CHAPTER

6

METHODS

Objectives

- To define methods with formal parameters (§6.2).
- To invoke methods with actual parameters (i.e., arguments) (§6.2).
- To define methods with a return value (§6.3).
- To define methods without a return value (§6.4).
- To pass arguments by value (§6.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§6.6).
- To write a method that converts hexadecimals to decimals (§6.7).
- To use method overloading and understand ambiguous overloading (§6.8).
- To determine the scope of variables (§6.9).
- To apply the concept of method abstraction in software development (§6.10).
- To design and implement methods using stepwise refinement (§6.10).

6.1 Introduction

Methods can be used to define reusable code and organize and simplify coding.

problem

Suppose that you need to find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively. You may write the code as follows:

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

why methods?

You may have observed that computing these sums from 1 to 10, from 20 to 37, and from 35 to 49 are very similar except that the starting and ending integers are different. Wouldn't it be nice if we could write the common code once and reuse it? We can do so by defining a method and invoking it.

The preceding code can be simplified as follows:

define sum method

main method

invoke sum

```
1  public static int sum(int i1, int i2) {
2    int result = 0;
3    for (int i = i1; i <= i2; i++)
4      result += i;
5
6    return result;
7  }
8
9  public static void main(String[] args) {
10   System.out.println("Sum from 1 to 10 is " + sum(1, 10));
11   System.out.println("Sum from 20 to 37 is " + sum(20, 37));
12   System.out.println("Sum from 35 to 49 is " + sum(35, 49));
13 }
```

Lines 1–7 define the method named `sum` with two parameters `i1` and `i2`. The statements in the `main` method invoke `sum(1, 10)` to compute the sum from 1 to 10, `sum(20, 37)` to compute the sum from 20 to 37, and `sum(35, 49)` to compute the sum from 35 to 49.

method

A *method* is a collection of statements grouped together to perform an operation. In earlier chapters you have used predefined methods such as `System.out.println`, `System.exit`, `Math.pow`, and `Math.random`. These methods are defined in the Java library. In this chapter, you will learn how to define your own methods and apply method abstraction to solve complex problems.

6.2 Defining a Method

A method definition consists of its method name, parameters, return value type, and body.

The syntax for defining a method is as follows:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

Let's look at a method defined to find the larger between two integers. This method, named `max`, has two `int` parameters, `num1` and `num2`, the larger of which is returned by the method. Figure 6.1 illustrates the components of this method.

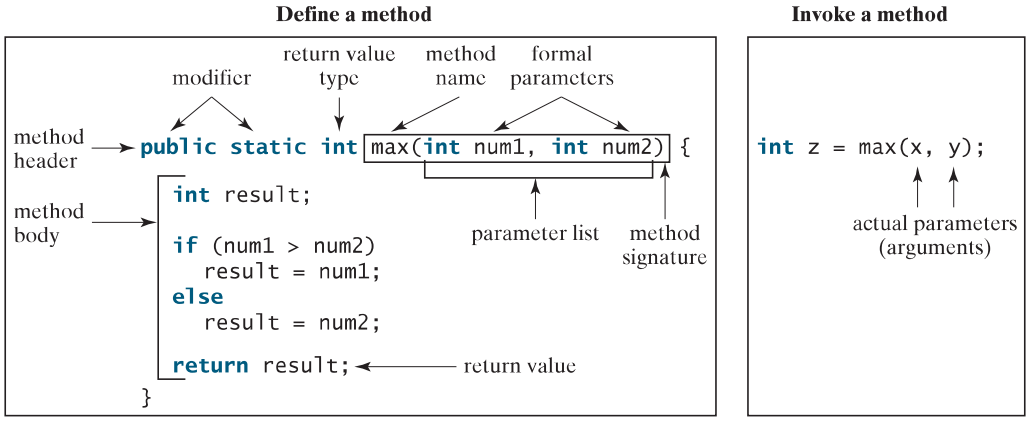

FIGURE 6.1 A method definition consists of a method header and a method body.

The *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The `static` modifier is used for all the methods in this chapter. The reason for using it will be discussed in Chapter 8, Objects and Classes.

method header
modifier

A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`. For example, the `returnValueType` is `void` in the `main` method, as well as in `System.exit`, and `System.out.println`. If a method returns a value, it is called a *value-returning method;* otherwise it is called a *void method.*

value-returning method
void method

The variables defined in the method header are known as *formal parameters* or simply *parameters.* A parameter is like a placeholder: when a method is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter or argument.* The *parameter list* refers to the method's type, order, and number of the parameters. The method name and the parameter list together constitute the *method signature.* Parameters are optional; that is, a method may contain no parameters. For example, the `Math.random()` method has no parameters.

formal parameter
parameter
actual parameter
argument
parameter list
method signature

The method body contains a collection of statements that implement the method. The method body of the `max` method uses an `if` statement to determine which number is larger and return the value of that number. In order for a value-returning method to return a result, a return statement using the keyword `return` is *required.* The method terminates when a return statement is executed.

Note

Some programming languages refer to methods as *procedures* and *functions.* In those languages, a value-returning method is called a *function* and a void method is called a *procedure.*

Caution

In the method header, you need to declare each parameter separately. For instance, `max(int num1, int num2)` is correct, but `max(int num1, num2)` is wrong.

define vs. declare

Note

We say "*define* a method" and "*declare* a variable." We are making a subtle distinction here. A definition defines what the defined item is, but a declaration usually involves allocating memory to store data for the declared item.

6.3 Calling a Method

Calling a method executes the code in the method.

In a method definition, you define what the method is to do. To execute the method, you have to *call* or *invoke* it. There are two ways to call a method, depending on whether the method returns a value or not.

If a method returns a value, a call to the method is usually treated as a value. For example,

```
int larger = max(3, 4);
```

calls `max(3, 4)` and assigns the result of the method to the variable `larger`. Another example of a call that is treated as a value is

```
System.out.println(max(3, 4));
```

which prints the return value of the method call `max(3, 4)`.

If a method returns `void`, a call to the method must be a statement. For example, the method `println` returns `void`. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

Note

A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value. This is not often done, but it is permissible if the caller is not interested in the return value.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

Listing 6.1 shows a complete program that is used to test the `max` method.

VideoNote

Define/invoke `max` method

LISTING 6.1 TestMax.java

```
1  public class TestMax {
2    /** Main method */
3    public static void main(String[] args) {
4      int i = 5;
5      int j = 2;
6      int k = max(i, j);
7      System.out.println("The maximum of " + i +
8        " and " + j + " is " + k);
9    }
10
11   /** Return the max of two numbers */
12   public static int max(int num1, int num2) {
13     int result;
14
15     if (num1 > num2)
16       result = num1;
17     else
18       result = num2;
19
20     return result;
21   }
22 }
```

`main` method

invoke `max`

define method

```
The maximum of 5 and 2 is 5
```

	line#	i	j	k	num1	num2	result
	4	5					
	5		2				
Invoking max	12				5	2	
	13						undefined
	16						5
	6			5			

This program contains the `main` method and the `max` method. The `main` method is just like any other method except that it is invoked by the JVM to start the program.

`main` method

The `main` method's header is always the same. Like the one in this example, it includes the modifiers `public` and `static`, return value type `void`, method name `main`, and a parameter of the `String[]` type. `String[]` indicates that the parameter is an array of `String`, a subject addressed in Chapter 7.

The statements in `main` may invoke other methods that are defined in the class that contains the `main` method or in other classes. In this example, the `main` method invokes `max(i, j)`, which is defined in the same class with the `main` method.

When the `max` method is invoked (line 6), variable `i`'s value `5` is passed to `num1`, and variable `j`'s value `2` is passed to `num2` in the `max` method. The flow of control transfers to the `max` method, and the `max` method is executed. When the `return` statement in the `max` method is executed, the `max` method returns the control to its caller (in this case the caller is the `main` method). This process is illustrated in Figure 6.2.

`max` method

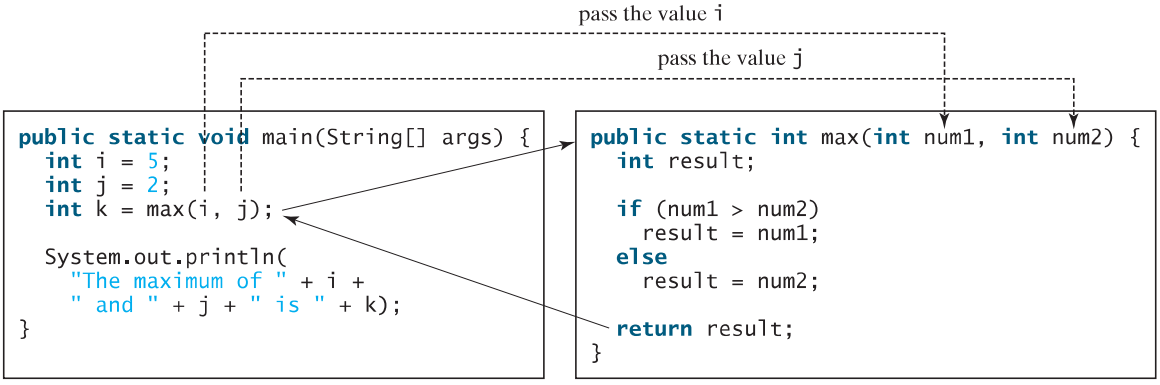

FIGURE 6.2 When the `max` method is invoked, the flow of control transfers to it. Once the `max` method is finished, it returns control back to the caller.

Caution

A `return` statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compile error because the Java compiler thinks that this method might not return a value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

To fix this problem, delete `if (n < 0)` in (a), so the compiler will see a `return` statement to be reached regardless of how the `if` statement is evaluated.

reusing method

Note

Methods enable code sharing and reuse. The `max` method can be invoked from any class, not just `TestMax`. If you create a new class, you can invoke the `max` method using `ClassName.methodName` (i.e., `TestMax.max`).

activation record

call stack

Each time a method is invoked, the system creates an *activation record* (also called an *activation frame*) that stores parameters and variables for the method and places the activation record in an area of memory known as a *call stack*. A call stack is also known as an *execution stack*, *runtime stack*, or *machine stack*, and it is often shortened to just "the stack." When a method calls another method, the caller's activation record is kept intact, and a new activation record is created for the new method called. When a method finishes its work and returns to its caller, its activation record is removed from the call stack.

A call stack stores the activation records in a last-in, first-out fashion: The activation record for the method that is invoked last is removed first from the stack. For example, suppose method `m1` calls method `m2`, and `m2` calls method `m3`. The runtime system pushes `m1`'s activation record into the stack, then `m2`'s, and then `m3`'s. After `m3` is finished, its activation record is removed from the stack. After `m2` is finished, its activation record is removed from the stack. After `m1` is finished, its activation record is removed from the stack.

Understanding call stacks helps you to comprehend how methods are invoked. The variables defined in the `main` method in Listing 6.1 are `i`, `j`, and `k`. The variables defined in the `max` method are `num1`, `num2`, and `result`. The variables `num1` and `num2` are defined in the method signature and are parameters of the `max` method. Their values are passed through method invocation. Figure 6.3 illustrates the activation records for method calls in the stack.

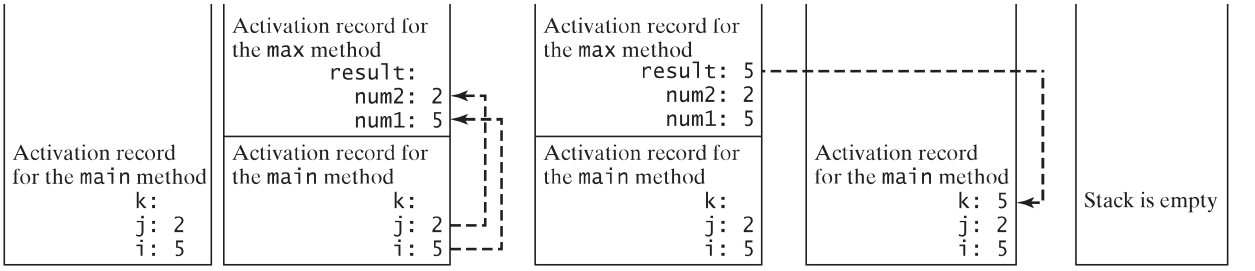

FIGURE 6.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.

6.4 **void** Method Example

A **void** *method does not return a value.*

The preceding section gives an example of a value-returning method. This section shows how to define and invoke a **void** method. Listing 6.2 gives a program that defines a method named **printGrade** and invokes it to print the grade for a given score.

VideoNote
Use `void` method

LISTING 6.2 TestVoidMethod.java

```
1  public class TestVoidMethod {
2    public static void main(String[] args) {
3      System.out.print("The grade is ");
4      printGrade(78.5);
5
6      System.out.print("The grade is ");
7      printGrade(59.5);
8    }
9
10   public static void printGrade(double score) {
11     if (score >= 90.0) {
12       System.out.println('A');
13     }
14     else if (score >= 80.0) {
15       System.out.println('B');
16     }
17     else if (score >= 70.0) {
18       System.out.println('C');
19     }
20     else if (score >= 60.0) {
21       System.out.println('D');
22     }
23     else {
24       System.out.println('F');
25     }
26   }
27 }
```

`main` method

invoke `printGrade`

`printGrade` method

```
The grade is C
The grade is F
```

The **printGrade** method is a **void** method because it does not return any value. A call to a **void** method must be a statement. Therefore, it is invoked as a statement in line 4 in the **main** method. Like any Java statement, it is terminated with a semicolon.

invoke `void` method

To see the differences between a void and value-returning method, let's redesign the **printGrade** method to return a value. The new method, which we call **getGrade**, returns the grade as shown in Listing 6.3.

void vs. value-returned

LISTING 6.3 TestReturnGradeMethod.java

```
1  public class TestReturnGradeMethod {
2    public static void main(String[] args) {
3      System.out.print("The grade is " + getGrade(78.5));
4      System.out.print("\nThe grade is " + getGrade(59.5));
5    }
6
```

`main` method

invoke `getGrade`

getGrade method

```
 7    public static char getGrade(double score) {
 8      if (score >= 90.0)
 9        return 'A';
10      else if (score >= 80.0)
11        return 'B';
12      else if (score >= 70.0)
13        return 'C';
14      else if (score >= 60.0)
15        return 'D';
16      else
17        return 'F';
18    }
19  }
```

```
The grade is C
The grade is F
```

The **getGrade** method defined in lines 7–18 returns a character grade based on the numeric score value. The caller invokes this method in lines 3–4.

The **getGrade** method can be invoked by a caller wherever a character may appear. The **printGrade** method does not return any value, so it must be invoked as a statement.

return in void method

Note

A **return** statement is not needed for a **void** method, but it can be used for terminating the method and returning to the method's caller. The syntax is simply

```
return;
```

This is not often done, but sometimes it is useful for circumventing the normal flow of control in a **void** method. For example, the following code has a return statement to terminate the method when the score is invalid.

```
public static void printGrade(double score) {
  if (score < 0 || score > 100) {
    System.out.println("Invalid score");
    return;
  }

  if (score >= 90.0) {
    System.out.println('A');
  }
  else if (score >= 80.0) {
    System.out.println('B');
  }
  else if (score >= 70.0) {
    System.out.println('C');
  }
  else if (score >= 60.0) {
    System.out.println('D');
  }
  else {
    System.out.println('F');
  }
}
```

6.1 What are the benefits of using a method?

6.2 How do you define a method? How do you invoke a method?

6.3 How do you simplify the `max` method in Listing 6.1 using the conditional operator?

6.4 True or false? A call to a method with a `void` return type is always a statement itself, but a call to a value-returning method cannot be a statement by itself.

6.5 What is the `return` type of a `main` method?

6.6 What would be wrong with not writing a `return` statement in a value-returning method? Can you have a `return` statement in a `void` method? Does the `return` statement in the following method cause syntax errors?

```
public static void xMethod(double x, double y) {
  System.out.println(x + y);
  return x + y;
}
```

6.7 Define the terms parameter, argument, and method signature.

6.8 Write method headers (not the bodies) for the following methods:

a. Return a sales commission, given the sales amount and the commission rate.

b. Display the calendar for a month, given the month and year.

c. Return a square root of a number.

d. Test whether a number is even, and returning `true` if it is.

e. Display a message a specified number of times.

f. Return the monthly payment, given the loan amount, number of years, and annual interest rate.

g. Return the corresponding uppercase letter, given a lowercase letter.

6.9 Identify and correct the errors in the following program:

```
1  public class Test {
2    public static method1(int n, m) {
3      n += m;
4      method2(3.4);
5    }
6
7    public static int method2(int n) {
8      if (n > 0) return 1;
9      else if (n == 0) return 0;
10     else if (n < 0) return -1;
11   }
12 }
```

6.10 Reformat the following program according to the programming style and documentation guidelines proposed in Section 1.9, Programming Style and Documentation. Use the next-line brace style.

```
public class Test {
  public static double method(double i, double j)
  {
  while (i < j) {
    j--;
  }

  return j;
  }
}
```

6.5 Passing Arguments by Values

The arguments are passed by value to parameters when invoking a method.

The power of a method is its ability to work with parameters. You can use `println` to print any string and `max` to find the maximum of any two `int` values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is known as *parameter order association*. For example, the following method prints a message `n` times:

parameter order association

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

You can use `nPrintln("Hello", 3)` to print `Hello` three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter `Hello` to the parameter `message`, passes `3` to `n`, and prints `Hello` three times. However, the statement `nPrintln(3, "Hello")` would be wrong. The data type of `3` does not match the data type for the first parameter, `message`, nor does the second argument, `Hello`, match the second parameter, `n`.

Caution

The arguments must match the parameters in *order*, *number*, and *compatible type*, as defined in the method signature. Compatible type means that you can pass an argument to a parameter without explicit casting, such as passing an `int` value argument to a `double` value parameter.

When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method. As shown in Listing 6.4, the value of `x` (`1`) is passed to the parameter `n` to invoke the `increment` method (line 5). The parameter `n` is incremented by `1` in the method (line 10), but `x` is not changed no matter what the method does.

pass-by-value

LISTING 6.4 Increment.java

```
1  public class Increment {
2    public static void main(String[] args) {
3      int x = 1;
4      System.out.println("Before the call, x is " + x);
5      increment(x);
6      System.out.println("After the call, x is " + x);
7    }
8
9    public static void increment(int n) {
10     n++;
11     System.out.println("n inside the method is " + n);
12   }
13 }
```

invoke increment

increment n

```
Before the call, x is 1
n inside the method is 2
After the call, x is 1
```

Listing 6.5 gives another program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The `swap` method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

LISTING 6.5 TestPassByValue.java

```
1  public class TestPassByValue {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare and initialize variables
5      int num1 = 1;
6      int num2 = 2;
7
8      System.out.println("Before invoking the swap method, num1 is " +
9        num1 + " and num2 is " + num2);
10
11     // Invoke the swap method to attempt to swap two variables
12     swap(num1, num2);                                    false swap
13
14     System.out.println("After invoking the swap method, num1 is " +
15       num1 + " and num2 is " + num2);
16   }
17
18   /** Swap two variables */
19   public static void swap(int n1, int n2) {
20     System.out.println("\tInside the swap method");
21     System.out.println("\t\tBefore swapping, n1 is " + n1
22       + " and n2 is " + n2);
23
24     // Swap n1 with n2
25     int temp = n1;
26     n1 = n2;
27     n2 = temp;
28
29     System.out.println("\t\tAfter swapping, n1 is " + n1
30       + " and n2 is " + n2);
31   }
32 }
```

```
Before invoking the swap method, num1 is 1 and num2 is 2
  Inside the swap method
    Before swapping, n1 is 1 and n2 is 2
    After swapping, n1 is 2 and n2 is 1
After invoking the swap method, num1 is 1 and num2 is 2
```

Before the `swap` method is invoked (line 12), `num1` is `1` and `num2` is `2`. After the `swap` method is invoked, `num1` is still `1` and `num2` is still `2`. Their values have not been swapped. As shown in Figure 6.4, the values of the arguments `num1` and `num2` are passed to `n1` and `n2`, but `n1` and `n2` have their own memory locations independent of `num1` and `num2`. Therefore, changes in `n1` and `n2` do not affect the contents of `num1` and `num2`.

Another twist is to change the parameter name `n1` in `swap` to `num1`. What effect does this have? No change occurs, because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the method with its own memory space. The variable is allocated when the method is invoked, and it disappears when the method is returned to its caller.

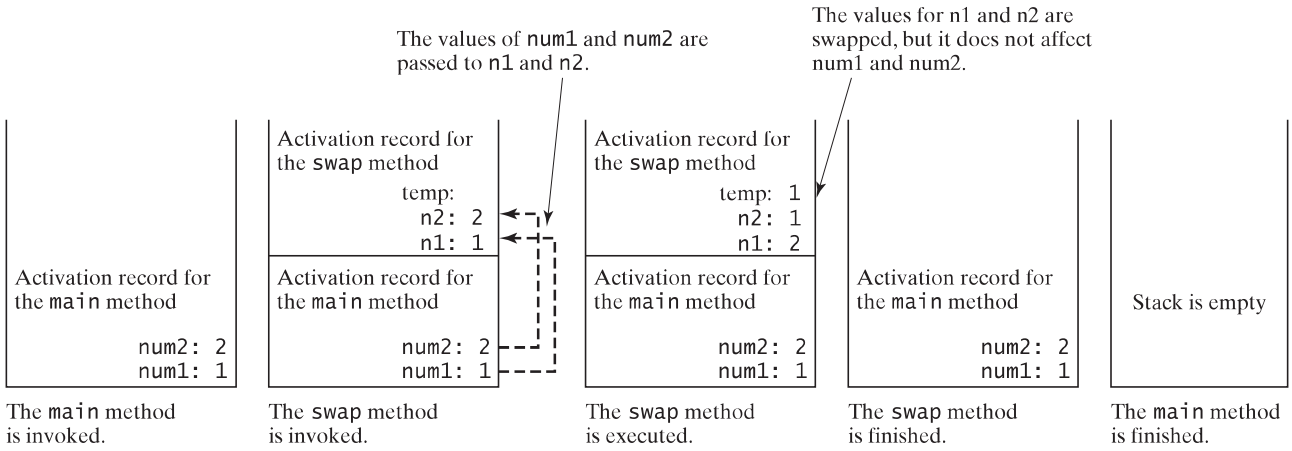

FIGURE 6.4 The values of the variables are passed to the method's parameters.

Note

For simplicity, Java programmers often say *passing x to y*, which actually means *passing the value of argument x to parameter y*.

6.11 How is an argument passed to a method? Can the argument have the same name as its parameter?

6.12 Identify and correct the errors in the following program:

```
1   public class Test {
2     public static void main(String[] args) {
3       nPrintln(5, "Welcome to Java!");
4     }
5
6     public static void nPrintln(String message, int n) {
7       int n = 1;
8       for (int i = 0; i < n; i++)
9         System.out.println(message);
10    }
11  }
```

6.13 What is pass-by-value? Show the result of the following programs.

```
public class Test {
  public static void main(String[] args) {
    int max = 0;
    max(1, 2, max);
    System.out.println(max);
  }

  public static void max(
      int value1, int value2, int max) {
    if (value1 > value2)
      max = value1;
    else
      max = value2;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    while (i <= 6) {
      method1(i, 2);
      i++;
    }
  }

  public static void method1(
      int i, int num) {
    for (int j = 1; j <= i; j++) {
      System.out.print(num + " ");
      num *= 2;
    }

    System.out.println();
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    // Initialize times
    int times = 3;
    System.out.println("Before the call,"
      + " variable times is " + times);

    // Invoke nPrintln and display times
    nPrintln("Welcome to Java!", times);
    System.out.println("After the call,"
      + " variable times is " + times);
  }

  // Print the message n times
  public static void nPrintln(
      String message, int n) {
    while (n > 0) {
      System.out.println("n = " + n);
      System.out.println(message);
      n--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 0;
    while (i <= 4) {
      method1(i);
      i++;
    }

    System.out.println("i is " + i);
  }

  public static void method1(int i) {
    do {
      if (i % 3 != 0)
        System.out.print(i + " ");
      i--;
    }
    while (i >= 1);

    System.out.println();
  }
}
```

(d)

6.14 For (a) in the preceding question, show the contents of the activation records in the call stack just before the method `max` is invoked, just as `max` is entered, just before `max` is returned, and right after `max` is returned.

6.6 Modularizing Code

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

Listing 5.9 gives a program that prompts the user to enter two integers and displays their greatest common divisor. You can rewrite the program using a method, as shown in Listing 6.6.

VideoNote
Modularize code

LISTING 6.6 GreatestCommonDivisorMethod.java

```
1  import java.util.Scanner;
2
3  public class GreatestCommonDivisorMethod {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter two integers
10     System.out.print("Enter first integer: ");
11     int n1 = input.nextInt();
12     System.out.print("Enter second integer: ");
13     int n2 = input.nextInt();
14
```

invoke gcd

compute gcd

return gcd

```
15      System.out.println("The greatest common divisor for " + n1 +
16        " and " + n2 + " is " + gcd(n1, n2));
17    }
18
19    /** Return the gcd of two integers */
20    public static int gcd(int n1, int n2) {
21      int gcd = 1; // Initial gcd is 1
22      int k = 2;  // Possible gcd
23
24      while (k <= n1 && k <= n2) {
25        if (n1 % k == 0 && n2 % k == 0)
26          gcd = k; // Update gcd
27        k++;
28      }
29
30      return gcd; // Return gcd
31    }
32  }
```

```
Enter first integer: 45 Enter
Enter second integer: 75 Enter
The greatest common divisor for 45 and 75 is 15
```

By encapsulating the code for obtaining the gcd in a method, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the main method. Thus, the logic becomes clear and the program is easier to read.
2. The errors on computing the gcd are confined in the **gcd** method, which narrows the scope of debugging.
3. The **gcd** method now can be reused by other programs.

Listing 6.7 applies the concept of code modularization to improve Listing 5.15, PrimeNumber.java.

LISTING 6.7 PrimeNumberMethod.java

invoke printPrimeNumbers

printPrimeNumbers method

invoke isPrime

```
1  public class PrimeNumberMethod {
2    public static void main(String[] args) {
3      System.out.println("The first 50 prime numbers are \n");
4      printPrimeNumbers(50);
5    }
6
7    public static void printPrimeNumbers(int numberOfPrimes) {
8      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
9      int count = 0; // Count the number of prime numbers
10     int number = 2; // A number to be tested for primeness
11
12     // Repeatedly find prime numbers
13     while (count < numberOfPrimes) {
14       // Print the prime number and increase the count
15       if (isPrime(number)) {
16         count++; // Increase the count
17
```

```
18          if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
19            // Print the number and advance to the new line
20            System.out.printf("%-5s\n", number);
21          }
22          else
23            System.out.printf("%-5s", number);
24        }
25
26        // Check whether the next number is prime
27        number++;
28      }
29    }
30
31    /** Check whether number is prime */
32    public static boolean isPrime(int number) {
33      for (int divisor = 2; divisor <= number / 2; divisor++) {
34        if (number % divisor == 0) { // If true, number is not prime
35          return false; // Number is not a prime
36        }
37      }
38
39      return true; // Number is prime
40    }
41  }
```

isPrime method

```
The first 50 prime numbers are

2    3    5    7    11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
73   79   83   89   97   101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229
```

We divided a large problem into two subproblems: determining whether a number is a prime and printing the prime numbers. As a result, the new program is easier to read and easier to debug. Moreover, the methods **printPrimeNumbers** and **isPrime** can be reused by other programs.

6.7 Case Study: Converting Hexadecimals to Decimals

This section presents a program that converts a hexadecimal number into a decimal number.

Listing 5.11, Dec2Hex.java, gives a program that converts a decimal to a hexadecimal. How would you convert a hex number into a decimal?

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number **AB8C** is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the `i`'s position, and then add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0$$
$$= (\ldots((h_n \times 16 + h_n - 1) \times 16 + h_n - 2) \times 16 + \ldots + h_1) \times 16 + h_0$$

This observation, known as the Horner's algorithm, leads to the following efficient code for converting a hex string to a decimal number:

```
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
  char hexChar = hex.charAt(i);
  decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Here is a trace of the algorithm for hex number `AB8C`:

	i	hexChar	hexCharToDecimal (hexChar)	decimalValue
before the loop				0
after the 1st iteration	0	A	10	10
after the 2nd iteration	1	B	11	10 * 16 + 11
after the 3rd iteration	2	8	8	(10 * 16 + 11) * 16 + 8
after the 4th iteration	3	C	12	((10 * 16 + 11) * 16 + 8) * 16 + 12

Listing 6.8 gives the complete program.

LISTING 6.8 Hex2Dec.java

```
1  import java.util.Scanner;
2
3  public class Hex2Dec {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a hex number: ");
11     String hex = input.nextLine();
12
13     System.out.println("The decimal value for hex number "
14       + hex + " is " + hexToDecimal(hex.toUpperCase()));
15   }
16
17   public static int hexToDecimal(String hex) {
18     int decimalValue = 0;
19     for (int i = 0; i < hex.length(); i++) {
20       char hexChar = hex.charAt(i);
21       decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
```

input string

hex to decimal

```
22      }
23
24      return decimalValue;
25    }
26
27    public static int hexCharToDecimal(char ch) {
28      if (ch >= 'A' && ch <= 'F')
29        return 10 + ch - 'A';
30      else // ch is '0', '1', ..., or '9'
31        return ch - '0';
32    }
33  }
```

hex char to decimal

check uppercase

```
Enter a hex number: AB8C Enter
The decimal value for hex number AB8C is 43916
```

```
Enter a hex number: af71 Enter
The decimal value for hex number af71 is 44913
```

The program reads a string from the console (line 11), and invokes the `hexToDecimal` method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the `hexToDecimal` method.

The `hexToDecimal` method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking `hex.length()` in line 19.

The `hexCharToDecimal` method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, `'5' - '0'` is `5`.

6.8 Overloading Methods

Overloading methods enables you to define the methods with the same name as long as their signatures are different.

The `max` method that was used earlier works only with the `int` data type. But what if you need to determine which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

If you call `max` with `int` parameters, the `max` method that expects `int` parameters will be invoked; if you call `max` with `double` parameters, the `max` method that expects `double` parameters will be invoked. This is referred to as *method overloading*; that is, two methods have the same name but different parameter lists within one class. The Java compiler determines which method to use based on the method signature.

method overloading

Listing 6.9 is a program that creates three methods. The first finds the maximum integer, the second finds the maximum double, and the third finds the maximum among three double values. All three methods are named `max`.

LISTING 6.9 TestMethodOverloading.java

```
1  public class TestMethodOverloading {
2    /** Main method */
3    public static void main(String[] args) {
4      // Invoke the max method with int parameters
5      System.out.println("The maximum of 3 and 4 is "
6        + max(3, 4));
7
8      // Invoke the max method with the double parameters
9      System.out.println("The maximum of 3.0 and 5.4 is "
10       + max(3.0, 5.4));
11
12     // Invoke the max method with three double parameters
13     System.out.println("The maximum of 3.0, 5.4, and 10.14 is "
14       + max(3.0, 5.4, 10.14));
15   }
16
17   /** Return the max of two int values */
18   public static int max(int num1, int num2) {
19     if (num1 > num2)
20       return num1;
21     else
22       return num2;
23   }
24
25   /** Find the max of two double values */
26   public static double max(double num1, double num2) {
27     if (num1 > num2)
28       return num1;
29     else
30       return num2;
31   }
32
33   /** Return the max of three double values */
34   public static double max(double num1, double num2, double num3) {
35     return max(max(num1, num2), num3);
36   }
37 }
```

overloaded max

overloaded max

overloaded max

```
The maximum of 3 and 4 is 4
The maximum of 3.0 and 5.4 is 5.4
The maximum of 3.0, 5.4, and 10.14 is 10.14
```

When calling `max(3, 4)` (line 6), the `max` method for finding the maximum of two integers is invoked. When calling `max(3.0, 5.4)` (line 10), the `max` method for finding the maximum of two doubles is invoked. When calling `max(3.0, 5.4, 10.14)` (line 14), the `max` method for finding the maximum of three double values is invoked.

Can you invoke the `max` method with an `int` value and a `double` value, such as `max(2, 2.5)`? If so, which of the `max` methods is invoked? The answer to the first question is yes. The answer to the second question is that the `max` method for finding the maximum of two `double` values is invoked. The argument value `2` is automatically converted into a `double` value and passed to this method.

You may be wondering why the method `max(double, double)` is not invoked for the call `max(3, 4)`. Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4)`. The Java compiler finds the method that best matches a method invocation. Since the method `max(int, int)` is a better matches for `max(3, 4)` than `max(double, double)`, `max(int, int)` is used to invoke `max(3, 4)`.

Tip

Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.

Note

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.

Note

Sometimes there are two or more possible matches for the invocation of a method, but the compiler cannot determine the best match. This is referred to as *ambiguous invocation*. Ambiguous invocation causes a compile error. Consider the following code:

ambiguous invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Both `max(int, double)` and `max(double, int)` are possible candidates to match `max(1, 2)`. Because neither is better than the other, the invocation is ambiguous, resulting in a compile error.

6.15 What is method overloading? Is it permissible to define two methods that have the same name but different parameter types? Is it permissible to define two methods in a class that have identical method names and parameter lists but different return value types or different modifiers?

6.16 What is wrong in the following program?

```
public class Test {
  public static void method(int x) {
  }

  public static int method(int y) {
```

```
    return y;
  }
}
```

6.17 Given two method definitions,

```
public static double m(double x, double y)

public static double m(int x, double y)
```

tell which of the two methods is invoked for:

a. `double z = m(4, 5);`

b. `double z = m(4, 5.4);`

c. `double z = m(4.5, 5.4);`

6.9 The Scope of Variables

The scope of a variable is the part of the program where the variable can be referenced.

scope of variables
local variable

Section 2.5 introduced the scope of a variable. This section discusses the scope of variables in detail. A variable defined inside a method is referred to as a *local variable*. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial-action part of a `for`-loop header has its scope in the entire loop. However, a variable declared inside a `for`-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 6.5.

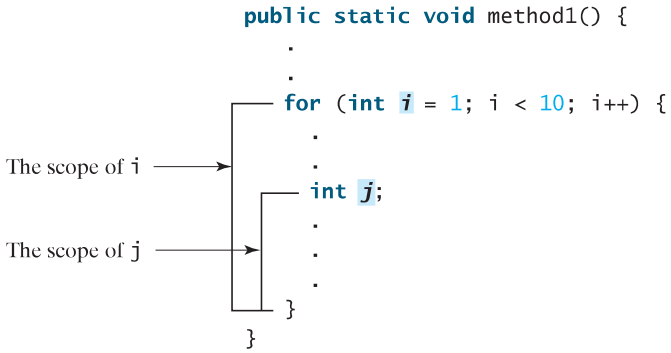

FIGURE 6.5 A variable declared in the initial action part of a `for`-loop header has its scope in the entire loop.

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in Figure 6.6.

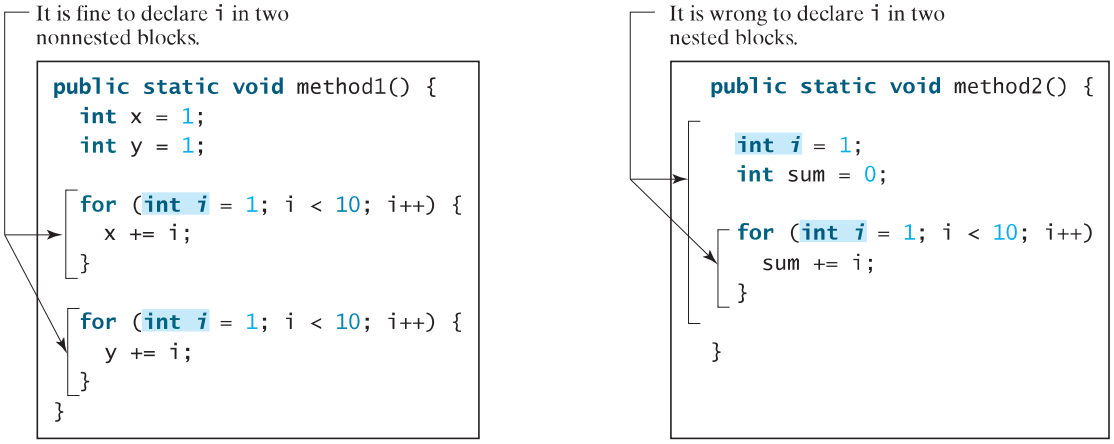

FIGURE 6.6 A variable can be declared multiple times in nonnested blocks, but only once in nested blocks.

Caution

Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```
for (int i = 0; i < 10; i++) {
}

System.out.println(i);
```

The last statement would cause a syntax error, because variable `i` is not defined outside of the `for` loop.

6.18 What is a local variable?

6.19 What is the scope of a local variable?

6.10 Case Study: Generating Random Characters

A character is coded using an integer. Generating a random character is to generate an integer.

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example of generating random characters.

As introduced in Section 4.3, every character has a unique Unicode between `0` and `FFFF` in hexadecimal (`65535` in decimal). To generate a random character is to generate a random integer between `0` and `65535` using the following expression (note that since `0 <= Math.random() < 1.0`, you have to add `1` to `65535`):

```
(int)(Math.random() * (65535 + 1))
```

Now let's consider how to generate a random lowercase letter. The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `a`, then that for `b`, `c`, . . . , and `z`. The Unicode for `a` is

```
(int)'a'
```

Thus, a random integer between `(int)'a'` and `(int)'z'` is

```
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

As discussed in Section 4.3.3, all numeric operators can be applied to the `char` operands. The `char` operand is cast into a number if the other operand is a number or a character. Therefore, the preceding expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

and a random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

Hence, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

This is a simple but useful discovery. Listing 6.10 defines a class named `RandomCharacter` with five overloaded methods to get a certain type of character randomly. You can use these methods in your future projects.

LISTING 6.10 RandomCharacter.java

```
1  public class RandomCharacter {
2    /** Generate a random character between ch1 and ch2 */
3    public static char getRandomCharacter(char ch1, char ch2) {
4      return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
5    }
6
7    /** Generate a random lowercase letter */
8    public static char getRandomLowerCaseLetter() {
9      return getRandomCharacter('a', 'z');
10   }
11
12   /** Generate a random uppercase letter */
13   public static char getRandomUpperCaseLetter() {
14     return getRandomCharacter('A', 'Z');
15   }
16
17   /** Generate a random digit character */
18   public static char getRandomDigitCharacter() {
19     return getRandomCharacter('0', '9');
20   }
21
22   /** Generate a random character */
23   public static char getRandomCharacter() {
24     return getRandomCharacter('\u0000', '\uFFFF');
25   }
26 }
```

getRandomCharacter

getRandomLowerCaseLetter()

getRandomUpperCaseLetter()

getRandomDigitCharacter()

getRandomCharacter()

Listing 6.11 gives a test program that displays 175 random lowercase letters.

LISTING 6.11 TestRandomCharacter.java

```
1  public class TestRandomCharacter {
2    /** Main method */
3    public static void main(String[] args) {
4      final int NUMBER_OF_CHARS = 175;
5      final int CHARS_PER_LINE = 25;
6
7      // Print random characters between 'a' and 'z', 25 chars per line
8      for (int i = 0; i < NUMBER_OF_CHARS; i++) {
```

constants

```
 9         char ch = RandomCharacter.getRandomLowerCaseLetter();
10         if ((i + 1) % CHARS_PER_LINE == 0)
11           System.out.println(ch);
12         else
13           System.out.print(ch);
14       }
15     }
16   }
```

lower-case letter

```
gmjsohezfkgtazqgmswfclrao
pnrunulnwmaztlfjedmpchcif
lalqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni
```

Line 9 invokes `getRandomLowerCaseLetter()` defined in the `RandomCharacter` class. Note that `getRandomLowerCaseLetter()` does not have any parameters, but you still have to use the parentheses when defining and invoking the method.

parentheses required

6.11 Method Abstraction and Stepwise Refinement

The key to developing software is to apply the concept of abstraction.

You will learn many levels of abstraction from this book. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a "black box," as shown in Figure 6.7.

VideoNote
Stepwise refinement

method abstraction

information hiding

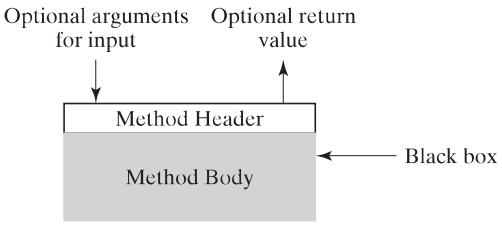

FIGURE 6.7 The method body can be thought of as a black box that contains the detailed implementation for the method.

You have already used the `System.out.print` method to display a string and the `max` method to find the maximum number. You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

divide and conquer

stepwise refinement

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, then displays the entire calendar for the month, as shown in the following sample run.

```
Enter full year (e.g., 2012): 2012 ↵Enter

Enter month as number between 1 and 12: 3 ↵Enter

         March 2012
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
                   1   2   3
   4   5   6   7   8   9  10
  11  12  13  14  15  16  17
  18  19  20  21  22  23  24
  25  26  27  28  29  30
```

Let us use this example to demonstrate the divide-and-conquer approach.

6.11.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: get input from the user and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see Figure 6.8a).

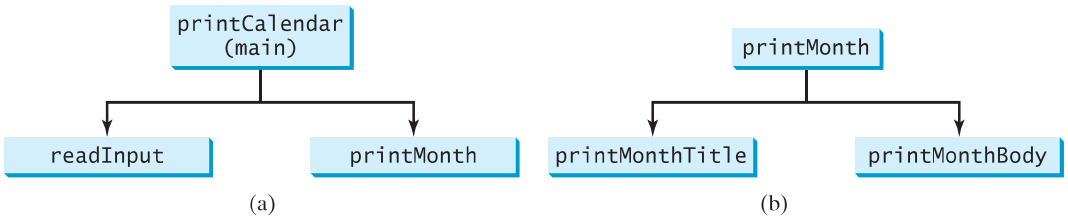

FIGURE 6.8 The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

You can use `Scanner` to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: print the month title and print the month body, as shown in Figure 6.8b. The month title consists of three lines: month and year, a dashed line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `getMonthName` (see Figure 6.9a).

In order to print the month body, you need to know which day of the week is the first day of the month (`getStartDay`) and how many days the month has (`getNumberOfDaysInMonth`),

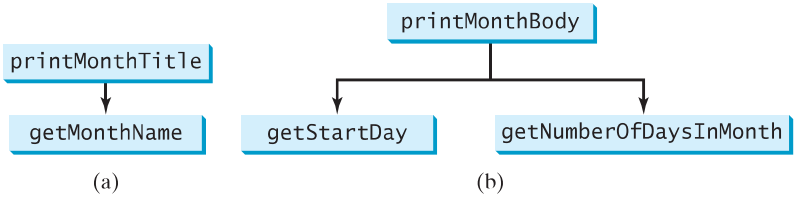

FIGURE 6.9 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

as shown in Figure 6.9b. For example, December 2013 has 31 days, and December 1, 2013, is a Sunday.

How would you get the start day for the first date in a month? There are several ways to do so. For now, we'll use an alternative approach. Assume you know that the start day for January 1, 1800, was a Wednesday (`START_DAY_FOR_JAN_1_1800 = 3`). You could compute the total number of days (`totalNumberOfDays`) between January 1, 1800, and the first date of the calendar month. The start day for the calendar month is `(totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7`, since every week has seven days. Thus, the `getStartDay` problem can be further refined as `getTotalNumberOfDays`, as shown in Figure 6.10a.

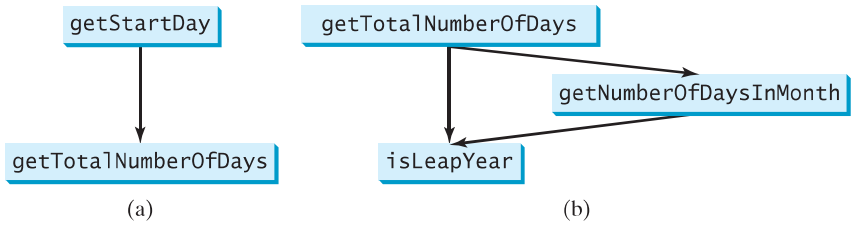

FIGURE 6.10 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. Thus, `getTotalNumberOfDays` can be further refined into two subproblems: `isLeapYear` and `getNumberOfDaysInMonth`, as shown in Figure 6.10b. The complete structure chart is shown in Figure 6.11.

6.11.2 Top-Down and/or Bottom-Up Implementation

Now we turn our attention to implementation. In general, a subproblem corresponds to a method in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as methods and which to combine with other methods. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem `readInput` can be simply implemented in the `main` method.

top-down approach
stub

You can use either a "top-down" or a "bottom-up" approach. The top-down approach implements one method in the structure chart at a time from the top to the bottom. *Stubs*—a simple but incomplete version of a method—can be used for the methods waiting to be implemented. The use of stubs enables you to quickly build the framework of the program. Implement the `main` method first, and then use a stub for the `printMonth` method. For example,

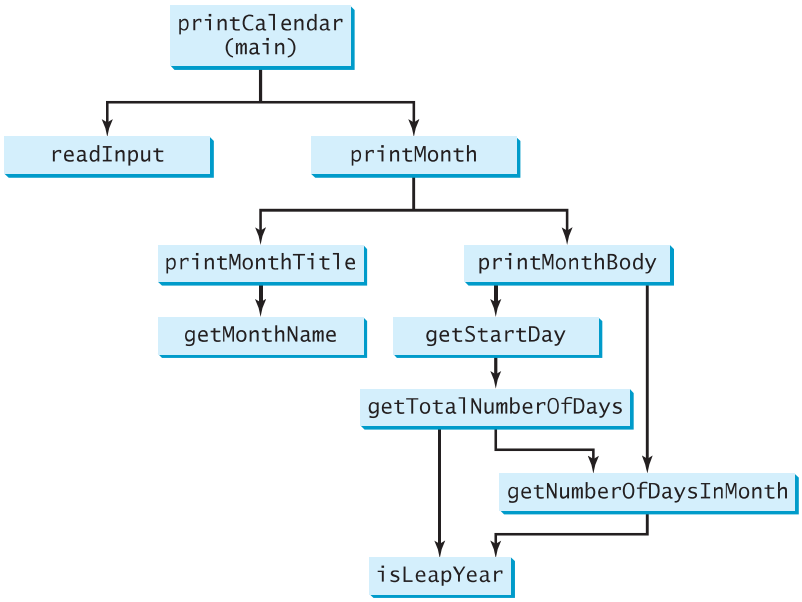

FIGURE 6.11 The structure chart shows the hierarchical relationship of the subproblems in the program.

let `printMonth` display the year and the month in the stub. Thus, your program may begin like this:

```
public class PrintCalendar {
  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter year
    System.out.print("Enter full year (e.g., 2012): ");
    int year = input.nextInt();

    // Prompt the user to enter month
    System.out.print("Enter month as a number between 1 and 12: ");
    int month = input.nextInt();

    // Print calendar for the month of the year
    printMonth(year, month);
  }

  /** A stub for printMonth may look like this */
  public static void printMonth(int year, int month){
    System.out.print(month + " " + year);
  }

  /** A stub for printMonthTitle may look like this */
  public static void printMonthTitle(int year, int month){
  }

  /** A stub for getMonthBody may look like this */
  public static void printMonthBody(int year, int month){
  }
```

```
  /** A stub for getMonthName may look like this */
  public static String getMonthName(int month) {
    return "January"; // A dummy value
  }

  /** A stub for getStartDay may look like this */
  public static int getStartDay(int year, int month) {
    return 1; // A dummy value
  }

  /** A stub for getTotalNumberOfDays may look like this */
  public static int getTotalNumberOfDays(int year, int month) {
    return 10000; // A dummy value
  }

  /** A stub for getNumberOfDaysInMonth may look like this */
  public static int getNumberOfDaysInMonth(int year, int month) {
    return 31; // A dummy value
  }

  /** A stub for isLeapYear may look like this */
  public static Boolean isLeapYear(int year) {
    return true; // A dummy value
  }
}
```

Compile and test the program, and fix any errors. You can now implement the `printMonth` method. For methods invoked from the `printMonth` method, you can again use stubs.

bottom-up approach
driver

The bottom-up approach implements one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program, known as the *driver*, to test it. The top-down and bottom-up approaches are equally good: Both approaches implement methods incrementally, help to isolate programming errors, and make debugging easy. They can be used together.

6.11.3 Implementation Details

The `isLeapYear(int year)` method can be implemented using the following code from Section 3.11:

```
return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
```

Use the following facts to implement `getTotalNumberOfDaysInMonth(int year, int month)`:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year and 29 days during a leap year. A regular year, therefore, has 365 days, a leap year 366 days.

To implement `getTotalNumberOfDays(int year, int month)`, you need to compute the total number of days (`totalNumberOfDays`) between January 1, 1800, and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is `totalNumberOfDays`.

To print a body, first pad some space before the start day and then print the lines for every week.

The complete program is given in Listing 6.12.

LISTING 6.12 PrintCalendar.java

```
1  import java.util.Scanner;
2
3  public class PrintCalendar {
4    /** Main method */
5    public static void main(String[] args) {
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter year
9      System.out.print("Enter full year (e.g., 2012): ");
10     int year = input.nextInt();
11
12     // Prompt the user to enter month
13     System.out.print("Enter month as a number between 1 and 12: ");
14     int month = input.nextInt();
15
16     // Print calendar for the month of the year
17     printMonth(year, month);
18   }
19
20   /** Print the calendar for a month in a year */
21   public static void printMonth(int year, int month) {
22     // Print the headings of the calendar
23     printMonthTitle(year, month);
24
25     // Print the body of the calendar
26     printMonthBody(year, month);
27   }
28
29   /** Print the month title, e.g., March 2012 */
30   public static void printMonthTitle(int year, int month) {
31     System.out.println("         " + getMonthName(month)
32       + " " + year);
33     System.out.println("-----------------------------");
34     System.out.println(" Sun Mon Tue Wed Thu Fri Sat");
35   }
36
37   /** Get the English name for the month */
38   public static String getMonthName(int month) {
39     String monthName = "";
40     switch (month) {
41       case 1: monthName = "January"; break;
42       case 2: monthName = "February"; break;
43       case 3: monthName = "March"; break;
44       case 4: monthName = "April"; break;
45       case 5: monthName = "May"; break;
46       case 6: monthName = "June"; break;
47       case 7: monthName = "July"; break;
48       case 8: monthName = "August"; break;
49       case 9: monthName = "September"; break;
50       case 10: monthName = "October"; break;
51       case 11: monthName = "November"; break;
52       case 12: monthName = "December";
53     }
54
55     return monthName;
56   }
57
58   /** Print month body */
```

printMonth

printMonthTitle

getMonthName

```
59    public static void printMonthBody(int year, int month) {              printMonthBody
60      // Get start day of the week for the first date in the month
61      int startDay = getStartDay(year, month)
62
63      // Get number of days in the month
64      int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);
65
66      // Pad space before the first day of the month
67      int i = 0;
68      for (i = 0; i < startDay; i++)
69        System.out.print("    ");
70
71      for (i = 1; i <= numberOfDaysInMonth; i++) {
72        System.out.printf("%4d", i);
73
74        if ((i + startDay) % 7 == 0)
75          System.out.println();
76      }
77
78      System.out.println();
79    }
80
81    /** Get the start day of month/1/year */
82    public static int getStartDay(int year, int month) {                  getStartDay
83      final int START_DAY_FOR_JAN_1_1800 = 3;
84      // Get total number of days from 1/1/1800 to month/1/year
85      int totalNumberOfDays = getTotalNumberOfDays(year, month);
86
87      // Return the start day for month/1/year
88      return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;
89    }
90
91    /** Get the total number of days since January 1, 1800 */
92    public static int getTotalNumberOfDays(int year, int month) {         getTotalNumberOfDays
93      int total = 0;
94
95      // Get the total days from 1800 to 1/1/year
96      for (int i = 1800; i < year; i++)
97        if (isLeapYear(i))
98          total = total + 366;
99        else
100         total = total + 365;
101
102     // Add days from Jan to the month prior to the calendar month
103     for (int i = 1; i < month; i++)
104       total = total + getNumberOfDaysInMonth(year, i);
105
106     return total;
107   }
108
109   /** Get the number of days in a month */
110   public static int getNumberOfDaysInMonth(int year, int month) {       getNumberOfDaysInMonth
111     if (month == 1 || month == 3 || month == 5 || month == 7 ||
112       month == 8 || month == 10 || month == 12)
113       return 31;
114
115     if (month == 4 || month == 6 || month == 9 || month == 11)
116       return 30;
117
118     if (month == 2) return isLeapYear(year) ? 29 : 28;
```

```
119
120       return 0; // If month is incorrect
121     }
122
123     /** Determine if it is a leap year */
124     public static boolean isLeapYear(int year) {
125       return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
126     }
127   }
```

isLeapYear

The program does not validate user input. For instance, if the user enters either a month not in the range between `1` and `12` or a year before `1800`, the program displays an erroneous calendar. To avoid this error, add an `if` statement to check the input before printing the calendar.

This program prints calendars for a month but could easily be modified to print calendars for a whole year. Although it can print months only after January `1800`, it could be modified to print months before `1800`.

6.11.4 Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

Simpler Program

The print calendar program is long. Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods. This simplifies the program and makes the whole program easier to read and understand.

Reusing Methods

Stepwise refinement promotes code reuse within a program. The `isLeapYear` method is defined once and invoked from the `getTotalNumberOfDays` and `getNumberOfDayInMonth` methods. This reduces redundant code.

Easier Developing, Debugging, and Testing

Since each subproblem is solved in a method, a method can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier.

When implementing a large program, use the top-down and/or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

incremental development and testing

Better Facilitating Teamwork

When a large problem is divided into subprograms, subproblems can be assigned to different programmers. This makes it easier for programmers to work in teams.

Key Terms

actual parameter 205
ambiguous invocation 221
argument 205
divide and conquer 225
formal parameter (i.e., parameter) 205
information hiding 225
method 204
method abstraction 225
method overloading 219
method signature 205
modifier 205
parameter 205
pass-by-value 212
scope of a variable 222
stepwise refinement 225
stub 227

CHAPTER SUMMARY

1. Making programs modular and reusable is one of the central goals in software engineering. Java provides many powerful constructs that help to achieve this goal. *Methods* are one such construct.

2. The method header specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The `static` modifier is used for all the methods in this chapter.

3. A method may return a value. The `returnValueType` is the data type of the value the method returns. If the method does not return a value, the `returnValueType` is the keyword `void`.

4. The *parameter list* refers to the type, order, and number of a method's parameters. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method doesn't need to contain any parameters.

5. A return statement can also be used in a `void` method for terminating the method and returning to the method's caller. This is useful occasionally for circumventing the normal flow of control in a method.

6. The arguments that are passed to a method should have the same number, type, and order as the parameters in the method signature.

7. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

8. A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value.

9. A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.

10. A variable declared in a method is called a local variable. The *scope of a local variable* starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and initialized before it is used.

11. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or *encapsulation*.

12. Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes method reusability.

13. When implementing a large program, use the top-down and/or bottom-up coding approach. Do not write the entire program at once. This approach may seem to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.