There is more madness to my method than method to my madness."
—Salvador Dali

Objectives

The objectives of Chapter 6 include an understanding of

- the concept of a method as a "black box,"
- the methods of Java's Math class,
- how to construct methods that carry out simple tasks,
- the differences between void methods and methods that return a value,
- the scope of a name, and
- method overloading: advantages and potential pitfalls.

6.1 INTRODUCTION

Not too long ago, in the pioneer days of programming (that's circa 1966), mathematicians Corrado Bohm and Guiseppe Jacopini proved that *any* computer program can be written using just three basic structures:

1. sequence (statements in a program are executed sequentially),
2. selection (if-else statements), and
3. repetition (loops).

These three fundamental ideas are the principal concepts of Chapters 2 through 5. So, at least *theoretically*, you can put aside this text and implement any program that you dare to dream up! You have the tools.

Needless to say, complex computer programs are built with tools more sophisticated than three simple, albeit powerful, structures. Indeed, a carpenter could theoretically build a house using nothing more than nails, a saw, a hammer, and some lumber; but the task wouldn't be easy, and the finished product may be unsightly. As a carpenter needs more powerful equipment, the programmer requires tools beyond sequence, selection, and repetition. One such programming construct is the *method.*

A *method* is a named sequence of instructions that are grouped together to perform a task.

Complicated programs perform many different tasks. Methods enable the programmer to organize various tasks into neat, manageable, independent bundles of code. Every Java application that we have written contains one method; its name is main and its instructions appear between the opening and closing braces of main.

Every Java application must have a main method, and the execution of every Java application begins with the main method.

Other methods that we have used are print(…), println(…), and Math.random().

In this chapter you will learn about a few more prepackaged methods provided by Java as well as how to construct your own methods. We begin with a "black box" view of a method.

6.2 JAVA'S PREDEFINED METHODS

Imagine a mathematical, if not magical, "black box" that works in such a way that whenever you supply a number to the box, the box gives or *returns* the positive square root of that number. See Figure 6.1a.

FIGURE 6.1a A *square root* box

Figure 6.1b illustrates a similar mechanism that accepts two numbers, perhaps the length and width of a rectangle, and returns the area of the rectangle.

FIGURE 6.1b An *area* box

Or can you fathom a gizmo that receives a character and returns the integer (ASCII) value of that character? See Figure 6.1c.

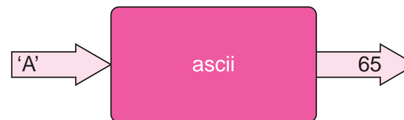


FIGURE 6.1c An *ASCII converter* box

Such a "box" is a metaphor for a *method*. A method is very much like a mathematical function—a black box that computes an output given some inputs.

The values that you supply or *pass* to the method are called *arguments*. The value computed by the method is the *returned value*.

Later, you will see that a method may perform a task without accepting arguments or returning a value.

Java comes bundled with an extraordinary number of methods. Each of these built-in methods is comprised of Java code that performs some specific task. Fortunately, the

programmer need not know *how* these Java-supplied methods work "inside the box" or "under the hood," but simply how to use them.

How do you use these methods? Where do you get them? Let's start with a simple example.

6.2.1 The Square Root Method

EXAMPLE 6.1

Imagine that you are standing on a beach gazing out at the sea. What is the distance to the horizon? How far ahead can you see? How far can you see if you are standing on a cliff above the beach?

In general, the distance to the horizon (in miles) can be estimated as follows:

- Determine the distance (in feet) from sea level to your eyes.
- Compute the square root of that distance.
- Multiply the result by 1.23.

Problem Statement Write a program that prompts a user for the distance measured from the ground to his/her eyes and calculates the distance to the horizon.

Notice that the following program must calculate a square root. This calculation is performed compliments of the method Math.sqrt(x)—a black box.

Java Solution

```
1.  import java.util.*;
2.  public class DistanceToHorizon
3.  {
4.     public static void main(String[] args)
5.     {
6.        Scanner input;
7.        double distanceToEyes;          // measured from the ground
8.        double distanceToHorizon;
9.        int answer = 1;                 // used to repeat the calculation
10.       input = new Scanner(System.in);
11.       do
12.       {
13.          System.out.print("Distance from the ground to your eyes in feet: ");
14.          distanceToEyes = input.nextDouble();
15.          distanceToHorizon = 1.23 * Math.sqrt(distanceToEyes);
16.          System.out.println("The distance to the horizon is " + distanceToHorizon + "mi.");
17.          System.out.print("Again? Enter 1 for YES; any other number to Exit: ");
18.          answer = input.nextInt();
19.       }while (answer == 1);
20.    }
21. }
```

Output

```
Distance from the ground to your eyes in feet: 16.0
The distance to the horizon is 4.92 mi.
Again? Enter 1 for YES; any other number to Exit: 1

Distance from the ground to your eyes in feet: 5.25
The distance to the horizon is 2.8182840523978414 mi
Again? Enter 1 for YES; any other number to Exit: 0
```

Discussion On line 15, the program utilizes the method

double Math.sqrt(double x)

to calculate the square root of distanceToEyes. The method Math.sqrt(…) hides the details of its implementation. *How* the square root of a number is calculated is hidden

from the programmer. The method functions as a black box, and the programmer simply *uses* this method in the program.

The *argument* passed to the method is distanceToEyes (a double), and the returned value (a double) is the square root of distanceToEyes.

For example, if distanceToEyes has the value 16.0, then Math.sqrt(distanceToEyes) returns the value 4.0 and that value is used in the expression

```
distanceToHorizon = 1.23 * Math.sqrt(distanceToEyes);
```

That's all there is to it.

The program of Example 6.1 utilizes the Math.Sqrt(…) method. To understand how a Java method works, let's take a closer look at the mechanics of this particular method.

Consider the statement

```
double root = Math.sqrt(25.0);
```

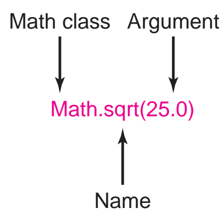


FIGURE 6.2 The *sqrt* method of the *Math* class

The effect of this statement is that variable root is assigned the value 5.0, the square root of 25.0.

This method, which calculates square root, is a member of Java's Math class. The Math class is a Java-supplied collection (or library) of methods that performs mathematical tasks or functions. Math.sqrt(…) is one of several methods in the Math class. The name of the method is sqrt, and the argument that is supplied to the method is the number 25.0. Notice the period that separates the class name Math from the method name, sqrt. See Figure 6.2.

In the statement

```
double root = Math.sqrt(25.0)
```

the Math.sqrt(…) method is *called* (or *invoked*) with the argument 25.0 and *returns* the value 5.0 (the square root of 25.0), which is subsequently assigned to the variable root. This action is similar to that of the statement:

```
double sum = 5.0 + 8.0;
```

Here, the expression 5.0 + 8.0 evaluates to (or returns) 13.0, which is assigned to sum.

The argument that is passed to a method may be a constant, an expression, or a variable. And a method call may be used within an expression. The following are valid method calls:

```
System.out.println(Math.sqrt(456));    // prints the square root of 245 ( double)

double w = Math.sqrt(input.nextInt()); // here input is a Scanner object

double x = input.nextDouble();

double y = input.nextDouble();

double z = 3.14 * Math.sqrt(x + y);    // method is used within an expression
```

A method is described by its *header,* which has the following form:

return-type **name(*parameter-list*)**

- The *return-type* specifies the data type of the value returned by the method.
- The *parameter-list* enumerates the number (implicitly) and type (explicitly) of the arguments that must be passed or given to the method.
- The names in the parameter-list are called *formal parameters,* or simply *parameters*.

For example, the header of Figure 6.3 tells us that the method named Math.sqrt accepts one argument of type double and returns a double. Parameter x is a (formal) parameter.

Return-type   Parameter-list

double Math.sqrt(double x)

FIGURE 6.3 The header for *Math.sqrt(…)*

Although the header specifies that the argument passed to the Math.sqrt(…) be of type double, an argument of any data type may be used, provided that the argument can be automatically cast to type double. Thus, the argument of

Math.sqrt(25)

is first cast to the double 25.0. The returned value is 5.0 (not 5). The returned value is always type double regardless of the argument. To obtain an integer, you can perform an explicit cast on the method's return value:

```
(int)Math.sqrt(25);
```

Figure 6.4 lists some useful methods found in the Math class. In each case, the first two columns comprise the header for each method.

Return Type	Method	Description	Example
double	abs(double x)	absolute value	Math.abs(−3.1) returns 3.1
int	abs(int a)	absolute value	Math.abs(−25) returns 25
double	ceil(double x)	returns the smallest whole number (as a double) greater than or equal to x	Math.ceil(3.14159) returns 4.0
double	cos(double x)	cosine function, x is in radians	Math.cos(3.141592653589793) returns −1.0 ($\cos(\pi) = -1$)
double	exp(double x)	the exponential function, e^x	Math.exp(0.0) returns 1.0 ($e^0 = 1$)
double	floor(double x)	returns the largest whole number (as a double) less than or equal to x	Math.floor(3.14159) returns 3.0
double	log(double x)	natural logarithm, $ln(x)$	Math.log(1.0) returns 0.0 ($\ln(1) = 0$)
double	max(double x, double y)	returns the greater of x and y	Math.max(3.0,4.0) returns 4.0
int	max(int a, int b)	returns the greater of x and y	Math.max(3,4) returns 4 (int)
double	min(double x, double y)	returns the lesser of x and y	Math.min(3.0,4.0) returns 3.0
int	min(int a, int b)	returns the lesser of a and b	Math.min(3,4) returns 3 (int)
double	pow(double x, double y)	x^y	Math.pow(2.0,5.0) returns 32.0
double	random()	returns a random number x such that $0.0 <= x < 1$	Math.random() may return 0.2345676889 or perhaps 0.654678756
long	round(x double)	rounds to the nearest whole number (long)	Math.round(3.14) returns 3 (long) Math.round (5.67) returns 6 (long)
double	sin(double x)	sine function, x is in radians	Math.sin(3.141592653589793) returns 0.0 ($\sin(\pi) = 0$)
double	sqrt(double x)	square root	Math.sqrt(144.0) returns 12.0
double	tan(double x)	tangent function, x is in radians	Math.tan(3.141592653589793) returns 0.0 ($\tan(\pi) = 0$)

FIGURE 6.4 Methods of the *Math* class

6.2.2 A Method that Computes Powers

The next example uses the method

```
double Math.pow(double x, double y)
```

to calculate x^y.

Notice that the parameter list of the header specifies that the method requires two arguments of type double. For example, Math.pow(5.0,2.0) returns $5.0^{2.0}$, that is, 25.0. See Figure 6.5.

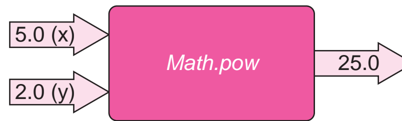


FIGURE 6.5 The power method, *Math.pow(...)*

EXAMPLE 6.2 Legend tells us that approximately 380 years ago Peter Minuit purchased the island of Manhattan for the grand sum of 60 Dutch guilders (approximately $24). If Mr. Minuit had instead deposited his $24 in the local bank at 5% interest, compounded daily, what would his money be worth today? Was his real estate investment a wise one?

To calculate the present value of Peter Minuit's original $24, we use the interest formula:

$$value = amount(1 + rate/360)^{360*years}$$

where *value* represents the present value, *amount* is the initial investment, *rate* is the yearly interest rate, and *years* is the time (in years) of the investment. Thus, for the problem at hand, *value* is calculated as

$$value = 24(1 + .05/360)^{360*380}$$

Here, we use 360 days (a 30-day month) for a "bank year," rather than 365.

Problem Statement Write a program that prompts the user for:

- the initial investment,
- the interest rate, and
- the term in years,

and calculates the present value. To perform the calculation, we use Java's "power method," Math.pow(x,y), which calculates x^y.

Java Solution

```
1.  import java.util.*;
2.  public class Interest
3.  {
4.      public static void main(String[] args)
5.      {
6.          Scanner input;
7.          double value;
8.          double amount;
9.          double rate;
10.         double years;
11.         final int DAYS = 360;              // one year
12.         // prompt for initial investment
13.         input = new Scanner(System.in);
```

```
14.        System.out.print("Initial amount: ");
15.        amount = input.nextDouble();
16.        // prompt for yearly interest rate
17.        System.out.print("Interest rate: ");
18.        rate = input.nextDouble();
19.        // prompt for number of years
20.        System.out.print("Time in years: ");
21.        years = input.nextDouble();
22.        // value = amount * (1 + rate / DAYS)^(DAYS*years) – standard interest formula
23.        value = amount * Math.pow (1 + rate / DAYS, DAYS * years); // (1 + rate / DAYS)^DAYS*years
24.        System.out.println("Present value $" + value);
25.    }
26. }
```

Output (Using the Minuit Data)

```
Initial amount: 24.00
Interest rate: .05
Time in years: 380
Present value $4.2779275332526875E9
```

Discussion The method Math.pow(...) is invoked on line 23 with two arguments, both expressions. Notice that the present value is displayed in scientific notation. In decimal notation, that's about $4,277,927,533. Considering the value of real estate in Manhattan, it appears that Peter made a very wise investment.

6.2.3 Random Numbers

The

```
double Math.random()
```

method returns a random number that is greater than or equal to 0.0 and strictly less than 1.0. Notice that Math.random() requires no parameter or argument.

For example, the first time that a program invokes Math.random(), the returned value might be 0.8787954399107227, and the next time it might be 0.31799656386438013. Each subsequent number returned by Math.random() is supposedly unpredictable. The following small program calls Math.random() ten times. There is no discernible pattern to the output . . . it's random.

```
1.  public class TenRandomNumbers
2.  {
3.      public static void main(String[] args)
4.      {
5.          for (int i = 1; i <= 10; i++)
6.              System.out.println(Math.random());
7.      }
8.  }
```

Output

```
0.6516831128923004
0.3159760705754926
0.945877632966408
0.04538322890407964
0.8815999823052094
```

```
0.07672479266883347
0.04423548066038108
0.4441137107417066
0.15348060768674676
0.1833850393131755
```

Random numbers are indispensable for performing simulations. Such simulations are useful in all kinds of applications, including earthquake modeling, epidemic predictions, rocket testing, and games. For example, a card game that uses a deck of 52 cards might associate each card with a number from 1 to 52. Dealing a card amounts to nothing more than choosing a random number in that range. Or, a program might use a random integer, either 0 or 1, to simulate the toss of a coin: 0 for heads and 1 for tails.

Using Math.random() to Generate Integers

With a little hocus pocus we can use Math.random() in all sorts of situations. For example, to simulate the roll of a single die, a program requires a random integer between 1 and 6 inclusive. We can use Math.random() to generate integers in the range 1 through 6 by "magnifying" its 0 through 1 range.

If

```
r = Math.random();
```

then r is of type double and

$0.0 \le r < 1.0.$

Therefore,

$0.0 \le 6 * r < 6.0$ (multiplying the inequality by 6), and
$1.0 \le 6 * r + 1 < 7.0.$ (adding 1 to each value in the inequality)

Thus 6* Math.random() + 1 is a number greater than or equal to 1 but strictly less than 7.

For example, if

$$r = 0.8929343993861253, \text{ then}$$
$$6 * r = 5.3576063963167518, \text{ and}$$
$$6 * r + 1 = 6.3576063963167518.$$

To obtain an integer value, cast $6 * r + 1$ to an integer, effectively dropping the fractional part. Thus,

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

returns a random integer between 1 and 6, inclusive. Similarly, (int)(52 * Math.random() +1) returns a random integer between 1 and 52, inclusive. You can use this trick to generate random integers in any range. For example, (int)(10 * Math.random() + 15) returns an integer between 15 and 24, inclusive.

Example 6.3 uses Math.random() to simulate a simple casino dice game.

EXAMPLE 6.3 Probably the simplest of all casino bets is the "over-under" bet. Two dice are rolled, and a player has the option of betting whether the sum of the spots displayed on the dice will be:

1. over 7,
2. under 7, or
3. exactly 7.

Bets (1) and (2) pay even money. So if a player bets $1, a win pays his money back plus $1. Bet (3) pays 4 to 1. Thus if a player bets $2 on 7, a win pays him back his $2 plus $8.

Problem Statement Write a program that simulates the over-under game. If the player wins, the winning amount (not including the returned original bet) is reported, and if the player loses, a message is printed.

Java Solution

```
1.  import java.util.*;
2.  public class Dice
3.  {
4.     public static void main(String [] args)
5.     {
6.        Scanner input;
7.        int bet;
8.        int wager;
9.        int die1,die2;
10.       int sum;
11.       input = new Scanner(System.in);

12.       // Place your bet
13.       System.out.print("Enter your bet\n (1) Over 7 \n (2) Under 7 \n (3) Exactly 7\n: ");
14.       bet = input.nextInt();
15.       System.out.print("Enter your wager (whole number): ");
16.       wager = input.nextInt();

17.       // Roll the dice
18.       die1 = (int)(6 * Math.random() + 1) ; // random integer 1..6
19.       die2 = (int)(6 * Math.random() + 1);
20.       sum = die1 + die2;
21.       System.out.println("The sum of the dice is " + sum);

22.       // Check for a win
23.       if ((sum > 7) && (bet == 1) || (sum < 7) && (bet == 2))
24.          System.out.println("You win $" + wager);
25.       else if ((sum == 7) && (bet == 3))
26.          System.out.println("You win $" + (4 * wager));
27.       else
28.          System.out.println("You lose!");
29.    }
30. }
```

Output (Two Games)

```
Enter your bet
  (1) Over 7
  (2) Under 7
  (3) Exactly 7:
2
Enter your wager (whole number): 3
The sum of the dice is 8
You lose!

Enter your bet
  (1) Over 7
  (2) Under 7
  (3) Exactly 7:
```

```
1
Enter your wager (whole number): 6
The sum of the dice is 9
You win $6
```

Discussion The expressions on lines 18 and 19 simulate the roll of a single die. As explained above, even though Math.random() returns a *floating*-point number that is greater than or equal to 0 and strictly less than 1, this Java method can be used to generate random *integers*.

6.3 WRITING YOUR OWN METHODS

Although there are thousands of methods in Java's extensive libraries, Java certainly cannot provide a method for every imaginable task. Fortunately, you can create your own methods that do whatever task you fancy—be it a method to calculate your taxes or one to determine your weight on the moon. Like Java's methods, a method that you create:

- has a name,
- may accept arguments,
- may return a value, and
- may be used as part of an expression.

The difference between a Java method and one of your own creation is that with your own method *you* must program the "black box." You are the designer, the architect and the builder. (Well, you can't expect Java to do *everything*.) In the following examples, we illustrate two types of Java methods: those that return a value and those that do not.

6.3.1 Methods that Return a Value

Many of the prepackaged methods that we have encountered perform a computation and return the result of the computation to the caller. For example, Math.sqrt(double x) returns the square root of x, and Math.random() returns a random number. The following application includes a method that returns a value but, unlike Math.sqrt(…) or Math.random(), this method is *not* part of Java's library.

EXAMPLE 6.4 Rapid Rick runs races regularly. Although Rick is determined to keep in shape, he does enjoy an occasional slice of cheesecake. If Rick knows approximately how many calories he burns while running, well, he just might treat himself to a little more dessert with a little less guilt.

The number of calories used while running depends on the runner's weight as well as the distance that he/she has run. A common rule of thumb used to estimate the number of calories burned is:

$$calories = .653 \times weight \times distance$$

where *weight* is the runner's weight in pounds and *distance* is in miles.

Problem Statement Write a program that calculates the number of calories burned as a function of weight and distance. Include a method

double caloriesBurned(double weight, double distance)

that accepts two arguments of type double and returns a value of type double. See Figure 6.6.

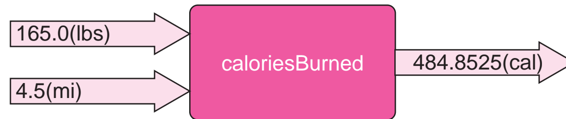


FIGURE 6.6 The method *double caloriesBurned(double weight, double distance)*

Java Solution

```
1.  import java.util.*;
2.  public class RunnersCalculator
3.  {

4.     public static double caloriesBurned(double weight, double distance)
5.     {

6.       // returns the number of calories burned using the formula
7.       // calories = .653 × weight × distance
8.         double calories = .653 * weight * distance;
9.         return calories;
10.    }

11.    public static void main(String[] args)
12.    {
13.       Scanner input;
14.       double myWeight, myDistance, totalCalories;

15.       input = new Scanner(System.in);
16.       System.out.print("Enter weight in pounds: ");
17.       myWeight = input.nextDouble();
18.       System.out.print("Enter distance in miles: ");
19.       myDistance = input.nextDouble();

20.       totalCalories = caloriesBurned(myWeight, myDistance);
21.       System.out.println("Calories burned: " + totalCalories);
22.    }
23. }
```

Output

Enter weight in pounds: **165.0**
Enter distance in miles: **6.0**
Calories burned: 646.47

Discussion Like all Java applications, RunnersCalculator begins execution with main(…) (lines 11–22). The main(…) method is similar to the main(…) method of any other program that we've written. You should notice, however, that within main(…) there is a call to the method caloriesBurned(…) on line 20:

totalCalories = **caloriesBurned(myWeight, myDistance);**

A call to caloriesBurned(…) is really no different than the call to Math.sqrt(…) in Example 6.1 or the call to Math.random() in Example 6.3. The method call to calories-Burned(…) has two arguments: myWeight and myDistance; the returned value is assigned to the variable totalCalories.

The instructions of the method caloriesBurned(…) are specified on lines 8 and 9. Unlike Math.sqrt(…) or Math.random(), we can now look "inside the box," so to speak. So let's do just that.

Line 4 is the header of the method:

```
public static double caloriesBurned(double weight, double distance)
```

For now, you can ignore the keywords public and static. They are necessary and soon they will make more sense to you. The remainder of the header specifies:

- the data type of the return value: double,
- the name of the method: caloriesBurned, and
- the parameters: weight and distance.

The parameters specify the type and number of the arguments that must be passed to the method. When this method is invoked with two arguments, the value of the first argument is assigned or passed to weight and the value of the second argument is passed to parameter distance. For example, if the method call is

```
caloriesBurned(155.5, 3.5)
```

the parameter weight gets the value 155.5, and distance the value 3.5. See Figure 6.7.

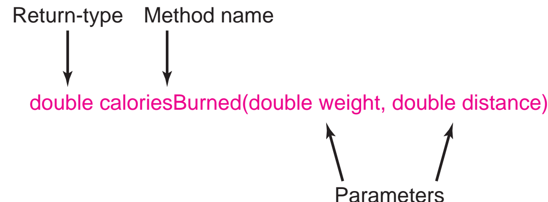


FIGURE 6.7 Parts of a method header

The block consisting of lines 5 through 10 contains the instructions of the method caloriesBurned(…).

- Line 8 is an expression that calculates the number of calories burned.
- Line 9 is a return statement. The return statement has the form:

return *expression*

> The return statement has two purposes:
> 1. It specifies the value that the method returns to the caller.
> 2. It terminates the method and returns program control to the caller.

That's all there is to it.

Figure 6.8 steps through the execution of the program. As you can see, the program executes main(…) sequentially, with a side trip to caloriesBurned(…) on line 20.

myWeight	myDistance	totalCalories	Explanation
			Line 14: Declare three variables, myWeight, myDistance, and totalCalories.
165.0			Line 17: Obtain a value for myWeight.
165.0	6.0		Line 19: Obtain a value for myDistance.
165.0	6.0		Line 20: Call caloriesBurned(…). Pass values of the arguments myWeight and myDistance to parameters weight and distance, respectively.

Program control passes to caloriesBurned(…).

weight	distance	calories	Explanation
165.0	6.0		Line 4: The parameters weight and distance are initialized with the values of arguments myWeight and myDistance.
165.0	6.0	646.47	Line 8: Declare the variable calories. Calculate the number of calories burned, and initialize calories to that value.
165.0	6.0	646.47	Line 9: Return the value of calories to the caller and exit.

Program control returns to the assignment on line 20.

myWeight	myDistance	totalCalories	Explanation
165.0	6.0	646.47	Line 20 (resumed): Assign the returned value to totalCalories.
165.0	6.0	646.47	Line 21: Print the results.

FIGURE 6.8 A trace of *RunnersCalculator*

For the correct values to be passed to the appropriate parameters, the order of the arguments is crucial. When caloriesBurned(…) is invoked, the values stored in the two arguments, myWeight and myDistance, are assigned, or passed, to the parameters specified in the header of caloriesBurned(…): weight and distance, respectively. See Figure 6.9.

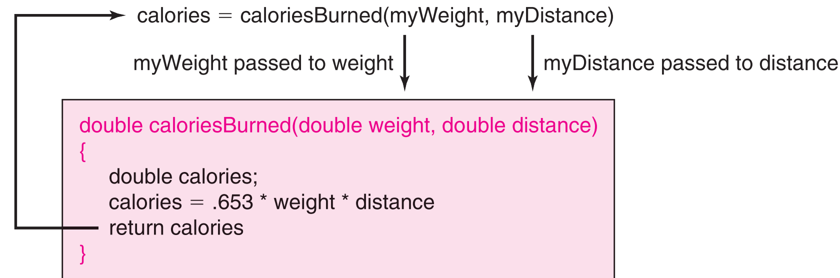


FIGURE 6.9 Arguments are passed to parameters: *weight* gets the value of *myWeight*, and *distance* the value of *myDistance*

The values of myWeight and myDistance that are passed to caloriesBurned(…) are the values used in the expression

```
.635 * weight * distance
```

on line 8.

The arguments myWeight and myDistance supply values to the parameters weight and distance. The arguments initialize the parameters. The parameters weight and distance are considered variables of the method. Once the arguments, myWeight and myDistance, pass their values to weight and distance, the role of the arguments is complete. Variables myWeight and myDistance have no further jobs in caloriesBurned(…). Indeed, if caloriesBurned(…) were to alter weight or distance, the change would not affect myWeight or myDistance. Except for the initial copying of argument values to parameters, there is no link between the parameters and the arguments.

When the *value* of an argument is copied to a parameter, the argument is said to be *passed by value.*

6.3.2 *void* Methods

A method can perform a task without returning a value. Such a method is called a void method. You have already seen two void methods: print(…) and println(…). Each method displays text but neither returns a value.

To specify a void method, use the reserved word void in place of the return type in the method header.

For example,

```
void drawSquare(int size)
```

might be the header of a method that draws a square on the screen and does not return a value. Because a void method does not return a value, it makes no sense to incorporate a void method into an expression. The expression

```
5 * Math.sqrt(25)
```

is certainly meaningful and has the value 25.0, but

```
5 * drawSquare(25)
```

makes no sense because drawSquare(25) does not return a value.

A call to a void method is a "standalone" statement consisting of the method name along with any arguments that must be passed to the method, such as

```
System.out.println("Print me!");
```

or

```
drawSquare(10);
```

In Example 6.5, coinChanger(…) is a void method: coinChanger(…) performs a task but does not return a value.

EXAMPLE 6.5

Problem Statement Write a program that includes a void method

```
void coinChanger(int amount)
```

that accepts a single integer argument between 1 and 100 that represents an amount of money between $.01 and $1.00. The method makes change for that amount using the minimum number of coins. Coins are in denominations of half dollars, quarters, dimes, nickels, and pennies.

To ensure that the smallest number of coins is used, first compute the maximum number of half dollars, followed by the maximum number of quarters, and so on. For example, if the initial amount is 83 cents, we first calculate, in order, the number of half dollars, quarters, dimes, nickels, and pennies:

- from 83 cents: 1 half dollar, 33 cents remain;
- from 33 cents: 1 quarter, 8 cents remain;
- from 8 cents: 0 dimes, 8 cents remain;
- from 8 cents: 1 nickel, 3 cents remain;
- finally, 3 pennies remain.

These calculations are accomplished using the / (integer divide) and % (remainder) operators.

Java Solution

```
1.  import java.util.*;
2.  public class MoneyChanger
3.  {
4.      public static void coinChanger (int amount)
5.      {
6.          // calculates the minimum number of half dollars, quarters, dimes, nickels
7.          // and pennies in amount

8.          int halfDollars, quarters, dimes, nickels, pennies;

9.          System.out.println();
10.         System.out.println(amount + " cents can be converted to:");

11.         halfDollars = amount / 50;      // determine number of half dollars
12.         amount = amount % 50;           // how much remains?
13.         quarters = amount / 25;         // determine number of quarters
```

```
14.         amount = amount % 25;          // how much remains?
15.         dimes = amount / 10;           // determine the number of dimes
16.         amount = amount % 10;          // how much remains?
17.         nickels = amount / 5;          // determine the number of nickels
18.         pennies = amount % 5;          // remainder is the number of pennies
19.         System.out.println("Half Dollars: " + halfDollars);
20.         System.out.println("Quarters : " + quarters);
21.         System.out.println("Dimes : " + dimes);
22.         System.out.println("Nickels : " + nickels);
23.         System.out.println("Pennies : " + pennies);
24.         return;                        // return statement is optional here
25.     }

26.     public static void main(String[] args)
27.     {
28.         Scanner input;
29.         input = new Scanner(System.in);
30.         System.out.print("Enter a value between 1 and 100: ");
31.         int money = input.nextInt();
32.         coinChanger(money);            // call to method coinChanger
33.     }
34. }
```

Output

```
Enter a value between 1 and 100: 83

83 cents can be converted to:
Half Dollars: 1
Quarters    : 1
Dimes       : 0
Nickels     : 1
Pennies     : 3
```

Discussion The program prompts the user for an initial amount of money and invokes the method coinChanger(…) with that value as an argument. Because coinChanger(…) does not return a value, the call to coinChanger(…) is not called within an expression. The method call is the Java statement (line 32):

```
coinChanger (money);
```

The parameter amount of coinChanger(…) accepts the value of the argument money, which is supplied interactively. Next, the number of half dollars is calculated, as well as how much remains after the half dollars have been removed from amount (lines 11 and 12). Likewise, the numbers of quarters, dimes, and nickels are determined. After calculating the number of nickels, the final remainder represents the number of pennies (line 18).

Take note of the return statement on line 24. Unlike the method of Example 6.4, this return statement does not include a return value or an expression. In this situation, the return statement merely causes the method to exit; no value is returned to the calling method.

> Execution of a return statement in a void method causes the method to exit without returning a value to the caller.

Indeed, the return statement on line 24 is unnecessary. After a void method executes its last statement, the method automatically returns; no final return statement is necessary. In contrast to a method that returns a value, a void method is not required to have *any* return statements.

6.3.3 Putting It All Together

Let's take a more general look at the components of a method and fill in a few details.

A Java method consists of a
- *header* followed by a
- *method block*.

The *parameters* in the header specify the number and type of the *arguments* that must be *passed* to the method. When a method is invoked, the values stored in the arguments are copied to the parameters.

In Example 6.4, weight and distance are parameters, and myWeight and myDistance are arguments. In Example 6.5, the parameter is amount and the argument is money. The parameters are sometimes called *formal parameters* and the arguments *actual parameters*.

The form of the *header* is:

***modifiers return-type name*(*parameter-list*)**

where:

- *modifiers* (for now) are the keywords public and static;
- *return-type* is the data type of the value that the method returns, or void if the method does not return a value;
- *parameter-list* is a (possibly empty) list of parameters that receive values from arguments passed to the method when the method is invoked.

The *method block* is a sequence of statements enclosed by curly braces:

```
{
        statement-1;
        statement-2;
        statement-3;
            . . .
        statement-n;
}
```

For example, Figure 6.10 shows a method that calculates the volume of a box.

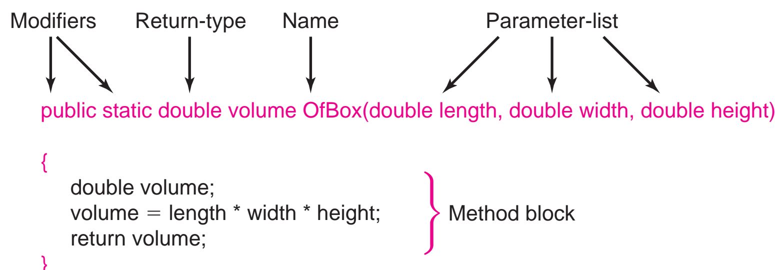


FIGURE 6.10 A method that calculates the volume of a box

That's the big picture, but a few details are in order:

1. **Method Name.** The name of a method must be a valid Java identifier. Moreover, a method name should convey the method's purpose, function, or task. For example, the name volumeOfBox is more suitable than the name myMethod or box. Standard Java convention specifies that the name of a method begins with a lowercase letter and

starts each succeeding word in the method name with an uppercase letter. For example, the names volumeOfBox and caloriesBurned both follow this convention; the names VolumeOfBox and volumeofbox do not.

2. **Parameter-List.** A method's *parameter-list* consists of pairs of the form:

   type parameter-name

separated by commas. Figure 6.11 shows the parameter-list of the method volumeOfBox.

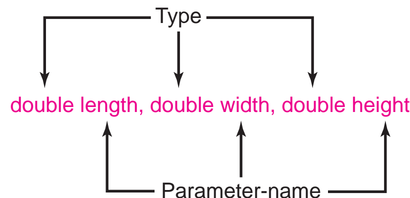


FIGURE 6.11 A parameter-list

For example:

- The parameter-list of method caloriesBurned in Example 6.4 is:

  double weight, double distance

- The method Math.random() has no parameter-list. Math.random() neither requires nor accepts any arguments. The parameter-list is empty.

3. **Argument Passing.** When calling a method, the caller passes arguments to the parameters. The calling statement must provide a type-suitable value for each parameter. If a method has five parameters, five arguments are required. Supplying more or fewer arguments than parameters is an error that the compiler can detect.

- For example, the method

  ```
  double volumeOfBox(double length, double width, double height)
  ```

  has three parameters each of type double. The following are valid calls to volumeOfBox(…):

  ```
  volumeOfBox(2.34, 5.765, 4.678)   // three doubles are passed
  volume of box(l, w, h)            // l, w, and h are type double
  volumeOfBox(3, 4, 5)              // an integer can be expanded to a double
  volumeOfBox(3.0*l, 1.5*w, 2.7*h); // expressions are OK
  ```

  In contrast, if

  ```
  int volumeOfBox(int length, int width, int height)
  ```

  is a method with integer parameters, then the call

  ```
  volumeOfBox(3.0, 4.0, 5.0)
  ```

  is unacceptable because a value of type double cannot be automatically cast to an integer.

- Finally, note that the invocation

  ```
  volumeOfBox(2.3, 4.5) // INVALID.
                        // Wrong number of arguments
  ```

  is illegal: only two values are passed and volumeOfBox(…) requires three.

4. **Pass by Value.** All arguments are passed "by value." This means that the arguments are evaluated and *values* of the arguments are copied to the parameters of a method. Subsequently, modifying the parameters in the method has no effect on the value of any variables passed as arguments.
5. **Method Block.** The statements of the *method block* accomplish the task of the method.
6. **The return Statement.** A method that returns a value *must* include a return statement. The form of the return statement is

   return *expression*

   If the data type of the returned value (as specified in the method header) is T, then the data type of *expression* should also be type T (or a type that is automatically cast to T). For example, the following method header specifies that the return type of method gimmeFive is double.

   double gimmeFive()

   The methods

   ```
   double gimmeFive()      and  double gimmeFive()
   {                            {
           return 5.0;                  return 5; // an integer is cast to double
   }                            }
   ```

   both contain valid return statements. However, the following method,

   ```
   int gimmeFive()
   {
           return 5.0;    // cannot cast a double to an int
   }
   ```

   does not have a valid return statement because the double 5.0 does not match the int return type of the method, and 5.0 is not automatically cast to an integer.

   When a method executes the return statement,

   - the method terminates,
   - program control passes back to the caller, and
   - any statements following the return statement are ignored.
7. **Local Variables.** Variables that are declared within a method are called the *local variables* of that method. Local variables exist and are known only within the method in which they are declared. When a method exits, the local variables are destroyed. Local variables do not exist beyond the life of a method call. We now look at local variables in a bit more detail.

6.3.4 Local Variables

In Example 6.4, the parameters weight and distance, as well as the variable calories that is declared on line 8, are known only within the method caloriesBurned(…), that is, between the curly braces surrounding the statements of the method. The main(…) method can neither see nor access these variables. Similarly, myWeight, myDistance, totalCalories, and even input are known only in main(…). The memory cells, myWeight, myDistance, and totalCalories of Figure 6.8, are not visible when program control passes to caloriesBurned(…).

When a method is invoked, memory for local variables is allocated, and when a method exits, that memory is de-allocated.

Consequently, a method's local variables do not retain values from call to call. When a method exits, its local variables no longer exist.

Example 6.6 includes three methods. Each method has its own collection of local variables. Notice that the same name is used for more than one variable, yet the computer is not at all confused.

EXAMPLE 6.6 Rapid Rick of Example 6.4 runs in all weather, rain or shine, and in all seasons, hot or cold. The actual heat or cold he experiences depends on more than the outdoor temperature. The Summer Sizzle Index, *SSI*, measures what the temperature actually feels like on a hot day by taking into account the relative humidity. The Wind Chill Temperature, *WCT*, does the same for a cold day by taking wind speed into consideration. On a hot, sticky summer evening when the temperature is a not-so-balmy 80°F and the relative humidity is 77%, the *SSI* is 94.5°F. On a blustery winter day, when the temperature is a crisp 23°F and the wind speed is 20 mph, the *WCT* is 8.2°F.

The Summer Sizzle Index (*SSI*) and Wind Chill Temperature (*WCT*) are calculated as follows:

$$SSI = 1.98 * (T - (0.55 - 0.0055 * H)*(T - 58)) - 56.83$$
$$WCT = 35.74 + 0.6215 * T - 35.75 * V^{0.16} + 0.4275 * T * V^{0.16}$$

where T is the temperature (in Fahrenheit), H is the relative humidity (as a percent), and V is the wind velocity (miles per hour).

Problem Statement Write a program that, given the temperature and relative humidity, calculates both the Summer Sizzle Index or, given the temperature and wind speed, computes the Wind Chill Temperature.

Java Solution

```
1.  import java.util.*;
2.  public class HotAndCold
3.  {
4.     public static double summerSizzleIndex(double temperature, double relativeHumidity)
5.     {
6.        // calculates and returns Summer Sizzle Index
7.        // temperature is in degrees Fahrenheit; relative humidity is a percent
8.        double SSI = 1.98 *
              (temperature – (0.55 – 0.0055 * relativeHumidity) * (temperature – 58)) – 56.83;
9.        return SSI;
10.    }

11.    public static double windChillTemperature(double temperature, double windSpeed)
12.    {
13.       // calculates and returns Wind Chill Temperature
14.       // temperature is in degrees Fahrenheit; wind speed is mph
15.       double windChill = 35.74 + .6215 * temperature – 35.75 *
              Math.pow(windSpeed, 0.16) + 0.4275 * temperature * Math.pow(windSpeed, 0.16);
16.       return windChill;
17.    }

18.    public static void main(String[] args)
19.    {
20.       Scanner input = new Scanner(System.in);
21.       double temperature, SSI, windChill, relativeHumidity, windSpeed;

22.       System.out.print("To calculate SSI enter 1; to calculate Wind Chill enter 2: ");
23.       int reply = input.nextInt();
```

```
24.        System.out.print("Temperature: ");
25.        temperature = input.nextDouble();
26.        if (reply == 1)
27.        {
28.          System.out.print("Relative Humidity: ");
29.          relativeHumidity = input.nextDouble();
30.          SSI = summerSizzleIndex(temperature, relativeHumidity);
31.          System.out.println("Summer Sizzle index: " + SSI);
32.        }
33.        else
34.        {
35.          System.out.print("Wind Speed: ");
36.          windSpeed = input.nextDouble();
37.          windChill = windChillTemperature(temperature, windSpeed);
38.          System.out.println("Wind chill temperature: " + windChill);
39.      }
40.   }
41. }
```

Output

```
To calculate SSI enter 1; to calculate Wind Chill enter 2: 1
Temperature: 80
Relative Humidity: 75
Summer Sizzle index: 95.58049999999999

To calculate SSI enter 1; to calculate Wind Chill enter 2: 2
Temperature: 25
Wind Speed: 15
Wind chill temperature: 12.623095109603938
```

Discussion The HotAndCold class has three methods, each with a number of local variables, as shown in Figure 6.12.

summerSizzleIndex	windChillTemperature	main
temperature (parameter)	temperature (parameter)	temperature (line 21)
relativeHumidity (parameter)	windSpeed (parameter)	SSI (line 21)
SSI (line 8)	windChill (line 15)	windChill (line 21)
		relativeHumidity (line 21)
		windSpeed (line 21)
		input (line 20)
		reply (line 23)

FIGURE 6.12 Local variables in three methods

Although several local variables have the same name, the variables are, in fact, distinct. For example, each method has a variable named temperature. The three temperature variables may have the same name but each has its own storage location. They are independent and distinct. Of course, too many variables with the same name can lead to confusion and bugs. In general, try to give variables unique names.

The concept of local variables is tied to the broader topic of *scope*, which we discuss in the next section.

6.3.5 Scope

The *scope* of a variable is that section of the program in which a variable can be accessed or referenced.

For example, consider the following void method that computes the sum and product of the first *n* positive integers:

```
1.  void sumAndProduct(int n)
2.  {
3.    int sum = 0;
4.    int product = 1;
5.    for (int i = 1; i <= n; i++)
6.    {
7.      sum += i;
8.      product *= i;
9.    }
10.      System.out.println( "Sum of the first " + n + " positive integers is " + sum);
11.      System.out.println("Product of the first " + n + " positive integers is " + product);
12. }
```

The method sumAndProduct has several local variables: n, sum, product, and i. The scope of each of these variables is as follows:

- The scope of parameter n is the entire method.
- The scope of sum begins with its declaration on line 3 and extends to the end of the method.
- Similarly, the scope of product extends from its declaration on line 4 to the method's end.
- As you already know, the variable i does not exist beyond the block of the for-loop. Thus, the scope of variable i is lines 5 through 9. Outside of the for-loop, i is inaccessible and unknown.

In general, the scope of a variable begins with its declaration and extends to the end of the *block* in which it is declared.

Recall that a block is a group of statements enclosed by curly braces { and }; so if you declare a variable in the outermost block of a method, its scope extends from the declaration to the end of the method. On the other hand, the scope of a variable declared within an inner or nested block begins at the declaration and terminates at the end of that block. In the segment

```
if (purchase > 200)
{
      double discount = .20 * purchase;
      double discountPrice = purchase − discount;
      tax = .05 * discountPrice;
      total = discountPrice + tax;
}
else
{
      tax = .05 * purchase;
      total = purchase + tax;
}
```

the scope of the variables discount and discountPrice extends from their definitions to the end of the "if block." Thus, neither variable is known within the "else block."

The scope of a variable declared in the header of a for loop is the entire for loop. In the segment

```
for (int i = 0; i <= 50; i++)
{
        // statements
}
```

The control variable i is unknown once the loop terminates.

Example 6.7 illustrates a few of these general scope rules.

EXAMPLE 6.7

Player Polly is quite a fan of the board game Monopoly. When it is Polly's turn to roll the dice, if she rolls "doubles," (i.e., both dice show the same number of spots), Polly gets another toss of the dice. However, if she unfortunately tosses doubles three times in a row, then Polly must "go to jail." Polly frequently plays Monopoly and has landed in jail more than a few times. So, Polly was wondering how likely it is that she tosses doubles three consecutive times and lands in Monopoly prison.

Problem Statement Write an application that prompts the user for an integer, numTurns representing some number of Monopoly turns. Using random numbers, the program simulates rolling the dice for that many turns. Each turn consists of one, two, or three rolls of the dice, depending on whether or not doubles appear. The program keeps track of the number of simulated turns that results in three tosses of doubles and reports the number of jail terms as well as the percentage of jail terms incurred.

Java Solution

```
1.  import java.util.*;
2.  public class GoDirectlyToJail
3.  {
4.     public static int jailTerms(int turns)
5.     {
6.      // returns the number of turns that result in three rolls of doubles

7.      int threeDoubles = 0;                  // number of turns that result in three Doubles
8.      for (int i = 1; i <= turns; i++)       // for each turn
9.      {
10.        int numDoubles = 0;                 // counts the number of doubles on any one turn
11.        for (int toss = 1; toss <= 3; toss++) // up to three tosses/turn
12.        {
13.          // die1 and die2 are local to the inner block
14.          int die1 = (int)(6 * Math.random() + 1);
15.          int die2 = (int)(6 * Math.random() + 1);
16.          if (die1 == die2 )                // do the dice show the same number?
17.             numDoubles++;
18.          else
19.             break; // not doubles, so end the turn
20.        }
21.        if (numDoubles == 3)                // oops, go to jail
22.           threeDoubles++;
23.      }
24.      return threeDoubles;                  // the number turns giving three doubles
25.    }
```

```
26.    public static void main(String[] args)
27.    {
28.      Scanner input;
29.      input = new Scanner(System.in);
30.      int numTurns;
31.      int numJailTerms;                       // three doubles on any turn
32.      System.out.print("How many Monopoly turns would you like to simulate?");
33.      numTurns = input.nextInt();
34.      numJailTerms = jailTerms(numTurns);
35.      System.out.println("Number of times you got three doubles:" + numJailTerms);
36.      System.out.println("Percent of times you went to jail:" +
                    100 * (((double)numJailTerms/numTurns)) + "percent");
37.    }
38.  }
```

Output

```
How many Monopoly turns would you like to simulate? 100000
Number of times you got three doubles: 454
Percent of times you went to jail: 0.45399999999999996 percent
```

Discussion The simulation indicates that the probability of landing in jail is less than one-half of a percent. (In fact, the actual probability is 1/216, or about 0.46296 percent).

We now look at the local variables and the scope of each. The scope of each variable declared in main(…) extends from its point of declaration to the end of the method. However, the variables of the method jailTerms(…) are a bit more interesting. Figure 6.13 lists those variables along with the scope of each.

Local Variable	Scope
turns (parameter)	the entire method jailTerms(…)
threeDoubles	the entire method jailTerms(…)
i (line 8)	the entire for loop (lines 8–23)
numDoubles (line 10)	from the declaration on line 10 to the end of the block (lines 10–23)
toss (line 11)	the entire for loop (lines 11–20)
die1 (line 14)	from the declaration on line 14 to the end of the block (lines 14–20)
die2 (line 15)	from the declaration on line 15 to the end of the block (lines 15–20)

FIGURE 6.13 Scope of variables

6.3.6 Multiple *return* Statements

A method may have more than one return statement, but only one executes before the method terminates.

The first return statement that executes terminates the method. In Example 6.8, the method isPrime(…) contains several return statements. The return statement that executes, and thereby terminates the method, depends on the input data.

EXAMPLE 6.8

A prime number *p* is a positive integer greater than 1 that has no positive integer divisors other than 1 and *p*. For example, 101 is a prime number since no positive integers other than 1 and 101 divide 101 evenly. The integers 2, 3, 5, 7, and 37 are all prime numbers. On the other hand, 100 is not a prime number because 5 is a divisor of 100. With the exception of 2, all prime numbers are odd.

Prime numbers have fascinated mathematicians for centuries. In approximately 300 BCE, Euclid proved that there is an infinite number of primes. Even today, prime numbers are the foundation of modern cryptography. Indeed, factoring large numbers into primes is a task necessary for cracking modern cryptographic codes. "New" prime numbers are discovered every year. Currently, the largest known prime number is $2^{43,112,609} - 1$, which has 12,978,189 digits. Of course, a larger prime may be unearthed tomorrow, if that hasn't happened already!

Deciding whether or not an integer with 12,978,189 digits is prime is not an easy task. That said, a rather naïve, yet intuitive, scheme for determining whether or not a positive integer, *n*, is prime might check all possible divisors of *n* that are greater than 1 and less than *n*. If *n* has no divisor, then *n* is prime. This simple algorithm executes quickly for small values of *n*, but it is hopelessly slow for large values like $2^{43,112,609} - 1$ and the large numbers used in cryptography.

Problem Statement Write a program that prompts a user for a positive integer and determines whether or not the number is prime. Include a method

```
boolean isPrime(int p)
```

that accepts an integer p as a parameter and returns true if p is prime; otherwise false. See Figure 6.14.

FIGURE 6.14 The *isPrime* (...) method

Java Solution

```
1.  import java.util.*;
2.  public class PrimeChecker
3.  {
4.     public static boolean isPrime(int p) // returns true if p is a prime number
5.     {
6.       if (p <= 1)                    // 0, 1, and all negatives are not prime
7.          return false;
8.       else if (p == 2)               // if p is 2; return true (exit) because 2 is prime
9.          return true;
10.      else if ( p % 2 == 0)          // if p is even and not 2, return false (exit);
11.         return false;

12.         // so p is odd; check for odd divisors
13.         // if a divisor is found, return false and exit

14.         for (int i = 3; i < p; i += 2) // i = 3, 5, 7, 9...
15.           if (p % i == 0)               // if p % i == 0 then i divides p so p is not prime
16.              return false;

17.         // if the method reaches this point, p is prime,
18.          return true;
19.    }
```

```
20.    public static void main(String[] args)
21.    {
22.       int number;
23.       Scanner input;
24.       input = new Scanner(System.in);

25.       System.out.print("What number would you like to test? ");
26.       number = input.nextInt();
27.       if (isPrime(number))
28.           System.out.println(number + " is a prime number");
29.       else
30.           System.out.println(number + " is not prime");
31.    }
32. }
```

Output

What number would you like to test? **6317**
6317 is a prime number

What number would you like to test? **7163**
7163 is not prime

Discussion The logic behind the method isPrime(…) is described in the comments on lines 6, 8, 10, 12–15, and 17.

The method isPrime(…) contains no less than five return statements. When any one return statement executes, the method exits and program control passes back to the caller. For example:

- If parameter p has the value 22, the condition on line 10 is true, and the return statement on line 11 executes, returning false and terminating the method.
- If p has the value 35, the loop of line 14 executes, and when i attains the value of 5, the return on line 16 executes, returning false (because 35 % 5 == 0, i.e., 35 is divisible by 5).
- If p is 23, then none of the conditions of the else-if statement is true nor does the condition on line 15 evaluate to true. Consequently, the return statement on line 18 returns true, that is, 23 is prime.

6.4 METHOD OVERLOADING

> Java allows two or more methods of the same class to share the same name. This practice is called *method overloading*.

For example, Java's Math class has several overloaded methods, including Math.max(…), which has two forms:

1. int Math.max(int x, int y)
2. double Math.max(double x, double y)

Notice that the parameter lists of the two methods differ. The first version of Math.max(…) accepts two integer parameters and the second version accepts two double parameters.

> So that the Java compiler can distinguish between methods of the same name, overloaded methods *must* differ in the types and/or number of parameters.

Because of this rule, Java (usually) has no difficulty deciding which version of a method to execute. For example, consider the four calls to Math.max(…) shown in Figure 6.15.

Method Call	Returns	Argument Types	Version
Math.max(10,5)	10	two int	1
Math.max(10.0, 5.0)	10.0	two double	2
Math.max(10.0, 5)	10.0	two double (5 is automatically cast to 5.0)	2
Math.max(10, 5.0)	10.0	two double (10 is automatically cast to 10.0)	2

FIGURE 6.15 Four calls to the overloaded *Math.max(…)* method

Overloading can make your programs more readable and less cluttered, but there are also hazards and pitfalls. Example 6.9 illustrates the benefits as well as some of the pitfalls of method overloading.

EXAMPLE 6.9

Problem Statement Carrie Cash shops only at stores that offer deep discounts. Write a method,

double cost(double price, double discount)

that provides Carrie with help in calculating the sale price of an item. The cost(…) method accepts two arguments: the price of an item and the discount (both double), and it returns the marked-down price. Include the method in an application called Sales.

Java Solution

```
1.  public class Sales
2.  {
3.    public static double cost(double price, double discount) // 0.0 < discount < 1.0
4.    {
        // returns the marked down price, i.e. price after discount
5.      return price - discount * price;            // marked down price
6.    }
7.
8.    public static void main(String[] args)
9.    {
10.     System.out.println("Cost is " + cost( 25.50, 0.10 ));
11.   }
12. }
```

Output

```
Cost is 22.95
```

Discussion The method cost(…) accepts two double parameters signifying the retail price of an item and the discount rate (a decimal number less than 1). The method returns the reduced or marked-down price. The method is simple to understand and simple to use.

Now consider another rather common scenario in which a 10% discount is passed to cost(…) not as the decimal 0.10 but as the integer 10, that is, change line 10 to:

```
System.out.println("Cost is " + cost( 25.50, 10 )).
```

The program compiles, runs, and produces the following erroneous output:

```
Cost is −229.5
```

What happened? The argument 10 is automatically converted to a double 10.0 when it is passed to the (double) parameter discount. Consequently, the method calculates the marked-down price as

$$22.50 - \mathbf{10.0} * 22.50 = -229.5$$

To provide the flexibility of passing both integer and double arguments to cost(…), you can provide several versions of cost(…). The following program has four different versions of cost(…) that accommodate any combination of decimal and/or integer arguments

```
1.  public class SalesTwo
2.  {
3.     public static double cost(double price, double discount)   // version 1 – double, double
4.     {
5.       return price – discount * price;
6.     }

7.     public static double cost (int price, int discount)          // version 2 – int, int
8.    {
9.      double dollarsPrice = price / 100.0;                        // convert to dollars and cents
10.     double decimalDiscount = discount / 100.0;                 // convert to decimal
11.     return dollarsPrice – dollarsPrice * decimalDiscount;
12.   }

13.   public static double cost(double price, int discount)        // version 3 – double, int
14.   {
15.     return price – price * (discount / 100.0);
16.   }

17.  public static double cost(int price, double discount)         // version 4 – int , double
18.  {
19.     return (price / 100.00) – (price / 100.0) * discount;
20.  }

21.  public static void main(String [] args)
22.  {
23.     System.out.println("Cost is " + cost(25.50, 0.10));   // double, double
24.     System.out.println("Cost is " + cost(2550, 10));      // int, int
25.     System.out.println("Cost is " + cost(25.50, 10));     // double, int
26.     System.out.println("Cost is " + cost(2550, 0.10));    // int double
27.  }
28. }
```

The program produces the following output:

```
Cost is 22.95
Cost is 22.95
Cost is 22.95
Cost is 22.95
```

The four calls to cost(…) on lines 23–26 invoke versions 1–4, respectively. Any variation of argument types is acceptable. Thus, a single method name accommodates four situations. Certainly, this is simpler and clearer than using four different method names such as cost1, cost2, cost3, and cost4.

The previous program illustrates the niceties of overloading; nonetheless, method overloading does not come free of problems. For example, the following program with just two versions of cost(…) does not compile.

```
1.  public class SalesToo
2.  {
3.     public static double cost(double price, int discount) // double and int
```

```
4.     {
5.        return price - price * (discount / 100.0);
6.     }

7.     public static double cost (int price, double discount) // int and double
8.     {
9.        return (price / 100.00) – (price / 100.0) * discount;;
10.    }

11.    public static void main(String[] args)
12.    {
13.       System.out.println("Cost is " + cost( 25.50, 10 ));    // double, int
14.       System.out.println("Cost is " + cost(2550, 0.10));     // int, double
15.       System.out.println("Cost is " + cost( 25.50, 0.10));   // double, double
16.       System.out.println("Cost is " + cost(2550, 10));       // int, int
17.    }
18. }
```

Two of the calls to cost(…) in main(…) create problems. The first two calls, on lines 13 and 14, are perfectly legal. The argument types—(double, int) and (int, double)—match the types in the parameter-lists declared on lines 3 and 7, respectively. However, the call on line 15 with two double arguments generates a compiler error. Each cost(…) method requires one integer argument. Java does not automatically cast a double to an int. The compiler generates the following message indicating that there is no version of cost(…) that satisfies the call:

cannot find symbol
symbol : method cost(double,double)

Finally, the call to cost(…) on line 16 is also problematic but for a different and more subtle reason. Because both arguments are integers, the compiler issues the following error message:

reference to cost is ambiguous, both method cost(int,double) in SalesToo and method cost(double,int) in SalesToo match cost(2550, 10))

The ambiguity occurs because Java can, in fact, choose either method. On one hand, the Java compiler *could* cast argument 2550 to 2550.0 and choose the first method (line 3). On the other hand, the second argument 10 might be cast to 10.0 to accommodate the second method (line 7). Java has a choice of two methods; each method appears suitable. Wisely, Java refuses to make an arbitrary choice and generates an error message. In general, if an ambiguous choice exists, a program does not compile.

The overloaded methods of Example 6.9 are distinguishable because the data types of their parameter lists differ. Overloaded methods can also differ in the *number* of arguments that they accept. For example, you might have two versions of a method max(…):

1. int max(int x, int y)
2. int max(int x, int y, int z)

Version 1 returns the greater of x and y, and version 2 the greatest of x, y, and z. The method call

max(a,b)

with only two arguments invokes method 1 and the call

max(a,b,c)

with three arguments, invokes method 2. The number of arguments determines the version.

Example 6.10 includes two methods, both named OnBasePercentage, that calculate the percentage of times during a season that a baseball player gets to first base. The first method accepts four integer arguments and the second method expects five.

EXAMPLE 6.10 Baseball uses many different statistics to measure the performance of a hitter. The *On Base Percentage* is the percentage of times that a batter reaches first base. Historically, two formulas have been used to calculate this statistic: one that was developed during the 1950s and a more modern version created in 1984.

The method developed in the 1950s computes the On Base Percentage as:

$$(hits + walks + hbp) /(atBat + walks + hbp)$$

The 1984 version performs the calculation:

$$(hits + walks + hbp) /(atBat + walks + hbp + \boldsymbol{sacrifices})$$

where

atBat is number of times a player gets a hit or makes an out,
hits is the number of times a player gets a hit,
walks is the number of times a player walks,
hbp is the number of times a player was hit by a pitch, and
sacrifices is the number of times a player makes a sacrifice fly.

Problem Statement Write a program with two methods, each named OnBasePercentage, that calculate this statistic. The first method uses the older formula and the second uses its more modern counterpart.

In the following program, the main(…) method of the class Baseball displays the 1920 season statistics for Babe Ruth, including both calculations of "The Babe's" On Base Percentage.

Java Solution

```
1. public class Baseball
2. {

3.    public static double OnBasePercentage(int atBat,int hits,int walks,int hbp)
4.    // old method from the 1950's
5.    {
6.       return (double)(hits + walks + hbp) / (double)(atBat + walks + hbp);
7.    }

8.    public static double OnBasePercentage(int atBat,int hits,int walks, int hbp,int sacrifices)
9.    // new method from 1984
10.   {
11.      return (double)(hits + walks + hbp) / (double)(atBat + walks + hbp + sacrifices);
12.   }

13.   public static void main(String [] args)
14.  {
15.
16.           System.out.println("1920 statistics for Babe Ruth:");
17.           System.out.println("At bat: 458");
18.           System.out.println("Hits: 172");
19.           System.out.println("Walks: 150");
```

```
20.         System.out.println("Hit by pitch: 3");
21.         System.out.println("Sacrifice flies: 5");
22.         System.out.print("Babe's On Base Percentage (old method): ");
23.         System.out.println(OnBasePercentage(458,172,150,3)); // Babe Ruth's statistics
24.         System.out.print("Babe's On Base Percentage (new method): ");
25.         System.out.println(OnBasePercentage(458,172,150,3,5)); //Babe Ruth's statistics
26.   }
27. }
```

Output

```
1920 statistics for Babe Ruth:
At bat: 458
Hits: 172
Walks: 150
Hit by pitch: 3
Sacrifice flies: 5
Babe's On Base Percentage (old method): 0.5319148936170213
Babe's On Base Percentage (new method): 0.5275974025974026
```

Discussion The class Baseball contains two methods named OnBasePercentage, declared on lines 3 and 8. The first method requires four integer arguments and the second method five. Because the two parameter lists differ in the number of parameters, the Java compiler can easily choose a method based on the number of arguments the caller passes. The call on line 23 passes four arguments, and the call on line 25 passes five.

Examples 6.9 and 6.10 present two very simple variations of method overloading. As Example 6.9 illustrates, method overloading based on different data types can lead to problems when automatic type conversion occurs. On the other hand, overloading via a different number of parameters is much safer.

Finally, it is not legal to overload a method based on the type of the return value. The Java compiler does *not* consider

```
int       MyMethod(int x) and
double    MyMethod(int x)
```

two distinct methods. If two such declarations appear in the same class, a compilation error, complaining that MyMethod(…) is already defined, occurs.

> Attempting to overload a method based on the return type is a common error.

As in the two previous examples, overloaded methods must differ in the types and/or number of parameters. The return value is not a player.

Many of Java's library methods are overloaded. Figure 6.4 gives several examples such as:

```
Math.max(int a, int b) and Math.max(double a, double b)
```

or

```
Math.abs(int a) and Math.abs(double a).
```

Indeed, the methods appearing most often in this book, Java's print(…) and println(…) methods, are also overloaded. Each can take an argument of any data type.

6.5 IN CONCLUSION

In this chapter we describe a *method* as a black box that performs some singular task. Some methods accept arguments and some do not; some methods return a value, some do not; some methods are prepackaged with Java and others are written by the programmer. In all cases, however, methods simplify your programming tasks by separating a large problem into simpler components.

In Chapter 7, we present another programming structure, the *array*, which provides another method of program simplification, but in a very different way.

Just the Facts

- A *method* is a named sequence of instructions that are grouped together to perform a task.
- The name of a method must be a valid Java identifier. By convention, the name of a method begins with a lowercase letter and each succeeding "word" in the name begins with an uppercase letter.
- Every Java application must have a main(…) method, and every Java application begins execution with main(…).
- A Java method consists of a *header* followed by a *method block*:

```
modifiers return-type name (parameter-list) // the header
 {
     // the method block
 }
```

- The modifiers of a method header (for now) are the words public and static.
- A method's *parameter-list* consists of pairs of the form *type parameter* separated by commas. For example, int x, double y. Parameters are sometimes called *formal parameters*.
- The values passed to a method are called *arguments* or *actual parameters.*
- Arguments may be expressions.
- All arguments in Java are passed "by-value." This means that the values of the arguments are initially copied to the parameters of a method. Subsequently modifying the parameters in the method has no effect on the value of any variables passed as arguments.
- The method block performs the task of the method.
- The method block must include a return statement unless the method is a void method. A void method includes an implicit return, the last statement.
- When a return statement executes, the method exits and program control returns to the caller.
- The *scope* of a variable is the section of the program in which a variable can be accessed or referenced. The scope of a variable begins with its declaration and extends to the end of the block in which it is declared. For example, a variable declared in the header of a for statement is known only in the block of the for statement.
- Variables declared in the method block are the method's *local* variables and are inaccessible outside the block.

- Java allows two or more methods of the same class to share the same name. This practice is called *method overloading*. Overloaded methods must differ in the types and/or number of parameters.
- Method overloading based on different data types can lead to problems when automatic type conversion occurs. Overloading based on different numbers of parameters is safer.
- Java does not distinguish two methods based on the type of the value returned. Thus, method overloading based on the type of the return value is not allowed.

Bug Extermination

A method usually performs a single well-defined task. A method that performs several jobs is probably too complicated. Complex methods can lead to bugs that are hard to uncover. Keep things simple.

If an application includes several methods, you should implement and test them, one method at a time. When method A is working correctly, then implement and test method B. Write a method; compile it; test it. Then start the process again with the next method. Simulate data to test each method. When you are satisfied with one method then begin work on another. Bugs are easier to find when they are confined to ten lines rather than one hundred and ten.

The following list enumerates some of the more common bugs associated with methods:

- Omitting the keyword void from the header of a method that does not return a value.
- Omitting a return statement from a method that returns a value. Your method may have indeed computed the required value, but the return statement is necessary to send the value back to the caller.
- Misplacing a return statement. Once a return statement executes, the method exits. Make sure that return statements are correctly placed in a method block. If they are not, code that you intended to execute may not execute.
- Specifying your arguments incorrectly. Multiple declarations are not allowed in a parameter list:

  public void example(int a, int b, double c) is correct.
  public void example(int a, b, double c) is not.

- Overloading a method based on the return type. The return type does not distinguish one method from another. The parameter-lists of overloaded methods must differ.
- Attempting a method call with the incorrect number of arguments.
- Passing arguments of the wrong type to a method. The arguments must match the parameter-list not only in number but also in type.
- Passing arguments to a method in the wrong order. For example, when calling

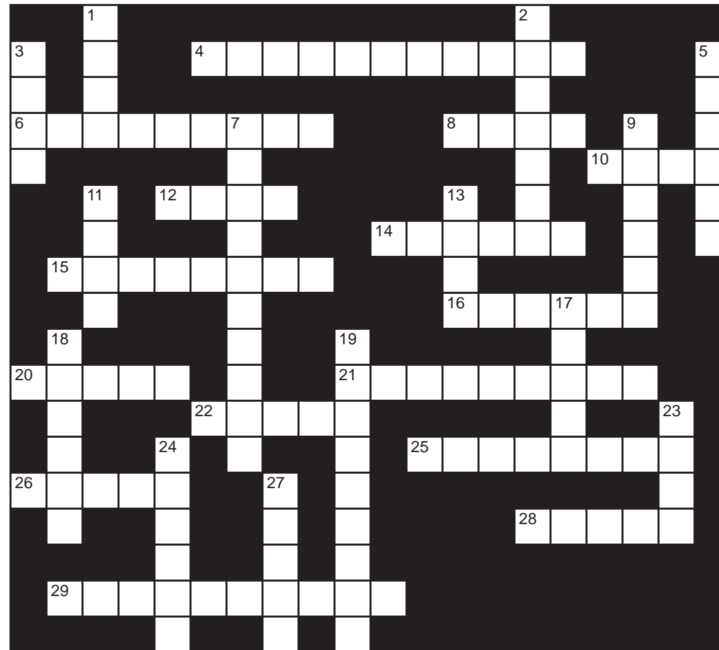
  ```
  double area( double length, double width) // area of a rectangle
  ```

  the first argument of the call should signify the length of a rectangle and the second, the width. If you reverse the arguments, your program will compile and run, but your results will not necessarily be correct.
- Omitting the empty parentheses () when invoking a method with no arguments.
- Overloading a method based on data type that results in an ambiguous choice for the compiler. When, due to automatic casting, there is a choice of more than one method to match a method call, the compiler issues a syntax error. When overloading methods, be sure that no ambiguity exists about which method is appropriate.

EXERCISES

LEARN THE LINGO

Test your knowledge of the chapter's vocabulary by completing the following crossword puzzle.



Across

4 A non-void method must specify a _____(two words)
6 The words public and static are ________
8 If the data type of the return type is T, then the type of the returned value must be T or a type that can be automatically _____ to T
10 Java library containing random()
12 Overloaded methods differ by the _____ or number of parameters
14 A method's ______ gives its name and parameter list
15 Used to pass a value to a method
16 The parameter-list specifies the type and _____ of arguments that must be given to a method
20 The _____ of a variable is the section of the program in which a variable can be accessed
21 If the choice of an overloaded method is _____ the compiler issues an error message
22 By convention, the name of a method begins with a(n) _____-case letter
25 Metaphor for a method
26 When a method is invoked, the _____ of an argument is passed to the method
28 Java-supplied void method
29 _____ methods share the same name

Down

1 Method that does not return a value
2 A variable declared in a block is unknown _____ that block
3 Local variables declared in different methods of the same application may have the same ________
5 Named set of instructions that performs a task
7 An argument may be a(n) ________
9 When a method exits, control returns to the ________
11 Java method for square root
13 Method that is always executed first
17 The statements of a method comprise the method ________
18 Another name for an argument is a(n) _____ parameter
19 Receives a value passed to a method
23 A return statement causes a method to ________
24 Not necessary in a void method
27 Variable declared in a method

SHORT EXERCISES

1. **True or False**
   If false, give an explanation.
   a. myMethod(…) may be overloaded as:
      int myMethod(int x, int y) and float myMethod(int x, double y).
   b. yourMethod(…) may be overloaded as:
      int yourMethod(int x, int y) and int yourMethod(int x, double y).
   c. hisMethod(…) may be overloaded as:
      int hisMethod(int x, int y) and float hisMethod(int x, int y).
   d. herMethod(…) may be overloaded as:
      int herMethod(int x, int y) and int herMethod(int x, int y, int z).
   e. Every Java application begins execution with main(…).
   f. main(…) can invoke at most three other methods.
   g. A method can call a method that in turn calls another method.
   h. Overloaded methods must have a different number of parameters.
   i. Overloaded methods must return the same type of data.
   j. The parameters in the header of a method are called the *actual* parameters.
   k. Arguments can be expressions or constants.
   l. The type of each parameter must match the type of its corresponding argument.
   m. The scope of a parameter in a method extends to the end of the method.
   n. The scope of a local variable extends to the end of the method in which it is defined.
   o. Every method returns a value.
   p. The name of a method cannot begin with an uppercase letter.
   q. Methods provide a programmer with a mechanism to segment a complicated application into simpler and easier-to-debug components.
   r. A method can use the same name for a local variable and a formal parameter.

2. **Playing Compiler**
   Determine the errors in each of the following segments. Fix the errors and then determine the output. Unusual formatting is not an error.

   a.
```
public class WhatTheHey
{
     public static int method1(int x,y)
    {
        return x + y;
    }

   public int method2(double x, double y)
   {
      return int(x − y);
   }

   public static void main(String[] args)
   {
     System.out.println("The output is: ", method1(method2(7.1, 6.2), method1(2, 3))
      "years of  bad luck" );
   }
}
```

b.
```
public class TheBookOnLove
    {
        public static void method1()
        {
            System.out.println("I wonder who wrote the book on love");
        }
        public static void method1(int x)
        {
            for (i = 0; i < x; i++)
            System.out.println("I wonder who wrote the book on love") + i;
        }
        public static void main(String[] args)
        {
            int i; for (i = 0; i < 6; i++)
            {
                System.out.println(int i);
                method1();
                method(i);
            }
        }
    }
```

c.
```
public class ThisComputesSomeWeirdStuff
{
    public static int method1(int a, int b)
    {
        if (a%2 == 0) return (a) else return (b)
    }
    public static int method2(int a, int b)
    {
        while (a != 1) {b++; a = a / 2;} return (b);
    }
    public static void main{String[] args}
    {
        System.out.println(method2(method1(3, 10), method1(16, 57)));
        System.out.println(method2(method1(190, 10), method1(16, 57)));
        System.out.println(method2(method2(3, 10), method1(16, 57)));
        System.out.println(method1(method2(3, 10), method2(16, 57)));
    }
}
```

d.
```
public class ThisOneIsPrettyCool
{
    public static int method1(int w)
    {
        int count = 0;
        while (w != 1) if (w % 2 == 0)
        {
            w = w / 2;
            count++;
        }
```

```
        else
            w = 3 * w + 1;
        return count;
    }
    public static void main(String[] args)
    {
        System.out.println(method1(10));
        System.out.println(method1(7));
    }
}
```

e.
```
public class OkIveHadEnough
{
    public static double method1(int a) { return a / 2;}
    public static double method1() {return 1.0;}
    public static int method2(double x) {return 3 * (int)x;}
    public static int method2() {return 0;}
    public static void main(String[] args)
    {
        System.out.println(method2(method1()));
        System.out.println(method1(method2()));
        for (int j = 0; j < 10; j++)
        {
            System.out.println(method2(method1(j));
            System.out.println(method1(method2(j)));
        }
    }
}
```

3. **Method Acting**
Methods can be used to accomplish each of the following tasks. Write only the method *headers* for each example. Overload a method name, if appropriate.

a. Calculate the largest of 2, 3, or 4 integer values.
b. Calculate your federal income tax percentage based on the following chart:

Adjusted Gross Income Range			Percentage
$0	–	$7,300.00	10%
$7,300.01	–	$29,700.00	15%
$29,700.01	–	$71,950.00	25%
$71,950.01	–	$150,150.00	28%
$150,150.01	–	$326,450.00	33%
$326,450.01	–	and up	35%

Allow the income to be expressed in dollars and cents, or simply rounded to the nearest thousand dollars. That is, an adjusted gross income of 52,736.98 may alternatively be expressed as 53.

c. Calculate the percentage score on an exam. You are given the number of questions on the exam, and the number that are correct.

d. Calculate your risk factor (*RF*) for auto insurance in MA, NY, and NJ. Your risk factor is either a, b, c, d, or e. In MA, *RF* depends on your age, the number of charged accidents on your record, and the number of traffic violations. In NY, *RF* depends on your age, your driving points (an integer between 6 and 35, inclusive), and the total dollars paid out to you in charged accident claims. In NJ, *RF* depends on your age, the distance from your home to the NY border (rounded to the nearest mile), the number of traffic violations on your record, and the number of people under 30 years of age in your family.

e. Decide whether or not you are eligible to become president. Eligibility is determined by your year of birth, the first letter of the country in which you were born, and the number of years that you have been a U.S. resident.

4. **Overloaded Methods**

a. Method add(…) is overloaded as follows:

```
static double add( int a, double b)      static double add(double a, int b)
{                                        {
      return a + b;                            return a + b;
}                                        }
```

Which, if any, of the following invocations fail to compile? Give reasons.

```
i.   add(1,2)
ii.  add(1.0,2.0)
iii. add(1.0, 2)
iv.  add(2.0,2)
```

b. Method sub(…) is overloaded as follows:

```
int sub( int a, int b)          double add(double a, double b)
{                               {
        return a - b;                   return a - b;
}                               }
```

Which, if any, of the following invocations fail to compile? Give reasons.

```
i.   sub(1,2)
ii.  sub(1.0,2.0)
iii. sub(1.0, 2)
iv.  sub(2.0,2)
```

c. What is the problem with the following overloaded method that returns a product as either an int or a long?

```
int mul(int a, int b)           long mul(int a, int b)
{                               {
        return a * b;                   return a * b;
}                               }
```

5. **Pass By Value**

Harry Hacker has written the following method that is supposed to swap the contents of two variables:

```
static void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

However, the statements

```
int a = 5;
int b = 6;
swap(a,b);
System.out.println("a = " + a + " and b = " + b);
```

produce the output

```
a = 5 and b = 6.
```

Explain why Harry's method does not work as intended.

PROGRAMMING EXERCISES

1. **Min and Max**
   Write two methods

   ```
   int myMax(int x, int y) and
   int myMin(int x, int y),
   ```

   each of which accepts two integers x and y, and outputs the larger/smaller of the two, respectively. The main method of your program should prompt for two numbers, pass these numbers to myMax(…) and myMin(…), and then print the results with appropriate explanatory text.

2. **Celsius to Fahrenheit**
   Write a method

   ```
   int cToF(int x)
   ```

   that converts a Celsius temperature to a Fahrenheit temperature. The conversion formula is:

   $$F = (9.0/5.0)C + 32.$$

   The returned value should be rounded to the nearest degree. Test your method by displaying a table of Celsius temperatures from −40 to 100, in increments of five degrees, with the Fahrenheit equivalents.

3. **Random Numbers**
   Write a method

   ```
   int randomInt(int x, int y)
   ```

   that returns a random integer between x and y, inclusive. Note that x and y can be positive or negative.

4. **Average**
   Write a method

   ```
   double average(int n)
   ```

that reads n numbers of type double and returns the average of those numbers. Include this method in a program that requests a value for n and displays the average of n numbers supplied by a user.

5. **Consumer Price Index**
The Consumer Price Index (CPI) represents the change in the prices paid by urban consumers for a representative basket of goods and services. It is a percentage value rounded to the nearest tenth, for example, 9.2 or −0.7. Write a method

```
double getCPI()
```

that asks a user to enter a number between −20 and 20 with one digit after the decimal point. If the user supplies an unacceptable number, the method should display an appropriate error message ("number too high," "number too low," or "number has wrong precision") and prompt the user for another value. When the user succeeds, the method should return that number.

Test your method by continually prompting a user for a value and displaying the value. When you are confident that the method is correct, write a second method

```
double inflation( double cpi, double expenses)
```

that accepts the CPI and last year's annual expenses. Method inflation(…) returns what you might expect to pay for the same goods in the coming year. Write a main(…) method that calls both getCPI() and inflation(…).

6. **Price Adjustment**
Write a method

```
int bumpMe(int price, int increase, boolean updown)
```

that accepts a price in dollars and returns a new price rounded to the nearest dollar, after increasing/decreasing price by increase percent. If updown is true then you should increase the price; otherwise, decrease the price. Write an appropriate main(…) method to test bumpMe(…).

7. **Simulations**
Simulation is one way that casinos analyze games; simulation is less expensive than hiring a mathematician. The "over-under" bet is described in Example 6.3. Write three methods, each of which simulates 10,000 plays of a $1/bet game and returns the amount of money that is won or lost over 10,000 games. A negative number denotes a loss. The three methods operate as follows:

- Method 1 chooses the bet ("over 7," "under 7," or "exact") at random.
- Method 2 always chooses the "over 7" bet.
- Method 3 chooses the "over 7" bet 4000 times, the "under 7" bet 4000 times, and the "exact" bet 2000 times.

Test these methods in a program. Write, test, and debug the methods one at a time, that is, get Method 1 working perfectly before including Method 2 in your program.

8. **Hello World Revisited**
Write a program that prompts a user for a positive integer *n* and prints "Hello There" *n* times. Of course, a value of *n* that is less than or equal to 0 is illegal. To ensure valid input, include a method

```
int getPos()
```

that prompts for a positive integer. If the value of that integer is less than or equal to 0, the method should print an appropriate message and request a positive number. When the user supplies a valid number, the method returns that number.

9. **Carnival Game Simulation**
The rules of a certain carnival game stipulate that a player throws one standard 6-sided die, one 20-sided die, one 8-sided die, one 4-sided die, and one 12-sided die. The player wins if the total on the five dice is greater than 35 or less than 20. Write a program that simulates the carnival game 100 times and reports the number of times a player wins. Your program should include a method

```
int dieRoll(int x)
```

that returns a random number between 1 and x.

10. **Present Value of an Investment**
The present value on an investment of A dollars for Y years at an annual rate of R percent compounded C times yearly is

$$\text{Present Value} = A(1 + R/C)^{YC} \tag{1}$$

Of course, if interest is compounded yearly, then $C = 1$ and (1) simplifies to:

$$\text{Present Value} = A(1 + R)^{Y}. \tag{2}$$

Overload a method presentValue(…) so that presentValue(…) implements formulas (1) and (2). Write a main(…) method that tests both versions of presentValue(…).

11. **Craps Simulation**
When playing craps, a player rolls two dice repeatedly until she wins or loses. The first roll of the dice is called the *come-out* roll. If the player rolls a 7 or an 11 on the come-out roll, then she wins immediately; a 2, 3, or 12 on the come-out roll results in an immediate loss. If she rolls a 4, 5, 6, 8, 9, or 10 on the come-out roll, then that number becomes her *point* and she continues rolling until she rolls either her point or a 7. If she rolls her point, she wins, but if she rolls a 7 before rolling her point, she loses. Once a player has established her point, no other numbers (including 2, 3, 11, or 12) affect her winning or losing.
Write a method

```
boolean craps()
```

that simulates one game of craps and returns *true* if and only if the player wins. Test your method by printing the values of each roll of the dice. When you are convinced that your simulation is correct, include this method in a program that executes craps() 1000 times and reports the percentage of wins.

12. **Mean Versus Median**
Implement two methods:

    1. int median(int x, int y, int z) that calculates the median of three integers.
    2. int mean(int x, int y, int z) that calculates the average of three integers, rounding the result to the nearest integer.

    Devise a main(…) method that accepts three integers and states whether the median of the three is larger than the mean, smaller than the mean, or equal to the mean.

13. **Zeno's Paradox**
A famous paradox devised by Zeno, an Eleatic philosopher (b. 488 BCE), asserts that to run from point A to point B, a runner must first traverse half the distance between A and B. Before he can do that, he must traverse a "half of the half," and so on ad infinitum. He must, therefore, pass through an infinite number of points, and that is impossible in a finite time. Design and implement a method

```
double zeno(int n)
```

that calculates the sum 1/2 + 1/4 + 1/8 + … + $1/2^n$. The method should call an auxiliary method

```
int powerTwo(int n)
```

that returns 2^n. Test both methods in a complete program. *Hint:* Implement and test the two methods one at a time. First write, compile, and test powerTwo(…). Once that method is working correctly, add zeno(…) to your program.

14. **Date Calculations**
Implement an application that prompts for two dates each comprised of three integers: a month, a day, and a year between 1900 and 2100, inclusive. If both dates are valid, your program should display the number of days between the two dates; otherwise, your program should issue an error message. Include a method

```
boolean validDate(int month, int day, int year)
```

that returns true if and only if a date is valid. For example (12, 29, 1980) is valid but (29, 12, 1980), (13, 11, 2007), and (1, 1, 1899) are not. You should also design a method

```
int dateDifference(int day1, int month1, int year1, int day2, int month2, int year2)
```

that returns the number of days between the dates (month1, day1, year1) and (month2, day2, year2), provided these dates are valid. Don't forget to take leap years into account, and recall that 1900 and 2100 are *not* leap years.

15. **Geometric Mean**
A home purchased for 300,000 dollars increases in value by 10% after one year and by another 20% after a second year. Thus, a year after purchasing the house, its value is **1.10** × 300,000 dollars and after two years **1.20** × 1.10 × 300,000 dollars. A third year *decrease* of 6% drops the value to 94% of the previous value. **0.94** × 1.20 × 1.10 × 300000 = 372240 dollars. Notice that the multiplier is 1 − 0.06 = 0.94. The *average* annual increase over the three-year period is the *geometric mean* of 1.10, 1.20 and 0.94.

In general, the *geometric mean* of *n* numbers is the nth root of their product. Thus, the geometric mean of 1.10, 1.20, and .94 is $(1.10 \times 1.20 \times .94)^{1/3}$ = 1.074568 and the product 1.074568 × 1.074568 × 1.074568 × 300000 equals 372240, as does the original product, .94 × 1.20 × 1.10 × 300000. In other words, the home's value after three annual changes of 10%, 20%, and −6% is the same as if, each year, the home's value increased by 7.4568%.

Write a program that calculates the average increase or decrease on an investment held from one to six years. Your program should first prompt for the length of time of the investment (an integer between 1 and 6, inclusive) and then the percent increase or decrease for each year. A negative number indicates a decrease. The program should display the average annual increase or decrease. For example, if a home, over a six-year period, has changed in value by 10%, 20%, 6%, −8%, −12%, and 3%, then to compute the average annual increase (or decrease), you would calculate the geometric mean of 1.1, 1.2, 1.06, 0.92, 0.88, and 1.03.

Hints: Overload a method, geometricMean(…). Make five versions that have two, three, four, five, and six parameters of type double. Use Math.pow(x,y).

16. **Harmonic Mean**
If it takes one hose 12 hours to fill a pool, and another hose 4 hours, then together they fill the pool in (2 × 4 × 12) / (4 + 12) = 6 hours. The *harmonic mean* of two positive numbers *a* and *b* is $2ab/(a + b)$. Write a method

```
double harmonicMean(int x, int y)
```

that returns the harmonic mean of $a > 0$ and $b > 0$. Write another method that returns the arithmetic mean of a and b, that is, the average of a and b. Finally, include a third method that returns the geometric mean of a and b, that is, the square root of $a \times b$ (see Exercise 15).

Test your methods in a program that reads two positive integers and displays their harmonic mean, arithmetic mean, and geometric mean. For example, if a and b have values 12 and 4, the harmonic mean is 6.0, the arithmetic mean is 8.0, and the geometric mean is $\sqrt{48} \approx 6.928$. Did you notice that the harmonic mean times the arithmetic mean equals the square of the geometric mean? This identity might be helpful to you when you design your methods.

17. **Median of Five**
A teacher wishes to use the median (middle value) of five grades as the final grade for each of n students. Write a method that returns the median of five integers. For example, the median of 10, 50, 48, 35, and 22 is 35. Test your method in a program that accepts the number of students, followed by five grades per student, and prints the final grade for each student. Do *not* assume that the grades are ordered.

18. **Lottery Games**
Most states sell lottery tickets of one of the following two types:
    a. A player picks k distinct numbers between 1 and n, inclusive. For example, to play Massachusetts' Megabucks game, a player picks six numbers between 1 and 42. In this case, the number of possible lottery tickets is:

$$\frac{42 \times 41 \times 40 \times 39 \times 38 \times 37}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 5{,}245{,}786$$

    (Notice that six numbers must be selected and there are six factors in the numerator, counting down from 42.) Thus, a player who buys a single ticket has one chance in 5,245,786 of attaining an instant fortune. In general, if a player must choose k numbers between 1 and n, the number of possible tickets is:

$$\frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{k \times (k-1) \times (k-2) \times \ldots \times 3 \times 2 \times 1}$$

    b. The second type of game requires that a player pick k numbers between 1 and n as well as one additional number between 1 and m. For example, to play California's Super Lotto game, a player picks five numbers between 1 and 47, inclusive, and one additional number between 1 and 27, inclusive. In this case the number of possible tickets is:

$$\frac{47 \times 46 \times 45 \times 44 \times 43}{5 \times 4 \times 3 \times 2 \times 1} \times 27 = 41{,}416{,}353$$

    In general, the number of possibilities is:

$$\frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{k \times (k-1) \times (k-2) \times \ldots \times 3 \times 2 \times 1} \times m$$

Write an application that calculates the number of possible lottery tickets for each type of game, (a) and (b). Overload numberOfTickets(...) as

```
int numberOfTickets(int n, int k) // choose k numbers from 1 to n
```

and

```
int numberOfTickets( int n, int m, int k)    // choose k numbers from 1 to n
                                             // and an additional number from 1 to m.
```

These methods return the number of possibilities described in (a) and (b) above, respectively.

To play New York's Lotto game, a player picks six numbers between 1 and 59; and to play the state's Mega Millions game, a player picks five numbers between 1 and 56 and an additional number between 1 and 46. Write a main(…) method that determines which of the two games gives the better chance of an instant fortune.

THE BIGGER PICTURE

1. TIME COMPLEXITY

The amount of time it takes to run a program is the most important measure of program performance. A clock can be used to measure the real running time of a program, but results can be misleading if programs are run on different computers. Some computers are faster than others, and a fast computer might conceivably run a poorly designed program in less time than a slow computer runs a well-designed one.

A better measure of performance treats the program as an abstract *algorithm*—that is, a step-by-step method for solving a problem—and calculates the number of steps that the algorithm requires as a function of input size. This focus on the algorithm cuts out the disparity in hardware and allows an even-handed comparison.

For example, consider the following algorithm that computes the average of n integers:

```
sum = first integer;
for each of the remaining n − 1 integers
{
   sum = sum + next integer;
}
average = sum/n;
```

This algorithm takes approximately n steps to accomplish its task, where each step is a single addition or division. The *time complexity* of the algorithm is n. A Java implementation would accomplish the task using a loop.

Following are two algorithms, written in Java-like pseudocode, that calculate the *greatest common divisor* (*gcd*) of two numbers a and b, where $a > b$. The greatest common divisor of a and b is the largest positive integer that evenly divides both a and b. For example, the greatest common divisor of 36 and 27 is 9, and the greatest common divisor of 101 and 68 is 1.

Algorithm I:

```
// This is a brute force algorithm that starts with the smaller number (b) and finds
// the first number less than or equal to b that divides evenly into both a and b.
// Assume a > b.

   for k = b downto 1                            // for each possible divisor, k
      if ( (b % k == 0) && (a % k == 0) )       // if k evenly divides both b and a
         return k;
```

Algorithm II:

```
// This is a clever, sophisticated algorithm called Euclid's algorithm.

        while (b != 0)
        {
            int temp = a % b;
            a = b;
            b = temp;
        }
          return a;
```

It should be clear why Algorithm I works, but perhaps it is not so obvious why Algorithm II accomplishes the same task. Algorithm II is based on a theorem of Euclid (300 BCE) that states that given two positive integers *a* and *b*, where $a > b$,

> the greatest common divisor of *a* and *b* is the same as the greatest common divisor of *b* and *a* % *b,* that is, gcd(*a*,*b*) = gcd(*b*, *a* % *b*).

For Algorithm I, the worst-case scenario is that the loop iterates from *b* down to 1. So, at worst, Algorithm I takes *b* steps. For example, to calculate the greatest common divisor of 101 and 37, Algorithm II requires 37 steps.

For Algorithm II, the number of steps, that is, the time complexity, is not obvious. However, 19th century mathematician Gabriel Lamé proved that Euclid's algorithm requires at most 5*k* steps, where *k* is the *number of digits* in *b*.[1] For example, Euclid's algorithm takes at most $5 \times 2 = 10$ steps to find the greatest common divisor of $a = 472$ and $b = 36$, since 36 has two digits.

Compare 5*k* with the time complexity of Algorithm I, which requires at most *b* steps. The difference is astronomical. For example, if $b = 10^{10} = 10{,}000{,}000{,}000$, then Algorithm I may take 10^{10} steps, but Euclid's algorithm takes at most $2 \times 11 = 22$ steps! Because a number with *n* digits is roughly 10^n, the difference between the two algorithms is akin to the difference between 10^n and *n*.

Exercises

1. Estimate the number of steps used by Algorithm I and Algorithm II when $a = 298765$ and $b = 89765$.

2. Implement methods for Algorithm I and Algorithm II. Include your implementations in a program.

3. Write a program that compares the running times of Algorithm I and Algorithm II. Use the data: $a = 12000111$, $b = 9899111$. To calculate the running time of each method, invoke System.currentTimeMillis(), which returns the current time (long) to the nearest millisecond:

```
long startTime = System.currentTimeMillis();
myMethod(); // call your method here
```

[1] Lamé used the *Fibonacci sequence* to prove his result. The Fibonacci sequence is a sequence of positive integers such that the first two terms of the sequence are both 1 and each succeeding term is the sum of the previous two numbers. The first 10 terms of the Fibonacci sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55. Lamé proved that if $a > b \geq 0$ and b is less than the $(n + 1)^{st}$ term of the Fibonacci sequence, then the number of steps required by Euclid's algorithm is at most *n*. For example, if *b* is 100, then the number of steps required by Euclid's algorithm is at most 11 because 100 is less than the 12th term of the Fibonacci sequence: (1 1 2 3 5 8 13 21 34 55 89 **144**). Lamé's theorem implies that the number of steps required by Euclid's algorithm is no more than 5 times the number of digits in *b*.

```
long endTime = System.currentTimeMillis();
long totalTime = endTime − startTime;
```

How do the two algorithms (I and II) compare?

4. To each method of Exercise 2, add a counter that keeps track of the number of loop iterations, that is, the number of steps performed by each algorithm. Compare the number of steps required by Algorithm I and Algorithm II when a = 12000111 and b = 989111. Are your results consistent with the theory? Run the program with numbers of your own choice.

2. RECURSION, A PREVIEW

You have seen that a method can invoke another method. Well, you may be surprised to learn that a method can call *itself*. A method that includes a call to itself is called a *recursive* method. You might surmise that a method that calls itself would create an infinite loop. And, indeed, that may happen. Trace through the following method that forever begs you to end its misery!

```
Public static void runMeForever()
{
   System.out.println("Stop me. This hurts!");
   runMeForever();
}
```

The method runMeForever() produces the following, rather redundant, cry for help:

```
Stop me. This hurts!
Stop me. This hurts!
Stop me. This hurts!
Stop me. This hurts!
Stop me. This hurts!
Stop me. This hurts!
Stop me. This hurts!
...
```

Yes, the runMeForever() invokes runMeForever() which invokes runMeForever() which invokes runMeForever(), well, forever. However, you can rewrite this method so that it prints the message just n times and then stops. As a boolean condition terminates a while loop, we can use a boolean condition to put a stop to the infinite chain of recursive method calls.

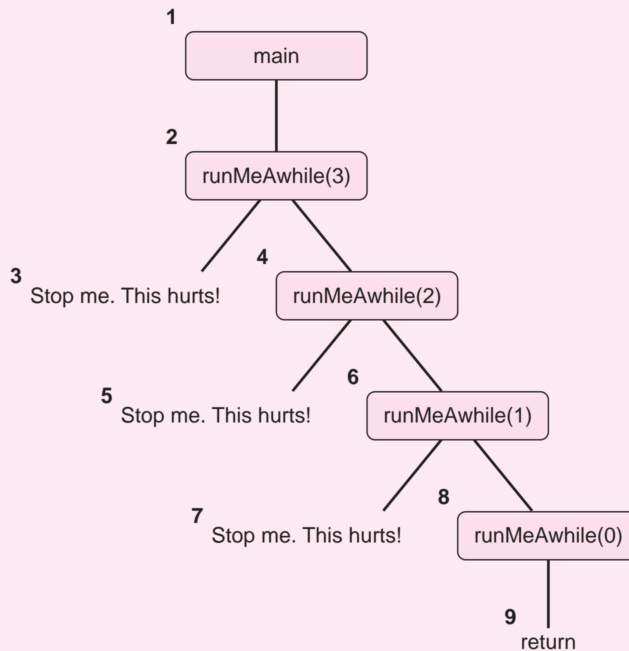
```
public static void runMeAwhile(int n)
{
        if (n != 0) // stops the chain of method calls to itself when n=0
        {
                System.out.println("Stop me. This hurts!");
                runMeAwhile(n − 1 );
        }
}
```

The following class invokes the method runMeAwhile(…) with the argument n = 3.

```
public class Test
{
   public static void runMeAwhile(int n)
   {
```

```
            if (n != 0)
            {
                  System.out.println("Stop me. This hurts!");
                  runMeAwhile(n – 1 );
            }
      }
      public static void main(String[] Args)
      {
            runMeAwhile(3); // invokes method for the first time
      }
}
```

The main(…) method of Test calls runMeAwhile(3) which displays "Stop me. This hurts!" and calls runMeAwhile(2), which prints "Stop me. This hurts!" and calls runMeAwhile(1), which prints "Stop me. This hurts!", and calls runMeAwhile(0), which does nothing because n == 0. The following diagram depicts the actions of Test:



In theory, recursion and iteration are equivalent; anything that you can accomplish with one you can do with the other. Java provides both recursion and iteration for the same reason that it provides three different loops (do-while, while, and for): different problems are solved more naturally with different tools.

Recursion, however, is a powerful way of thinking and problem solving that extends well beyond the notion of loops. Recursion is one of the major techniques employed in the development of powerful computer algorithms. A more thorough discussion of recursion follows in Chapter 8.

Exercises

1. Write a recursive method int getPos() that requests a positive integer supplied by a user. On input less than or equal to 0, the method displays an appropriate error message and asks again for a positive integer via a recursive call to itself. If the number is legal, the method returns the number.

2. Write a recursive method int addUp(int n) that returns the sum of the numbers from 1 through *n*. *Hint*: addUp(n – 1) will return the sum of the numbers from 1 through *n* – 1. All you need to do is add *n* to this sum and return.
3. Test the methods of Exercises 1 and 2 in a program that includes the following main(…) method:

```
public static void main(String[] args)
{
   System.out.println("Enter a positive integer: ");
   int n = getPos();
   System.out.println("The sum 1 through n is "+ addUp(n));
}
```

4. Determine the output of the following program. If you trace through the method calls carefully, you will discern a pattern.

```
public class Recursion
{
     public static int mystery(int a, int b, int c, int d)
     {
          if (a == b)
             return c;
          else
             return mystery(a, b + 1, d, c + d);
     }
     public static void main (String[] args)
     {
          for(int i = 1; i < 10; i++)
             System.out.println(mystery(i,1,1,1));
     }
}
```