Chapter 6

Methods

Objectives

- To define methods with formal parameters
- To invoke methods with actual parameters (i.e., arguments)
- To define methods with a return value
- To define methods without a return value
- To pass arguments by value
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain
- To use method overloading and understand ambiguous overloading
- To determine the scope of variables
- To apply the concept of method abstraction in software development
- To design and implement methods using stepwise refinement

Methods: Introduction

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

```
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);
```


Methods: Introduction (cont'd)

- The preceding code can be simplified.
- 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**.
- 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.

```
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 }
```

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;
}
```

- 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.

A Method to find the larger between two integers

- 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.

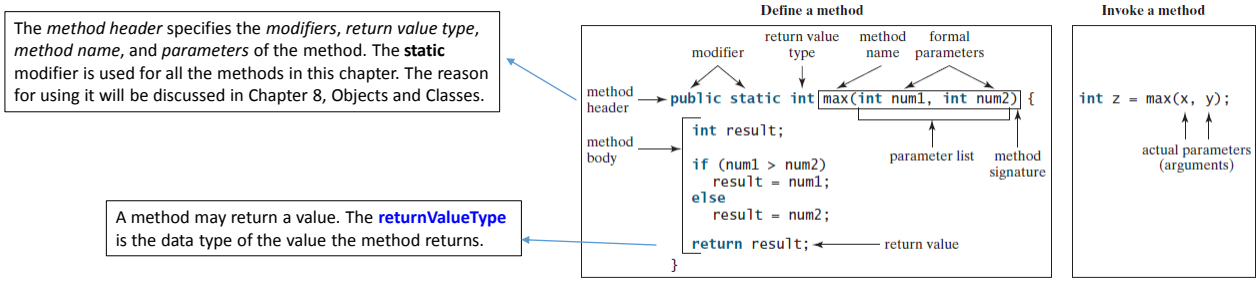
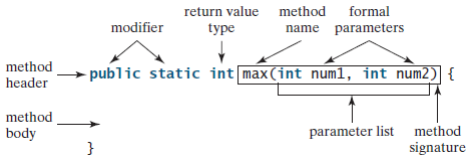


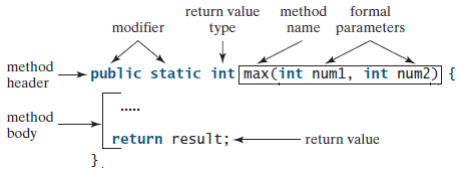
FIGURE 6.1 A method definition consists of a method header and a method body.

Method: parameters



- The *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the 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.
- 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.

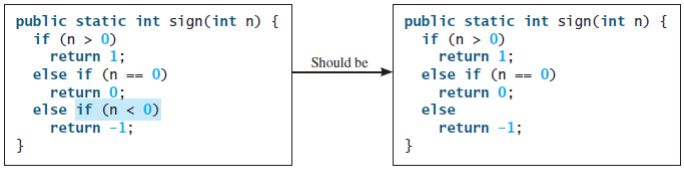
Method: returnValueType



- 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.
- If a method returns a value, it is called a *value-returning method;* otherwise it is called a *void method*.
- 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.

Method: returnValueType (cont'd)

- 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.



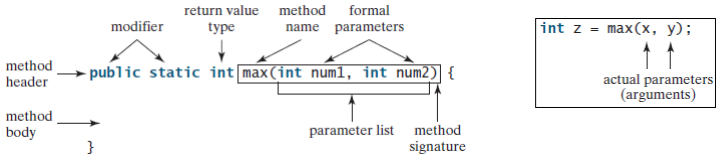
- To fix this problem, delete **if (n < 0)** in (a), so the compiler will see a **return** statement to be reached regardless of how the **if** statement is evaluated.

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)**.

Calling a Method (cont'd)

- 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*.



- 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 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 A complete program to test the **max** method.

- 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.
- The **main** method's header is always the same. 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.

LISTING 6.1 TestMax.java

```
1  public class TestMax {                                    Define/invoke max method
2    /** Main method */
3    public static void main(String[] args) {                main method
4      int i = 5;
5      int j = 2;
6      int k = max(i, j);                                    invoke max
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) {             define method
13     int result;
14
15     if (num1 > num2)
16       result = num1;
17     else
18       result = num2;
19
20     return result;
21   }
22 }
```

Calling a Method: The flow of control

- 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).

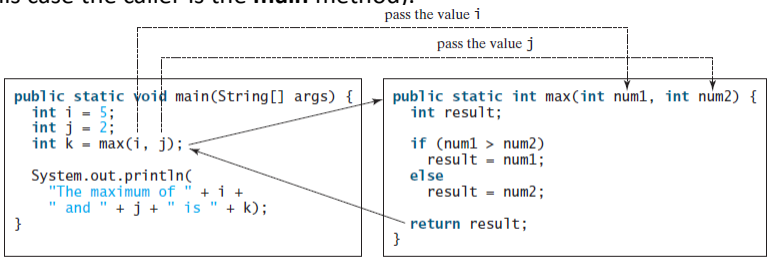


FIGURE 6.2 When the `max` method is invoked, the flow of control transfers to it. Once the `max` method is finished, it returns control back to the caller.

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.

Call Stack for Listing 6.1

- Understanding call stacks helps you to comprehend how methods are invoked.
- Figure 6.3 illustrates the activation records for method calls in the stack.

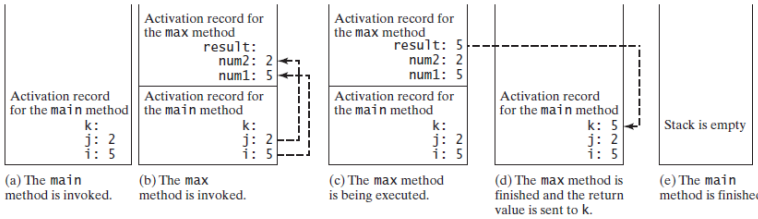


FIGURE 6.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.

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 }
```


void Method Example

- *A* **void** *method does not return a value.*
- Listing 6.2 gives a program that defines a method named **printGrade** and invokes it to print the grade for a given score.
- 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.
- 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;**

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 }
```

Differences between a void and value-returning 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.
- 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.

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
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 }
```


Passing Arguments by Values

- 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.

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 }
```

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

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.

Modularizing Code: an example

LISTING 5.9 GreatestCommonDivisor.java

```
1  import java.util.Scanner;
2
3  public class GreatestCommonDivisor {
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
15     int gcd = 1; // Initial gcd is 1
16     int k = 2; // Possible gcd
17     while (k <= n1 && k <= n2) {
18       if (n1 % k == 0 && n2 % k == 0)
19         gcd = k; // Update gcd
20       k++;
21     }
22
23     System.out.println("The greatest common divisor for " + n1 +
24       " and " + n2 + " is " + gcd);
25   }
26 }
```

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.

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
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 }
```

Modularizing Code: advantages

- 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.

Modularizing Code: another example

LISTING 5.15 PrimeNumber.java

```
1  public class PrimeNumber {
2    public static void main(String[] args) {
3      final int NUMBER_OF_PRIMES = 50; // Number of primes to display
4      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
5      int count = 0; // Count the number of prime numbers
6      int number = 2; // A number to be tested for primeness
7
8      System.out.println("The first 50 prime numbers are \n");
9
10     // Repeatedly find prime numbers
11     while (count < NUMBER_OF_PRIMES) {
12       // Assume the number is prime
13       boolean isPrime = true; // Is the current number prime?
14
15       // Test whether number is prime
16       for (int divisor = 2; divisor <= number / 2; divisor++) {
17         if (number % divisor == 0) { // If true, number is not prime
18           isPrime = false; // Set isPrime to false
19           break; // Exit the for loop
20         }
21       }
22
23       // Display the prime number and increase the count
24       if (isPrime) {
25         count++; // Increase the count
26
27         if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28           // Display the number and advance to the new line
29           System.out.println(number);
30         }
31         else
32           System.out.print(number + " ");
33       }
34
35       // Check if the next number is prime
36       number++;
37     }
38   }
39 }
```

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

LISTING 6.7 PrimeNumberMethod.java

```
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 }
```

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

Overloading Methods: *Ambiguous invocation*

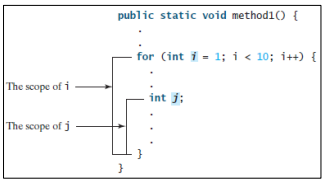
- Sometimes there are two or more possible matches for the invocation of a method, but the compiler cannot determine the best match. This is referred to as *ambiguous invocation*.
- Ambiguous invocation causes a compile error. Consider the following code:

```
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.

The Scope of Variables

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



The Scope of Variables (cont'd)

- 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.

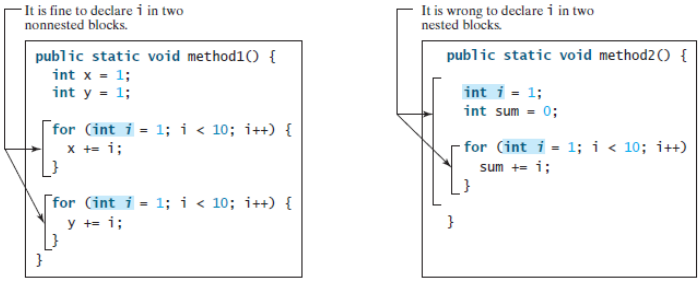


FIGURE 6.6 A variable can be declared multiple times in nonnested blocks, but only once in nested blocks.

The Scope of Variables (cont'd)

- 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.

Method Abstraction

- *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.

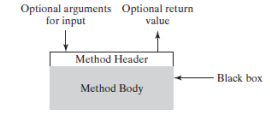


FIGURE 6.7 The method body can be thought of as a black box that contains the detailed implementation for the method.

- You have already used the **System.out.print** method to display a string. You know how to write the code to invoke this method in your program, but as a user, you are not required to know how it is implemented.

Stepwise Refinement

- 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.
- 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 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
```


Top-Down Design


```
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 the calendar example to demonstrate the divide-and-conquer approach.
- Beginning programmers often start by trying to work out the solution to every detail. In fact, concern for detail in the early stages may block the problem-solving process.
- 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.
- 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.

You can use **Scanner** to read input for the year and the month.

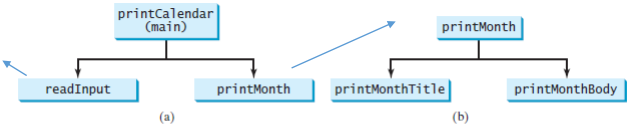


FIGURE 6.8 The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

Top-Down Design example: printMonth

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

Enter month as number between 1 and 12: 3

         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
```

- 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**), as shown in Figure 6.9b.

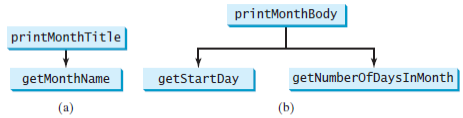


FIGURE 6.9 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

Top-Down Design example: getStartDay

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

Enter month as number between 1 and 12: 3

         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
```

- How would you get the start day for the first date in a month?
- There are several ways to do so. For now, 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.
- 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.

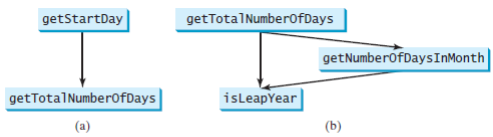


FIGURE 6.10 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

Listing 6.12 PrintCalendar

- The complete structure chart for the calendar example is shown in Figure 6.11

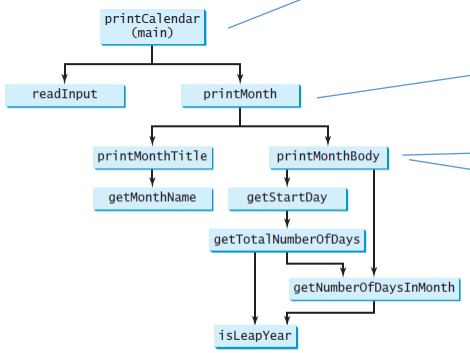


FIGURE 6.11 The structure chart shows the hierarchical relationship of the subproblems in the program.

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
```

To be continued on next slide

Listing 6.12 PrintCalendar.java (cont'd)

```
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
```

```
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 */
59   public static void printMonthBody(int year, int month) {
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   }
```

Please read the remaining implementation details from the textbook.

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. It also better facilitates teamwork.

Chapter Summary

- The method header specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method.
- 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**.
- 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.
- The arguments that are passed to a method should have the same number, type, and order as the parameters in the method signature.
- A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.

Chapter Summary

- 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.
- 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.
- *Method abstraction* is achieved by separating the use of a method from its implementation. 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.*

Ideas for further practice

- Read 6.10 Case Study: Generating Random Characters
- (*Sum the digits in an integer*) Write a method that computes the sum of the digits in an integer. Use the following method header:

  public static int sumDigits(**long** n)

  For example, **sumDigits(234)** returns **9** (2 + 3 + 4). (*Hint*: Use the **%** operator to extract digits, and the **/** operator to remove the extracted digit. For instance, to extract 4 from 234, use **234 % 10** (= 4). To remove 4 from 234, use **234 / 10** (= 23). Use a loop to repeatedly extract and remove the digit until all the digits are extracted. Write a test program that prompts the user to enter an integer and displays the sum of all its digits.