Each and every method in Java must be contained within a class in the same manner as a `main()` *method. However, within a class, there can be two fundamentally different types of methods: static and nonstatic. Although the term static is a holdover from C++ and is not very descriptive, its purpose is well defined: A static method is to receive all of its data as arguments and access or modify a variable that is shared by all objects of a class. How this is accomplished is presented in Chapter 10. In addition, if a static method's header line includes the keyword* `public`*, the method can be used by methods outside of its own class.*

Together, the static and public designations permit the construction of general-purpose methods that are not restricted to operating on objects of a specific class. Examples of such methods are the mathematical methods presented in Chapter 3. In this chapter, we learn how to write such general-purpose methods, pass data to them, process the passed data, and return a result. Almost all of the information learned about these methods is directly applicable to the nonstatic class methods presented in Chapter 9.

6.1 METHOD AND PARAMETER DECLARATIONS

In creating a general-purpose method, we must be concerned with both the method itself and how it interacts with other methods, such as `main()`. This includes correctly passing data into a method when it is called, having the method process the passed data, and returning one or more values when it has finished executing. In this section, we describe the first two parts of the interface: passing data to a general-purpose method and having the method correctly receive, store, and process the transmitted data.

As we have already seen with mathematical methods, a general-purpose method is called, or used, by giving the method's name and passing any data to it, as arguments, in the parentheses following the method name (Figure 6.1). As also shown in Figure 6.1, if a general-purpose method is called from outside of its class, it must also be preceded by its class name and a period.

```
MethodName (data passed to Method);
```

This identifies the called method

This passes data to the method

FIGURE 6.1 Calling and Passing Data to a Method

The called method must be able to accept the data passed to it by the method doing the calling. Only after the called method successfully receives the data can the data be manipulated to produce a useful result.

To clarify the process of sending and receiving data, consider Program 6.1, which calls a method named `findMaximum()`. The class, as shown, is not yet complete. Once the method `findMaximum()` is written and included in Program 6.1, the completed class, consisting of the methods `main()` and `findMaximum()`, can be compiled and executed.[1]

[1]The `findMaximum()` method could also be included in a separate class, as is demonstrated in Exercise 13. In practice, however, only one class would be used because one of the purposes of a class is to contain methods (and data) that are logically related.

PROGRAM 6.1

```
import javax.swing.*;
public class ShowTheCall
{
  public static void main(String[] args)
  {
    String s1;

    double firstnum, secnum;

    s1 = JOptionPane.showInputDialog("Enter a number:");
    firstnum = Double.parseDouble(s1);
    s1 = JOptionPane.showInputDialog("Great! Please enter a second number:");
    secnum = Double.parseDouble(s1);

    findMaximum(firstnum, secnum); // the method is called here

    System.exit(0);
  }  // end of method
}  // end of class
```

Let us examine the calling of the `findMaximum()` method from `main()`. We will then write `findMaximum()` to accept the data passed to it and determine the largest or maximum value of the two passed values.

The `findMaximum()` method is referred to as the **called method** because it is called or summoned into action by its reference in `main()`. The method that does the calling, in this case `main()`, is referred to as the **calling method.** The terms *called* and *calling* come from standard telephone usage, where one person calls another on a telephone. The person initiating the call is referred to as the calling party, and the person receiving the call is the called party. The same terms describe method calls.

Calling a general-purpose method is a rather easy operation. If the method is located in the same class as the method from which it is called, all that is required is that the name of the method be used and that any data passed to the method be enclosed within the parentheses following the method name; otherwise, the name of the called method must be preceded by its class name and a period. The items enclosed within the parentheses are called **arguments** of the called method (Figure 6.2). Other terms for arguments are **actual arguments** and **actual parameters.** All of these terms refer to the actual data values supplied to a method when the call is made.

```
findMaximum (firstnum, secnum);
```

This identifies the `findMaximum()` method

This causes two values to be passed to `findMaximum()`

FIGURE 6.2 Calling and Passing Two Values to `findMaximum()`

If a primitive data type variable is one of the arguments in a method call, the called method receives a copy of the value stored in the variable. For example, the statement `findMaximum(firstnum, secnum);` calls the method `findMaximum()` and causes the values currently residing in the variables `firstnum` and `secnum` to be passed to `findMaximum()`. The variable names in parentheses are arguments that provide values to the called method. After the values are passed, control is transferred to the called method.

As illustrated in Figure 6.3, the method `findMaximum()` *does not receive the variables named* `firstnum` *and* `secnum` *and has no knowledge of these variables' names.*[2] The method simply receives the values in these variables and must itself determine where to store these values before it does anything else.

Let us now begin writing the `findMaximum()` method to process the values passed to it.

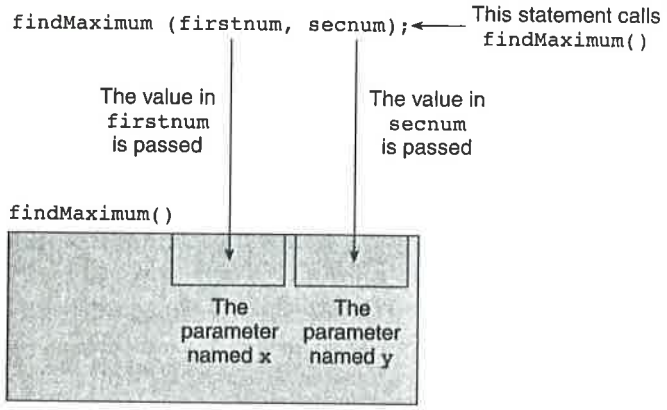


FIGURE 6.3 `findMaximum()` Receives Actual Values

Like the `main()` method, every Java method consists of two parts, a **method header** and a **method body**, as illustrated in Figure 6.4. The purpose of the method header is to specify access privileges (where the method can be called), identify the data type of the value returned by the method, provide the method with a name, and specify the number, order, and type of arguments expected by the method. The purpose of the method body is to operate on the passed data and directly return, at most, one value back to the calling method. (We will see, in Section 6.2, how a method can be made to return multiple values.)

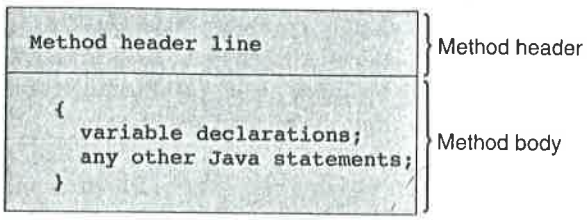


FIGURE 6.4 General Format of a Method

[2]This is significantly different from earlier high-level languages such as FORTRAN, where called methods received access to these variables and could directly alter the variables from within the called method. In Section 6.2, we will see how, using reference variables, Java permits direct access to the objects being referenced.

The structure of a header for a general-purpose method is illustrated in Figure 6.5. As shown, the header must contain the two access keywords, `public` and `static`, a mandatory return type, a name, and a set of parentheses.

```
public static returnType methodName (parameter list) <--- Required Parentheses
```

FIGURE 6.5 The Structure of a General Purpose Method's Header

Except for the `parameter list`, this header line should be very familiar to you because it is the one we have used throughout the text for our `main()` methods. To review, a general-purpose method's header must contain the keywords `public` and `static`. The `public` keyword means that the method can be used both within and outside of the class that includes the method, which is appropriate for a general-purpose method. As we will see in Chapter 9, we can restrict access to a class' methods using the `protected` and `private` keywords. The `static` keyword allows the method to be used without reference to a specific object constructed from a class. The `parameter list` in the header line provides the data types and names that will be used to hold the values passed to the method when it is called. Among other things, this list, which we will describe in detail shortly, specifies the number, sequence, and data types of the argument values that the called method expects to receive.

Using this generic header line, let us now construct a specific header line for our `findMaximum()` method. Because `findMaximum()` will not formally return any value and is to receive two double-precision values, the following header line can be used:

```
public static void findMaximum(double x, double y) <— no semicolon
```

The identifier names within the parentheses in the header are referred to as **formal parameters** of the method. You will also see them referred to as formal arguments, arguments, and parameters. Thus, the parameter `x` will store the first value passed to `findMaximum()`, and the parameter `y` will store the second value passed at the time of the method call. All parameters, such as `x` and `y`, receive values from the calling method, be they built-in or reference data types. The called method does not know where the values come from when the call is made from `main()`. The first part of the call procedure executed by the program involves going to the variables `firstnum` and `secnum` and retrieving the stored values. These values are then passed to `findMaximum()` and ultimately stored in the parameters `x` and `y` (see Figure 6.3).

The method name and all parameter names in the header, in this case `findMaximum`, `x`, and `y`, are chosen by the programmer. Any names selected according to the rules for choosing variable names can be used. All parameters listed in the method header line must be separated by commas and must have their individual data types declared separately.

Notice that as far as the method `findMaximum()` is concerned, the parameters `x` and `y` are dealt with exactly as variables. Parameter declarations declare the data type of the values expected by the method, and the order of declarations is important. The parameter declarations are identical to variable declarations. The only difference between parameter declarations and variable declarations is their placement in the method. Parameter declarations are always placed within the parentheses following the method's name, while method variable declarations are placed within the method's body. From a programming viewpoint, parameters can be considered as variables whose initialization occurs from outside the method. The number, order (sequence), and data

types of the arguments passed to a method must agree in number, order, and data type with the parameters declared in the method's header line. If the values passed to a method do not agree with the data types declared for a parameter and would result in a possible loss of precision (for example, attempting to send a double-precision argument into an integer parameter), the compiler error message `Incompatible type for method. Explicit cast needed to convert ...` or its equivalent is provided. If the number of passed arguments does not agree with the number of declared parameters, an equivalent compiler error message to `Wrong number of arguments in method` is provided.

Now that we have written the method header for `findMaximum()`, we can construct its body. Let us assume that the `findMaximum()` method selects and displays the larger of the two numbers passed to it.

As illustrated in Figure 6.6, a method body begins with an opening brace, `{`, contains any necessary declarations and other Java statements, and ends with a closing brace, `}`. Again, this should look familiar because it is the same structure used in all the `main()` methods we have written. This should not be a surprise because `main()` is itself a method and must adhere to the rules required for constructing all legitimate methods.

```
{
  variable declarations
  and other Java statements
}
```

FIGURE 6.6 The Structure of a Method Body

In the body of the `findMaximum()` method, we will declare one variable to store the maximum of the two numbers passed to it. We will then use an `if-else` statement to find the maximum of the two numbers.[3] Finally, a Message dialog will display the maximum. The complete method definition for the `findMaximum()` method is:

```
  // following is the findMaximum() method
public static void findMaximum(double x, double y)
{                  // start of method body
  double maxnum;      // variable declaration

  if (x >= y)      // find the maximum number
    maxnum = x;
  else
    maxnum = y;

  JOptionPane.showMessageDialog(null,
                "The maximum of " + x + " and " + y + " is " + maxnum,
                "Maximum Value", JOptionPane.INFORMATION_MESSAGE);

} // end of method body and end of method
```

[3]Alternatively, we could have used the `Math.max()` method (see Exercise 3).

Notice that the parameter declarations are contained within the header line and the variable declaration is placed immediately after the opening brace of the method's body. This is in keeping with the concept that parameter values are passed to a method from outside the method and method variables are declared and assigned values from within the method's body.

Program 6.2 includes the `findMaximum()` method within the program code previously listed in Program 6.1.

PROGRAM 6.2

```
import javax.swing.*;
public class CompleteTheCall
{
  public static void main(String[] args)
  {
    String s1;

    double firstnum, secnum;

    s1 = JOptionPane.showInputDialog("Enter a number:");
    firstnum = Double.parseDouble(s1);
    s1 = JOptionPane.showInputDialog("Great! Please enter a second number:");
    secnum = Double.parseDouble(s1);

    findMaximum(firstnum, secnum); // the method is called here

    System.exit(0);
  }  // end of main() method

   // following is the findMaximum() method
  public static void findMaximum(double x, double y)
  {                  // start of method body
    double maxnum;       // variable declaration

    if (x >= y)       // find the maximum number
      maxnum = x;
    else
      maxnum = y;

    JOptionPane.showMessageDialog(null,
                  "The maximum of " + x + " and " + y + " is " + maxnum,
                  "Maximum Value", JOptionPane.INFORMATION_MESSAGE);

  }  // end of method body and end of method
}  // end of class
```

Program 6.2 can be used to select and display the maximum of any two double-precision numbers entered by the user. Figure 6.7 illustrates the output displayed by Program 6.2 when the values 97.6 and 45.3 were accepted as the user-entered data.

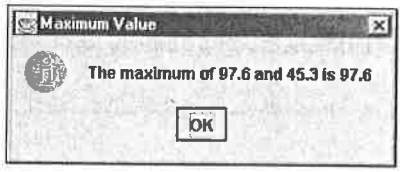


FIGURE 6.7 A Sample Display Produced by Program 6.2

The placement of the `findMaximum()` method after the `main()` method in Program 6.2 is a matter of choice. We will always list `main()` as the first `public static` method. How the processing gets "stitched together" can then be ascertained by referring to the `main()` method before the details of each method are encountered. In Part Two, when nonstatic class methods are introduced, these methods will always be written before `main()`.[4]

With one notable exception, nesting of methods is not permitted in Java, and each method must be written by itself outside of any other method. Thus, the definition of `findMaximum()` cannot be placed inside `main()`. The one exception, which is presented in detail in Part Three, is that classes can be nested within classes and can even be contained within a method. So although a Java method can never be directly nested within another method, it can be indirectly nested when it is contained within a nested class. For now, however, since we are only dealing with a single class in each of our programs, make sure to construct each Java method as a separate and independent entity with its own parameters and variables.

Placement of Statements

Java does not impose a rigid statement ordering structure on the programmer. The general rule for placing statements in a Java method is simply that all named constants and variables must be declared before they can be used. As we have noted previously, although this rule permits declaration statements to be placed throughout a method, doing so results in very poor program structure.

As a matter of good programming style, the following statement ordering should form the basic structure around which all of your Java general-purpose methods are constructed. With a minor modification to the header line, this is the same structure used for the nonstatic methods presented in Chapter 9.

```
public static returnType methodName(parameter declarations)
{
  named constants
```

[4]This is done because most of an object-oriented program's work should be defined by nonstatic class methods that are carefully thought out and listed first, which gives anyone reading the program an idea of what the class and its processing are all about. In these cases, the `main()` method may do no more than create an instance of the class within which it is contained.

```
    variable declarations
    other statements
    return value
}
```

As always, comment statements can be freely intermixed anywhere within this basic structure.

Method Stubs

An alternative to completing each method required in a complete program is to write the `main()` method first and add other general-purpose methods later as they are developed. The problem that arises with this approach, however, is the same problem that occurred with Program 6.1; that is, the program cannot be run until all of the methods are included. For convenience, we reproduce the code for Program 6.1 here:

PROGRAM 6.3

```
import javax.swing.*;
public class ShowTheCall
{
  public static void main(String[] args)
  {
    String s1;

    double firstnum, secnum;

    s1 = JOptionPane.showInputDialog("Enter a number:");
    firstnum = Double.parseDouble(s1);
    s1 = JOptionPane.showInputDialog("Great! Please enter a second number:");
    secnum = Double.parseDouble(s1);

    findMaximum(firstnum, secnum); // the method is called here

    System.exit(0);
  }  // end of method
}  // end of class
```

This program would be complete if there were a method defined for `findMaximum()`. But we really don't need a *correct* `findMaximum()` method to test and run what has been written; we just need a method that *acts* like it is: A "fake" `findMaximum()` that accepts the proper number and types of parameters and returns a value of the proper form for the method call is all we need to allow initial testing. This fake method is called a stub. A **stub** is the beginning of a method that can be used as a placeholder for the final method until the method is completed. A stub for `findMaximum()` is as follows:

```
public static void findMaximum(double x, double y)
{
  String outMessage;

  outMessage = "In findMaximum()"
             + "\nThe value of x is " + x
             + "\nThe value of y is " + y;

JOptionPane.showMessageDialog(null, outMessage, "Maximum Value",
             JOptionPane.INFORMATION_MESSAGE);
}
```

This stub method can now be compiled and linked with the previously completed `main()` method to obtain an executable program. The code for the stub method can then be further developed, with the "real" code, when it is completed, replacing the stub portion.

The minimum requirement of a stub method is that it compile and link with its calling module. In practice, it is a good idea to have a stub display its received parameters, as in the stub for `findMaximum()`. As the method is refined, you let it do more and more, perhaps allowing it to return intermediate or incomplete results. This incremental, or stepwise, refinement is an important concept in efficient program development that provides you with the means of running a program that does not yet meet all of its final requirements.

Methods with Empty Parameter Lists

Although useful general-purpose methods having an empty parameter list are extremely limited (one such method is provided in Exercise 12 and another is Section 6.6), they can occur. In any case, to indicate that no parameters are used requires writing nothing at all between the parentheses following the method's name. For example, the header line

```
public static int display()
```

indicates that the `display()` method takes no parameters and returns an integer. A method with an empty parameter list is called by its name with nothing written within the required parentheses following the method's name. For example, the statement `display();` correctly calls the `display()` method whose header line was just given.

Reusing Method Names (Overloading)[5]

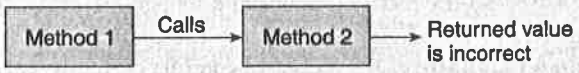
Java provides the capability of using the same method name for more than one method, which is referred to as **method overloading**. The only requirement in creating more than one method with the same name is that the compiler must be able to determine which method to use based on the data types of the parameters (not the data type of the return value, if any). For example, consider the three following methods, all named `calcAbs()`:

```
public static void calcAbs(int x)  // compute and display the
{                                  // absolute value of an integer
```

[5]This topic may be omitted on first reading with no loss of subject continuity.

POINT OF INFORMATION ISOLATION TESTING

One of the most successful software testing methods known is always to embed the code being tested within an environment of working code. For example, assume you have two untested methods that are called in the order shown, and the result returned by the second method is incorrect.



From the information in the figure, one or possibly both of the methods could be operating incorrectly. The first order of business is to isolate the problem to a specific method.

One of the most powerful methods of performing this code isolation is to decouple the methods. This is done either by testing each method individually or by testing one method first, and only when you know it is operating correctly, reconnecting it to the second method. Then, if an error occurs, you have isolated the error to either the transfer of data between methods or the internal operation of the second method.

This specific procedure is an example of the *Basic rule of testing*, which states that each method should only be tested in a program in which all other methods are known to be correct. This means that one method must first be tested by itself, using stubs if necessary for any called methods, that a second tested method should be tested either by itself or with a previously tested method, and so on. This ensures that each new method is isolated within a test bed of correct methods, with the final program effectively built up of tested method code.

```
  if ( x < 0 )
    x = -x;
  System.out.println("The absolute value of the integer is  " + x);
}
public static void calcAbs(float x)  // compute and display the
{                                     // absolute value of a float
  if ( x < 0 )
    x = -x;
  System.out.println("The absolute value of the float is  " + x);
}
public static void calcAbs(double x)  // compute and display the
{                                      // absolute value of a double
  if ( x < 0 )
    x = -x;
  System.out.println("The absolute value of the double is  " + x);
}
```

Which of the three methods named `calcAbs()` is called depends on the argument types supplied at the time of the call. Thus, the method call `calcAbs(10);` would cause the

compiler to use the method named `calcAbs()` that expects an integer argument, and the method call `calcAbs(6.28f);` would cause the compiler to use the method named `calcAbs()` that expects a floating-point argument.

Notice that overloading a method's name simply means using the same name for more than one method. Each method that uses the name must still be written and exists as a separate entity. The use of the same method name does not require that the code within the methods be similar, although good programming practice dictates that methods with the same name should perform similar operations.

Clearly, overloading a method requires that the compiler can distinguish which method to select based on the method's name and its parameter list. The part of a header line that contains this information (that is, name and parameter list) is referred to as the **method's parameter signature.** For a class to compile correctly, each method must have a unique parameter signature. This terminology is derived from everyday usage where it is expected that each individual has a unique signature that can uniquely identify a person. From a compiler's viewpoint, unless each method has a unique parameter signature, the compiler cannot correctly determine which method is being referenced.

EXERCISES 6.1

1. For the following method headers, determine the number, type, and order (sequence) of the values that must be passed to the method:
   a. `public static void factorial(int n)`
   b. `public static void price(int type, double yield, double maturity)`
   c. `public static void yield(int type, double price, double maturity)`
   d. `public static void interest(char flag, double price, double time)`
   e. `public static void total(double amount, double rate)`
   f. `public static void roi(int a, int b, char c, char d, double e, double f)`
   g. `public static void getVal(int item, int iter, char decflag, char delim)`
2. Enter and execute Program 6.2 on your computer.
3. Rewrite the `findMaximum()` method used in Program 6.2 to use the `Math.max()` method in place of the `if-else` statement used currently. Note that `Math.max(a, b)` returns the maximum of the values stored in the variables `a` and `b`.
4. a. Write a general-purpose method named `check()` that has three parameters. The first parameter should accept an integer number, the second parameter a floating-point number, and the third parameter a double-precision number. The body of the method should only display the values of the data passed to the method when it is called. (*Note:* When tracing errors in methods, it is helpful to have the method display the values it has been passed. Quite frequently, the error is not in what the body of the method does with the data, but in the data received and stored.)
   b. Include the method written in Exercise 4a in a working program. Make sure your method is called from `main()`. Test the method by passing various data to it.
5. a. Write a general-purpose method named `findAbs()` that accepts a double-precision number passed to it, computes its absolute value, and displays the absolute value. The absolute value of a number is the number itself if the number is positive and the negative of the number if the number is negative.

6.2 RETURNING A SINGLE VALUE

Using the method of passing arguments into a method presented in the previous section, the called method only receives copies of the values contained in the arguments at the time of the call (review Figure 6.3 if this is unclear). This is true for both built-in and reference arguments. Although this procedure for passing data to a method may seem surprising, it is really a safety procedure for ensuring that a called method does not inadvertently change data stored in the variables of the calling method. The called method gets a copy of the data to use. It may change its copy and, of course, change any variables declared inside itself, or an object referenced by a passed reference value.[6] This procedure, where only values are passed to a called method, is formally referred to as **pass by value** (the term **call by value** is also used).[6]

The method receiving the passed by value arguments may process the data sent to it in any fashion desired and directly return at most one, and only one, "legitimate" value to the calling method (Figure 6.8). In this section, we first see how such a value is returned to the calling method when primitive data types are used as arguments. As you might expect, given Java's flexibility, there is a way of returning more than a single value How to do this is presented at the end of this section after the passing and processing of reference variables is discussed.

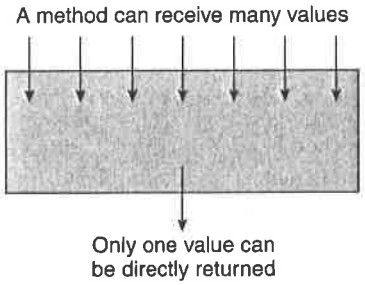


FIGURE 6.8 A Method Directly Returns at Most One Value

As with the calling of a method, directly returning a value requires that the interface between the called and calling methods be handled correctly. From its side of the return transaction, the called method must provide the following items:

- the data type of the returned value
- the actual value being returned

A method returning a value specifies the data type of the value that will be returned in its header line. As a specific example, consider the `findMaximum()` method written in the last section. It determined the maximum value of two numbers passed to the method. For convenience, the `findMaximum()` code is listed here again:

[6]Except, of course, for strings, which are immutable and, therefore, cannot be modified.

```
public static void findMaximum(double x, double y)
{                    // start of method body
  double maxnum;        // variable declaration

  if (x >= y)        // find the maximum number
    maxnum = x;
  else
    maxnum = y;

  JOptionPane.showMessageDialog(null,
          "The maximum of the two numbers is " + maxnum,
          "Maximum Value", JOptionPane.INFORMATION_MESSAGE);

}  // end of method body and end of method
```

As written, the method's header line is

```
public static void findMaximum (double x, double y)
```

where x and y are the names chosen for the method's parameters. It is the keyword `void` in the header line that is used to specify that the method will return no value. If `findMaximum()` is now to return a value, this keyword must be changed to indicate the data type of the value being returned. For example, if a double-precision value is to be returned, the proper method header line is[7]

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

Observe that this is the same as the original method header line for `findMaximum()` with the substitution of the keyword `double` for the keyword `void`. Similarly, if the method is to receive two integer values and return an integer value, the correct method header line is

```
public static int findMaximum (int x, int y)
```

and if the method is to receive two single-precision values and return a single-precision value, the header line is

```
public static float findMaximum(float x, float y)
```

Because `findMaximum()` will be used to return a double-precision value, which is the maximum value of the two double-precision numbers passed to it, the appropriate header line is

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

Having declared the data type that `findMaximum()` will return, we must now alter the method's body by including a statement that will force the return of the correct value. To return a value from a method requires using a `return` statement, which has the syntax:[8]

```
return expression;
```

[7]The return data type is related only to the parameter data types inasmuch as the returned value is computed from parameter values. In this case, because the method is used to return the maximum value of its parameters, it makes little sense to return a data type that does not match the function's parameter types.

[8]Some programmers place the expression within parentheses, yielding the statement `return (expression);`. The parentheses are not required but can be used.

When the return statement is encountered, the *expression* is evaluated first and it is this value that is sent back to the calling method. After the value is returned, program control reverts to the calling method. Because the maximum value determined by `findMaximum()` is stored in the double-precision variable `maxnum`, it is the value of this variable that should be returned. To return this value, all we need to do is add the statement `return maxnum;` before the closing brace of the `findMaximum()` method. The complete method code is:

```
public static double findMaximum(double x, double y)  // header line
{                                                      // start of method body
  double maxnum;                                       // variable declaration

   if (x >= y)
     maxnum = x;
   else
     maxnum = y;

   return maxnum;                                      // return statement
}
```

These should be the same data type

In this new code for the `findMaximum()` method, note that the data type of the expression contained in the `return` statement correctly matches the data type indicated for the returned type in the method's header line. It is up to the programmer to ensure that this is so for every method returning a value. Failure to match the return value exactly with the method's declared data type results in an error when your program is compiled. If an attempt is made to return a value that has more precision than the return type declared in the header line, the compiler will alert you with the error message `Incompatible type for return. Explicit cast needed to convert...` For example, you will receive this error message if you attempt to return a double-precision value when an integer has been declared as the return type. However, attempting to return a value that has less precision than that declared in the header line is permitted. For example, if you attempt to return an integer value from a method that has been declared as returning a double-precision value, the integer is automatically promoted to a double-precision value and no compiler error message is generated.

Having taken care of the sending side of the return transaction, we must now prepare the calling method to properly receive the returned value. To do so, the calling program must either provide a variable to store the returned value or use the value directly in an expression. Storing the returned value in a variable is accomplished using a standard assignment statement. For example, the statement

```
max = findMaximum(firstnum, secnum);
```

can be used to store the value returned by `findMaximum()` in the variable named `max`. This assignment statement does two things. First the right side of the assignment statement calls `findMaximum()`, and then the result returned by this method is stored in the variable `max`. Since the value returned by `findMaximum()` is a double-precision value, the variable `max` must also be declared as a double-precision variable within the calling method's variable declarations.

POINT OF INFORMATION PRECONDITIONS AND POSTCONDITIONS

Preconditions are any set of conditions required by a method to be true if it is to operate correctly when it is called. Thus, preconditions are primarily concerned with conditions that must be true at run time. On the other hand, a postcondition is a condition that will be true after the method is executed, assuming that the preconditions are met.

Pre- and postconditions are typically written as user comments. For example, consider the following method header line and comments:

```
boolean leapyr(int year)
/*  Precondition: the parameter year must be a year as a four-digit
      integer, such as 2003
    Postcondition: a true is returned if the year is a leap year; otherwise
      a false will be returned
*/
```

Pre- and postcondition comments should be included with both class and method definitions whenever clarification is needed.

The value returned by a method need not be stored directly in a variable, but can be used wherever an expression is valid. For example, the expression `2 * findMaximum(firstnum, secnum)` multiplies the value returned by `findMaximum()` by two, and the statement

```
System.out.println(findMaximum(firstnum, secnum));
```

displays the returned value.

Program 6.3 illustrates the inclusion of an assignment statement for `main()` to correctly call and store a returned value from `findMaximum()`. As before and in keeping with our convention of placing the `main()` method first, we have placed the `findMaximum()` method after `main()`.

In reviewing Program 6.3, it is important to note the four items we have introduced in this section. The first item to notice in `main()` is the use of an assignment statement to store the returned value from the `findMaximum()` call into the variable `max`. Second, we have also made sure to correctly declare `max` as a double within `main()`'s variable declarations so that it matches the data type of the returned value.

The last two items concern the coding of the `findMaximum()` method. The header line for `findMaximum()` declares that the method will return a double, and the expression in the `return` statement evaluates to a matching data type. Thus, `findMaximum()` is internally consistent in sending a double-precision value back to `main()`, and `main()` has been correctly written to receive and use the returned double-precision value.

PROGRAM 6.3

```
import javax.swing.*;
public class FindTheMaximum
{
   public static void main(String[] args)
  {
    String s1, outMessage;

    double firstnum, secnum, max;

    s1 = JOptionPane.showInputDialog("Enter a number:");
    firstnum = Double.parseDouble(s1);
    s1 = JOptionPane.showInputDialog("Great! Please enter a second number:");
    secnum = Double.parseDouble(s1);

    max = findMaximum(firstnum, secnum); // the method is called here

    JOptionPane.showMessageDialog(null,
            "The maximum of the two numbers is " + max,
            "Maximum Value", JOptionPane.INFORMATION_MESSAGE);

    System.exit(0);
  }

    // following is the method findMaximum()
  public static double findMaximum(double x, double y)
  {                  // start of method body
    double maxnum;      // variable declaration
    if (x >= y)       // find the maximum number
      maxnum = x;
    else
      maxnum = y;

    return maxnum;
  }  // end of method body and end of method
}
```

In writing your own general-purpose methods, you must always keep these four items in mind. For another example, see if you can identify these items in Program 6.4. In reviewing Program 6.4, let us first analyze the `convertTemp()` method. The complete definition of the method begins with the method's header line and ends with the closing brace after the `return` statement. The method is declared as returning a double; this means the expression in the method's `return` statement should evaluate to a double-precision number, which it does.

PROGRAM 6.4

```
import javax.swing.*;
public class CheckItems
{
  public static void main(String[] args)
  {
    final int CONVERTS = 4;  // number of conversions to be made
    String s1;
    int count;
    double fahren;

    for(count = 1; count <= CONVERTS; count++)
    {
      s1 = JOptionPane.showInputDialog("Enter a Fahrenheit temperature:");
      fahren = Double.parseDouble(s1);
      JOptionPane.showMessageDialog(null,
            "The Celsius equivalent is " + convertTemp(fahren),
            "Program 6.4", JOptionPane.INFORMATION_MESSAGE);
    }

    System.exit(0);
  }

  // convert fahrenheit to celsius

  public static double convertTemp(double inTemp)
  {
    return 5.0/9.0 * (inTemp - 32.0);
  }

}
```

On the receiving side, `main()` correctly calls `convertTemp()` and uses its returned value. No variable is declared in `main()` to store the returned value from `convertTemp()` because the returned value is immediately passed to a `showMessage()` method for display.

Passing a Reference Value

Java passes the value that is stored in both a primitive data type variable and a reference variable in the same manner: A copy of the value in the variable is passed to the called method and stored in one of the method's formal parameters. A consequence is that any change to the parameter's value has no effect on the argument's value. From a programming viewpoint, this means that an argument's value can never be altered from within a called method.

Passing arguments by value in a method call has its advantages. It allows methods to be written as independent entities that can use any variable or parameter name without concern that other methods may also be using the same name. It also alleviates any concern that altering a parameter or variable in one method may inadvertently alter the value of a variable in another method. Under this approach, parameters can be considered as variables that are initialized from the calling method.

Although both primitive and reference variable values are passed in the same way, passing a reference value does have implications that passing a primitive value does not. With respect to primitive values, the best a called method can do is receive values from the calling method, store and manipulate the passed values, and directly return at most a single value. With respect to reference values, however, the called method gets to access the exact same object as is referenced by the calling method. This situation is illustrated in Figure 6.9, where the calling method's reference variable and the called method's parameter reference the same object.

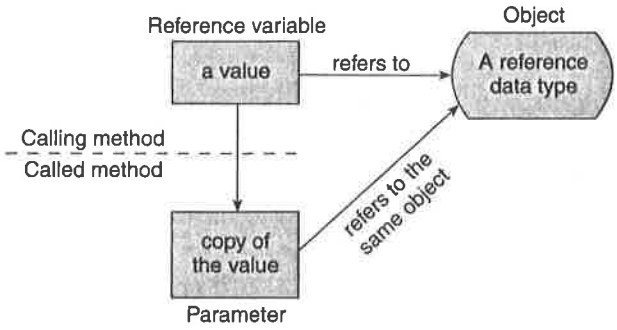


FIGURE 6.9 Passing a Reference Value

Program 6.5 illustrates how a reference value is passed in practice. Notice that the method call and parameter declaration are essentially the same as are used for primitive data types.

PROGRAM 6.5

```
public class PassAReference
{
  public static void main(String[] args)
  {
    String s1 = "Original Message";

    display(s1);   // call the method
  }  // end of main() method

  public static void display(String msg)
  {

    System.out.println("From within display(): " + msg);
  }
}  // end of class
```

The output produced by Program 6.5 is:

```
From within display(): Original Message
```

As seen by this output, we have successfully passed a reference value as an argument to a called method and used the passed value to access a string created in the calling method. Because both the value in the reference variable named `s1` in `main()` and the value passed to the parameter named `msg` in `display()` are the same, the `display()` method now has direct access to the string created in `main()`. What the `display()` method has been programmed to do with this access is simply to obtain the string value and display it.

Having obtained a reference value, any change to the object being referenced in the called method will be reflected whenever the object is referenced in the calling method. However, if the parameter is assigned to a completely new object, the change in reference value in the parameter will not affect the reference value in the calling method. Thus, if a new object is created within the called method, the originally referenced argument is not affected, and no change is made to the original object's value. This is in keeping with the basic Java philosophy that a called method cannot alter the value of a calling method's arguments.

The ability to alter a referenced object from within a method is useful in cases where a method must return more than one primitive value. For these cases, the primitive data type must first be converted into an equivalent wrapper object type (see Section 3.2), and then the reference to this object is passed to the called method. The called method can then compute a new value and, using its parameter, place the computed value into the referenced location, as illustrated in Figure 6.9. This new value is then accessed in the calling method using its reference variable and, if necessary, converted back to a primitive data type.

It should be noted that this will not work with a string value because the String class only creates immutable objects. This means that once a string object is initialized, its contents cannot be changed, and you actually create a new string object each time you assign a string constant to a string variable. Thus, for example, if the string referenced by `msg` in Program 6.5 is changed in `display()` by a statement such as `msg = "Have a happy day";`, the change will not be accessible from within `main()`. Creating a new string breaks the correspondence between the reference value stored in `s1` within `main()`, which still refers to the original string, and the reference value in the parameter `msg`. The value in `msg` will be changed to refer to the new string.

EXERCISES 6.2

1. a. Rewrite Program 6.3 to have the method `findMaximum()` accept two integer values and return an integer value. Make sure to modify `main()` so that it passes two integer values to `findMaximum()` and accepts and stores an integer value from `findMaximum()`.
   b. Modify the `findMaximum()` method written for Exercise 1a so that it attempts to return a double-precision value, even though its header line declares it as returning an integer. Determine the error message generated by the compiler.
2. For the following method headers, determine the number, type, and order (sequence) of values that should be passed to the method when it is called and the data type of the value returned by the method.