Chapter 9

Advanced Modularization Techniques

- Chapter Introduction
- 9-1 Using Methods with No Parameters
- 9-2 Creating Methods that Require Parameters
  - 9-2a Creating Methods that Require Multiple Parameters
- 9-3 Creating Methods that Return a Value
  - 9-3a Using an IPO Chart
- 9-4 Passing an Array to a Method
- 9-5 Overloading Methods
  - 9-5a Avoiding Ambiguous Methods
- 9-6 Using Predefined Methods
- 9-7 Method Design Issues: Implementation Hiding, Cohesion, and Coupling
  - 9-7a Understanding Implementation Hiding
  - 9-7b Increasing Cohesion
  - 9-7c Reducing Coupling
- 9-8 Understanding Recursion
- 9-9 Chapter Review
  - 9-9a Chapter Summary
  - 9-9b Key Terms
  - 9-9c Review Questions
  - 9-9d Exercises

Chapter Introduction

In this chapter, you will learn about:

- Methods with no parameters
- Methods that require parameters
- Methods that return a value
- Passing arrays to methods
- Overloading methods
- Using predefined methods
- Method design issues, including implementation hiding, cohesion, and coupling
- Recursion

9-1 Using Methods with No Parameters

In object-oriented programming languages such as Java and C#, modules are most often called *methods*. In Chapter 2, you learned about many features of methods and much of the vocabulary associated with them. For example:

- A **method** (A series of statements that carry out a task.) is a program module that contains a series of statements that carry out a task; you can invoke or call a method from another program or method.
- Any program can contain an unlimited number of methods, and each method can be called an unlimited number of times.
- The rules for naming methods are different in every programming language, but they often are similar to the language's rules for variable names. In this text, method names are followed by a set of parentheses.
- A method must include a **method header** (A program component that precedes a method's implementation; the header includes the method identifier and possibly other necessary information.) (also called the declaration or definition), which contains identifying information about the method.
- A method includes a **method body** (The set of all the statements in a method.) . The

body contains the method's **implementation** (The body of a method; the statements that carry out the tasks of a method.) —the statements that carry out the method's tasks.

- A **method return statement** (A statement that marks the end of the method and identifies the point at which control returns to the calling method.) returns control to the calling method after a method executes. Although methods with multiple `return` statements are allowed in many programming languages, that style is not recommended. Structured programming requires that a method should contain only one `return` statement—the last statement in the method.

- Variables and constants can be declared within a method. A data item declared in a method is **local** (Describes variables that are declared within the method that uses them.) to that method, meaning it is in scope, or recognized only within that method.

- The opposite of local is global. When a data item is known to all of a program's modules, it is a **global** (Describes variables that are known to an entire program.) data item. In general, programmers prefer local data items because when data is contained within the method that uses it, the method is more portable and less prone to error. In Chapter 2, you learned that when a method is described as *portable*, it can easily be moved to another application and used there.

Figure 9-1 shows a program that allows a user to enter a preferred language (English or Spanish) and then, using the chosen language, asks the user to enter his or her weight. The program then calculates the user's weight on the moon as 16.6 percent of the user's weight on Earth. The main program contains two variables and a constant, all of which are declared in the main program. The program calls the `displayInstructions()` method, which contains its own local variable and constants that are invisible to the main program. The method prompts the user for a language indicator and displays a prompt in the selected language. Figure 9-2 shows a typical program execution in a command-line environment.

Figure 9-1

A Program that Calculates the User's Weight on the Moon

Figure 9-2

Output of Moon Weight Calculator Program in Figure 9-1

In Chapter 2, you learned that this book uses a rectangle with a horizontal stripe across the top to represent a method call statement in a flowchart. Some programmers prefer to use a rectangle with two vertical stripes at the sides, and you should use that convention if your organization prefers it. This book reserves the shape with two vertical stripes to represent a method from a library that is external to the program.

In Figure 9-1, the main program and the called method each contain only data items that are needed at the local level. However, sometimes two or more parts of a program require access to the same data. When methods must share data, you can pass the data into methods and return data out of them. In this chapter, you will learn that when you call a method from a program or other method, you should know three things:

- The name of the called method
- What type of information to send to the method, if any
- What type of return data to expect from the method, if any

Two Truths & A Lie

Using Methods with No Parameters

1. A program can contain an unlimited number of methods, but each method can be called once.

   T F

2. A method includes a header and a body.

   T F

3. Variables and constants are in scope within, or local to, only the method in which they are declared.

   T F

9-2 Creating Methods that Require Parameters

Some methods require information to be sent in from the outside. When you pass a data item into a method from a calling program, it is called an **argument to the method** (A value passed to a method in the method call.) , or more simply, an argument. When the method receives the data item, it is called a **parameter to the method** (A data item passed into a method from the outside.) , or more simply, a parameter.

Parameter and *argument* are closely related terms. A calling method sends an argument to a called method. A called method accepts the value as its parameter.

If a method could not receive parameters, you would have to write an infinite number of methods to cover every possible situation. As a real-life example, when you make a restaurant reservation, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the person who carries out the method. The method that records the reservation is carried out in the same manner, no matter what date and time are supplied. If you design a `square()` method that squares numeric values, you should supply the method with a parameter that represents the value to be squared, rather than developing a `square1()` method that squares the value 1, a `square2()` method that squares the value 2, and so on. To call a `square()` method that accepts a parameter, you might write a statement like `square(17)` or `square(86)` and let the method use whatever argument you send.

When you write the declaration for a method that can receive a parameter, you must include the following items within the method declaration's parentheses:

- The type of the parameter
- A local name for the parameter

The types and names of parameters are the method's **parameter list** (All the data types and parameter names that appear in a method header.) . A method's name and parameter list constitute the method's **signature** (A method's name and parameter list.) .

For example, suppose that you decide to improve the moon weight program in Figure 9-1 by making the final output more user-friendly and adding the explanatory text in the chosen language. It makes sense that if the user can request a prompt in a specific language, the user would want to see the output explanation in the same language. However, in Figure 9-1, the `langCode` variable is local to the `displayInstructions()` method and therefore cannot be used in the main program. You could rewrite the program by taking several approaches:

- You could rewrite the program without including any methods. That way, you could prompt the user for a language preference and display the prompt and the result in the appropriate language. This approach would work, but you would not be taking advantage of the benefits provided by modularization. Those benefits include making the main program more streamlined and abstract, and making the `displayInstructions()` method a self-contained unit that can easily be transported to other programs—for example, applications that might determine a user's weight on Saturn or Mars.

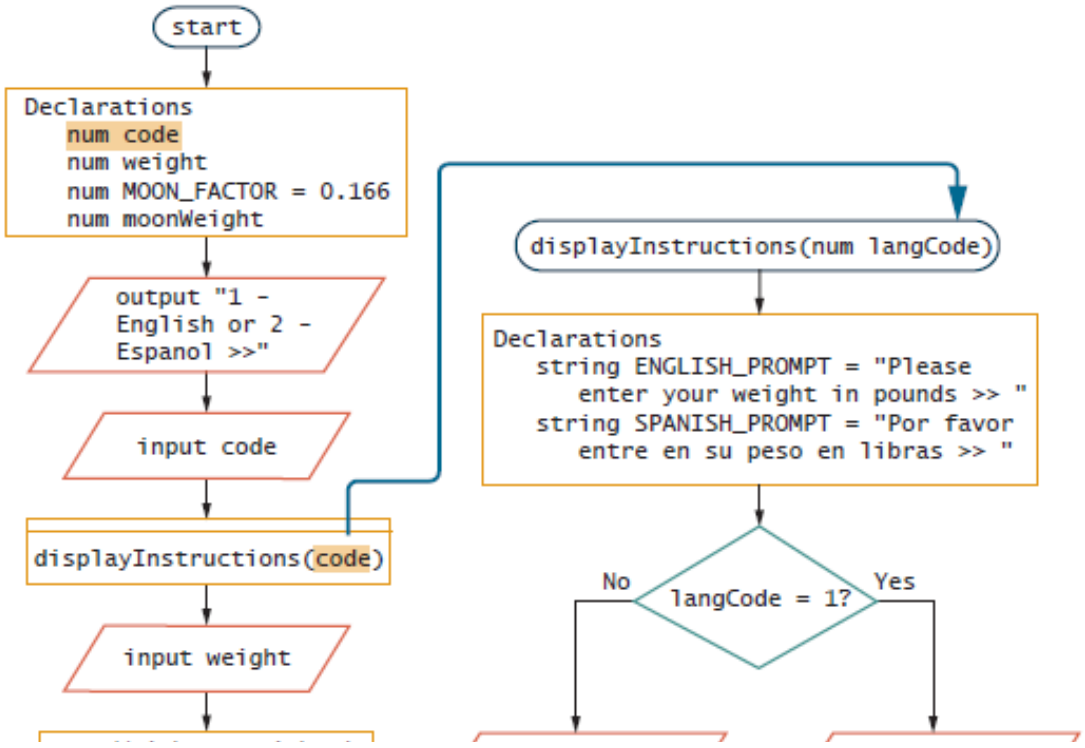
- You could retain the `displayInstructions()` method, but make at least the `langCode` variable global by declaring it outside of any methods. If you took this approach, you would lose some of the portability of the `displayInstructions()` method because everything it used would no longer be contained within the method.

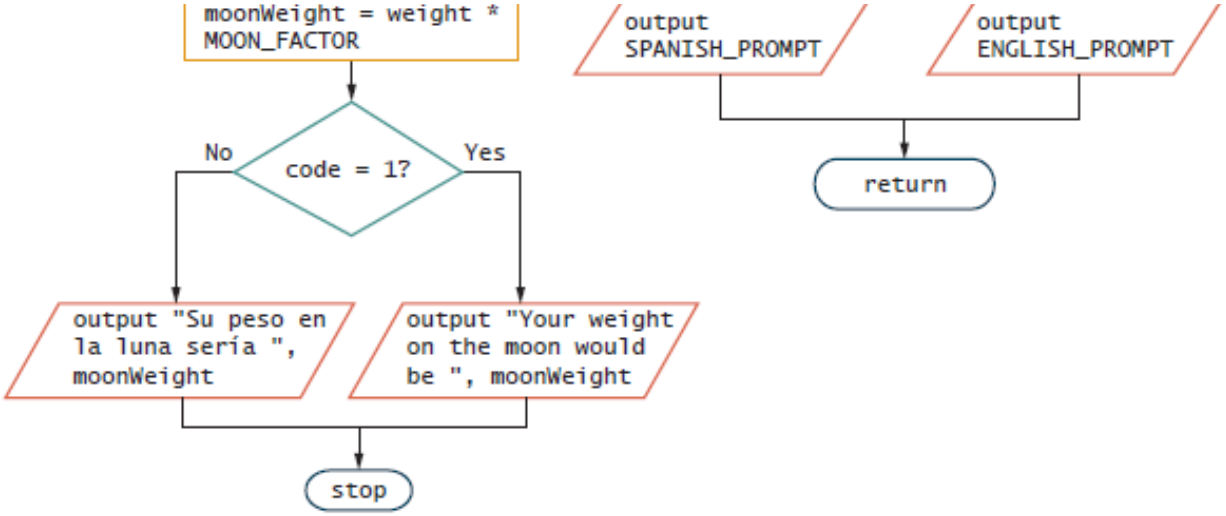
- You could retain the `displayInstructions()` method as is with its own local declarations, but add a section to the main program that also asks the user for a preferred language to display the result. The disadvantage to this approach is that the user must answer the same question twice during one execution of the program.

- You could store the variable that holds the language code in the main program so that it could be used to determine the result language. You could also retain the `displayInstructions()` method, but pass the language code to it so the prompt would appear in the appropriate language. This is the best choice, and is illustrated in Figures 9-3 and 9-4.

Figure 9-3

Moon Weight Program that Passes an Argument to a Method



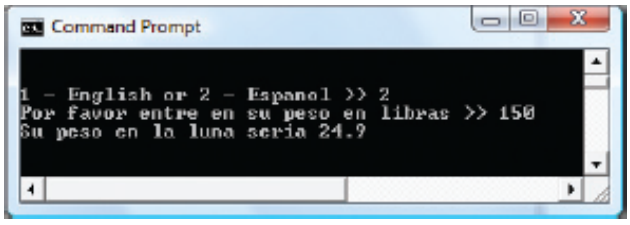



```
start
   Declarations
      num code
      num weight
      num MOON_FACTOR = 0.166
      num moonWeight
   output "1 - English or 2 - Espanol >>"
   input code
   displayInstructions(code)
   input weight
   moonWeight = weight * MOON_FACTOR
   if code = 1 then
      output "Your weight on the moon would be ", moonWeight
   else
      output "Su peso en la luna sería ", moonWeight
   endif
stop

displayInstructions(num langCode)
   Declarations
      string ENGLISH_PROMPT = "Please enter your weight in pounds >> "
      string SPANISH_PROMPT = "Por favor entre en su peso en libras >> "
   if langCode = 1 then
      output ENGLISH_PROMPT
   else
      output SPANISH_PROMPT
   endif
return
```

Figure 9-4

Typical Execution of Moon Weight Program in Figure 9-3



In the main program in Figure 9-3, a numeric variable named `code` is declared and the user is prompted for a value. The value then is passed to the `displayInstructions()` method. The value of the language code is stored in two places in memory:

- The main method stores the code in the variable `code` and passes it to `displayInstructions()` as an argument.

- The `displayInstructions()` method accepts the parameter as `langCode`. Within the method, `langCode` takes on the value that `code` had in the main program.

You can think of the parentheses in a method declaration as a funnel into the method; parameters listed there hold values that are "dropped in" to the method.

A variable passed into a method is **passed by value** (Describes a variable that has a copy of its value sent to a method and stored in a new memory location accessible to the method.); that is, a copy of its value is sent to the method and stored in a new memory location accessible to the method. The `displayInstructions()` method could be called using any numeric value as an argument, whether it is a variable, a named constant, or a literal constant. If the value used as an argument in the method call is a variable, it might possess the same identifier as the parameter declared in the method header, or it might possess a different identifier. Within a method, the passed variable is simply a temporary placeholder; it makes no difference what name the variable "goes by" in the calling program.

Each time a method executes, any parameter variables listed as parameters in the method header are redeclared—that is, new memory locations are reserved and named. When the method ends at the `return` statement, the locally declared parameter variables cease to exist. For example, Figure 9-5 shows a program that declares a variable, assigns a value to it, displays it, and sends it to a method. Within the method, the parameter is displayed, altered, and displayed again. When control returns to the main program, the original variable is displayed one last time. As the execution in Figure 9-6 shows, even though the variable in the method was altered, the original variable in the main program retains its starting value because it never was altered; it occupies a different memory address from the variable in the method.

Figure 9-5

A Program that Calls a Method in Which the Argument and Parameter Have the Same Identifier

Figure 9-6

Execution of the Program in Figure 9-5

Watch the video *Methods with a Parameter*.

9-2a Creating Methods that Require Multiple Parameters

You create and use a method with multiple parameters by doing the following:

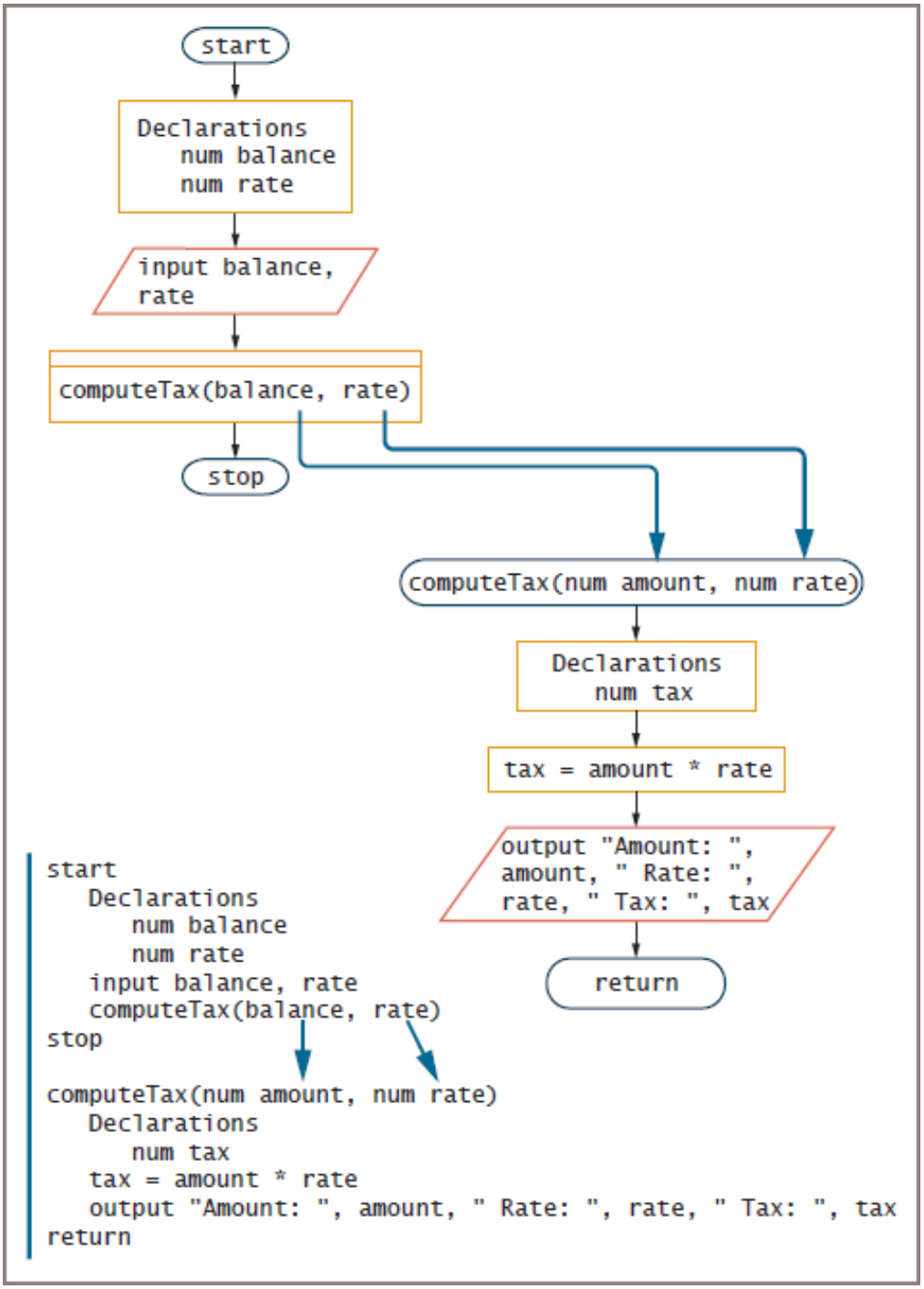
- You list the arguments within the method call, separated by commas.
- You list a data type and local identifier for each parameter within the method header's parentheses, separating each declaration with a comma. Even if multiple parameters are the same data type, the type must be repeated with each parameter.

The arguments sent to a method in a method call are its **actual parameters** (The arguments in a method call.) . The variables in the method declaration that accept the values from the actual parameters are **formal parameters** (The variables in a method declaration that accept values from the actual parameters.) .

For example, suppose that you want to create a `computeTax()` method that calculates a tax on any value passed into it. You can create a method to which you pass two values—the amount to be taxed as well as a rate by which to tax it. Figure 9-7 shows a method that accepts two such parameters.

Figure 9-7

A Program that Calls a `computetax()` Method that Requires Two Parameters



In Figure 9-7, notice that one of the arguments to the method has the same name as the corresponding method parameter, and the other has a different name from its corresponding parameter. Each could have the same identifier as its counterpart, or all could be different. Each identifier is local to its own method.

In Figure 9-7, two parameters (`num amount` and `num rate`) appear within the parentheses in the method header. A comma separates each parameter, and each requires its own declared type (in this case, both are numeric) as well as its own identifier. When multiple

values are passed to a method, they are accepted by the parameters in the order in which they are passed. You can write a method so that it takes any number of parameters in any order. However, when you call a method, the arguments you send to the method must match in order—both in number and in type—the parameters listed in the method declaration. A call of `computeTax(rate, balance)` instead of `computeTax(balance, rate)` would result in incorrect values being displayed in the output statement.

If method arguments are the same type—for example, two numeric arguments—passing them to a method in the wrong order results in a logical error; the program will compile and execute, but produce incorrect results. If a method expects arguments of diverse types—for example, a number and a string—then passing arguments in the wrong order is a syntax error, and the program will not compile.

 Watch the video *Methods with Multiple Parameters*.

Two Truths & A Lie

Creating Methods that Require Parameters

1. A value sent to a method from a calling program is a parameter.

   T F

2. When you write the declaration for a method that can receive parameters, you must include a data type for each parameter even if multiple parameters are the same type.

   T F

3. When a variable is used as an argument in a method call, it can have the same identifier as the parameter in the method header.

   T F

9-3 Creating Methods that Return a Value

A variable declared within a method ceases to exist when the method ends—it goes out of scope. When you want to retain a value that exists when a method ends, you can return the value from the method to the calling method. When a method returns a value, the method must have a **return type** (The data type for any value a method returns.) that matches the data type of the returned value. A return type can be any type, which includes `num` and `string`, as well as other types specific to the programming language you are using. A method can also return nothing, in which case the return type is `void`, and the method is a **void method** (A method that returns no value.) . (The term *void* means "nothing" or "empty.") A method's return type is known more succinctly as a **method's type** (The data type of a method's return value.) , which is listed in front of the method name when it is defined. Previously, this book has not included return types for methods because all the methods have been void. From this point forward, a return type is included with every method.

Along with an identifier and parameter list, a return type is part of a method's declaration. Some programmers claim a method's return type is part of its signature, but this is not the case. Only the method name and parameter list constitute the signature.

For example, a method that returns the number of hours an employee has worked might have the following header:

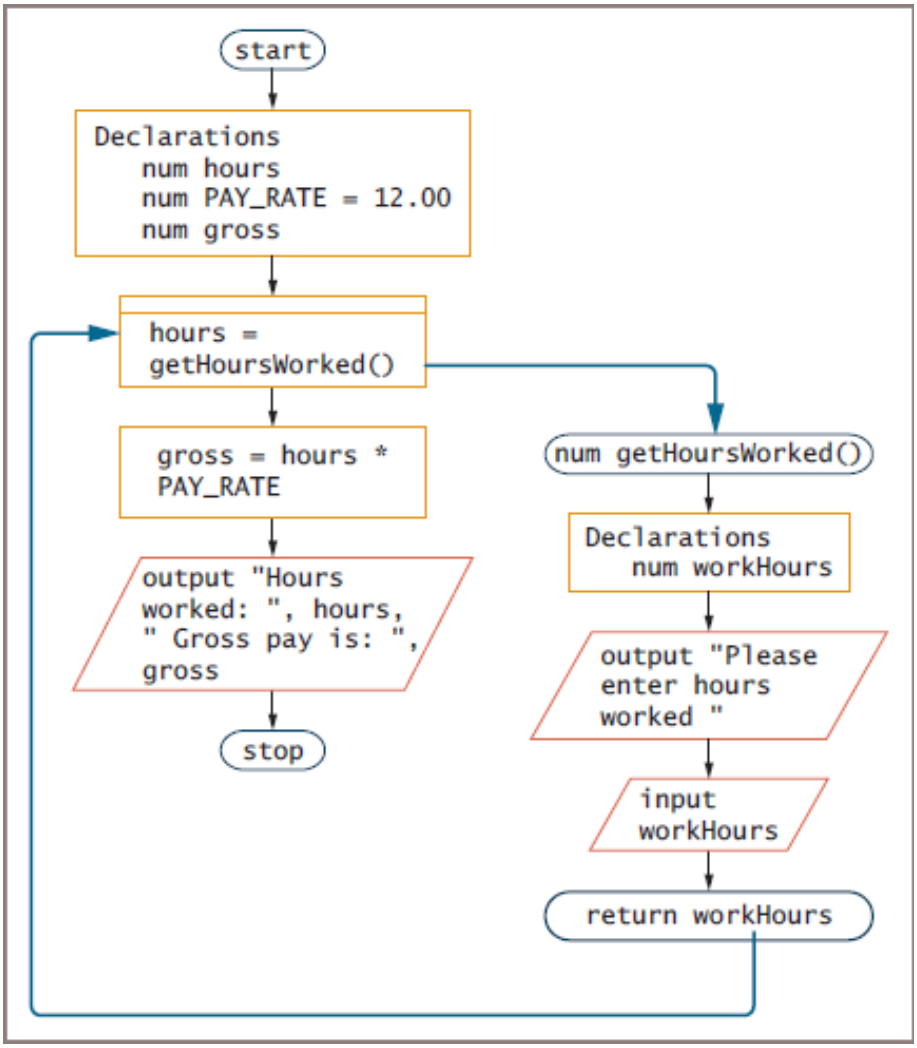
```
num getHoursWorked()
```

This method returns a numeric value, so its type is `num`.

When a method returns a value, you usually want to use the returned value in the calling method, although this is not required. For example, Figure 9-8 shows how a program might use the value returned by the `getHoursWorked()` method. A variable named `hours` is declared in the main program. The `getHoursWorked()` method call is part of an assignment statement. When the method is called, the logical control is transferred to the `getHoursWorked()` method, which contains a variable named `workHours`. A value is obtained for this variable, which is returned to the main program where it is assigned to `hours`. After logical control returns to the main program from the `getHoursWorked()` method, the method's local variable `workHours` no longer exists. However, its value has been stored in the main program where, as `hours`, it can be displayed and used in a calculation.

Figure 9-8

A Payroll Program that Calls a Method that Returns a Value



As an example of when you might call a method but not use its returned value, consider a method that gets a character from the keyboard and returns its value to the calling program. In some applications, you would want to use the value of the returned characters. However, in other applications, you might want to tell the user to press any key. Then, you could call the method to accept the character from the keyboard, but you would not care which key was pressed or which key value was returned.

In Figure 9-8, notice the return type `num` that precedes the method name in the `getHoursWorked()` method header. A method's declared return type must match the type of value used in the `return` statement; if it does not, the program will not compile. A numeric value is correctly included in the `return` statement—the last statement in the `getHoursWorked()` method. When you place a value in a `return` statement, the value is sent from the called method back to the calling method.

A method's `return` statement can return one value at most. The returned value can be a variable or a constant. The value can be a simple data type or a more complex type. For

example, in Chapter 10 you will learn to create objects, which are more complex data types.

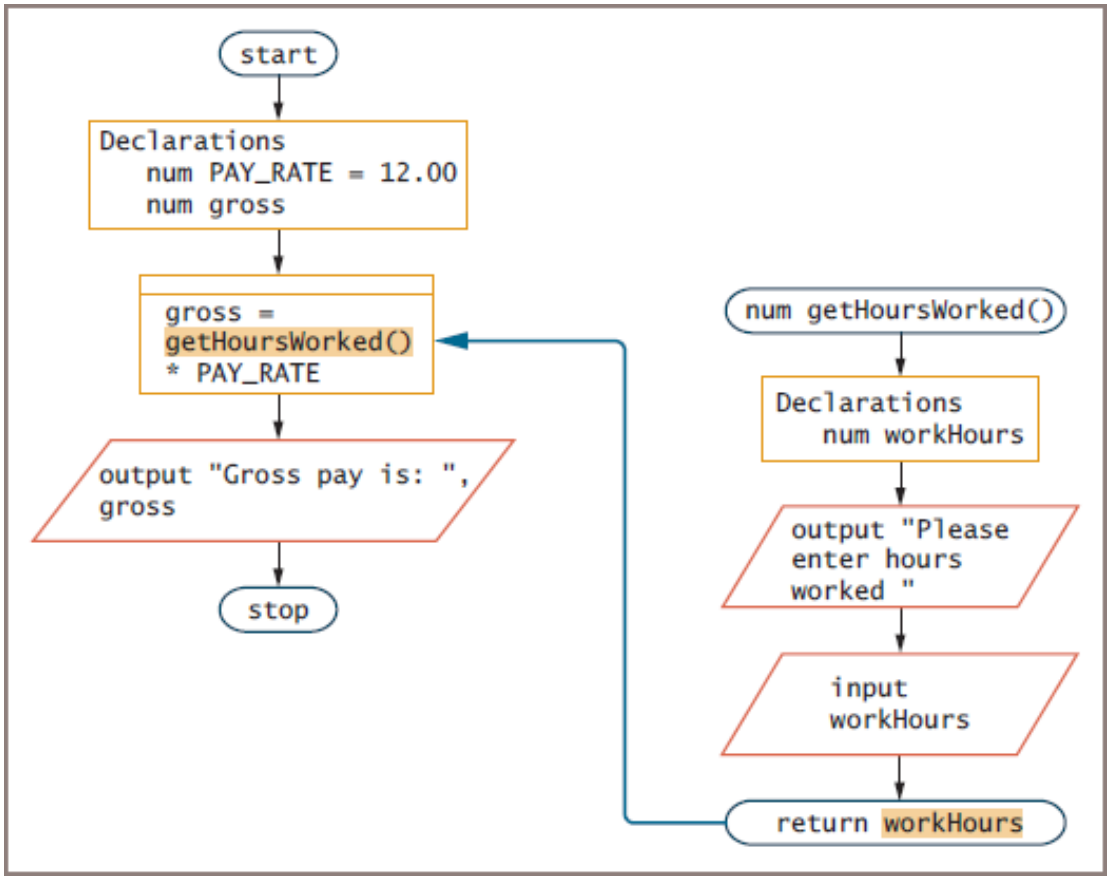
You are not required to assign a method's return value to a variable to use the value. Instead, you can use a method's returned value directly, without storing it. You use a method's value in the same way you would use any variable of the same type. For example, you can output a return value in a statement such as the following:

```
output "Hours worked is ", getHoursWorked()
```

Because `getHoursWorked()` returns a numeric value, you can use the method call `getHoursWorked()` in the same way that you would use any simple numeric value. Figure 9-9 shows an example of a program that uses a method's return value directly without storing it. The value of the shaded `workHours` variable returned from the method is used directly in the calculation of `gross` in the main program.

Figure 9-9

A Program that Uses a Method's Returned Value without Storing It



When a program needs to use a method's returned value in more than one place, it makes sense to store the returned value in a variable instead of calling the method multiple times. A program statement that calls a method requires more computer time and resources than a statement that does not call any outside methods. Programmers use the term **overhead** (All the resources and time

required by an operation.) to describe any extra time and resources required by an operation.

As mentioned earlier, in most programming languages you technically are allowed to include multiple `return` statements in a method, but this book does not recommend the practice for most business programs. For example, consider the `findLargest()` method in Figure 9-10. The method accepts three parameters and returns the largest of the values. Although this method works correctly and you might see this technique used, its style is awkward and not structured. In Chapter 3, you learned that structured logic requires each structure to contain one entry point and one exit point. The `return` statements in Figure 9-10 violate this convention by leaving decision structures before they are complete. Figure 9-11 shows the superior and recommended way to handle the problem. In Figure 9-11, the largest value is stored in a variable. Then, when the nested decision structure is complete, the stored value is returned in the last method statement.

Figure 9-10

Unstructured Approach to Returning One of Several Values

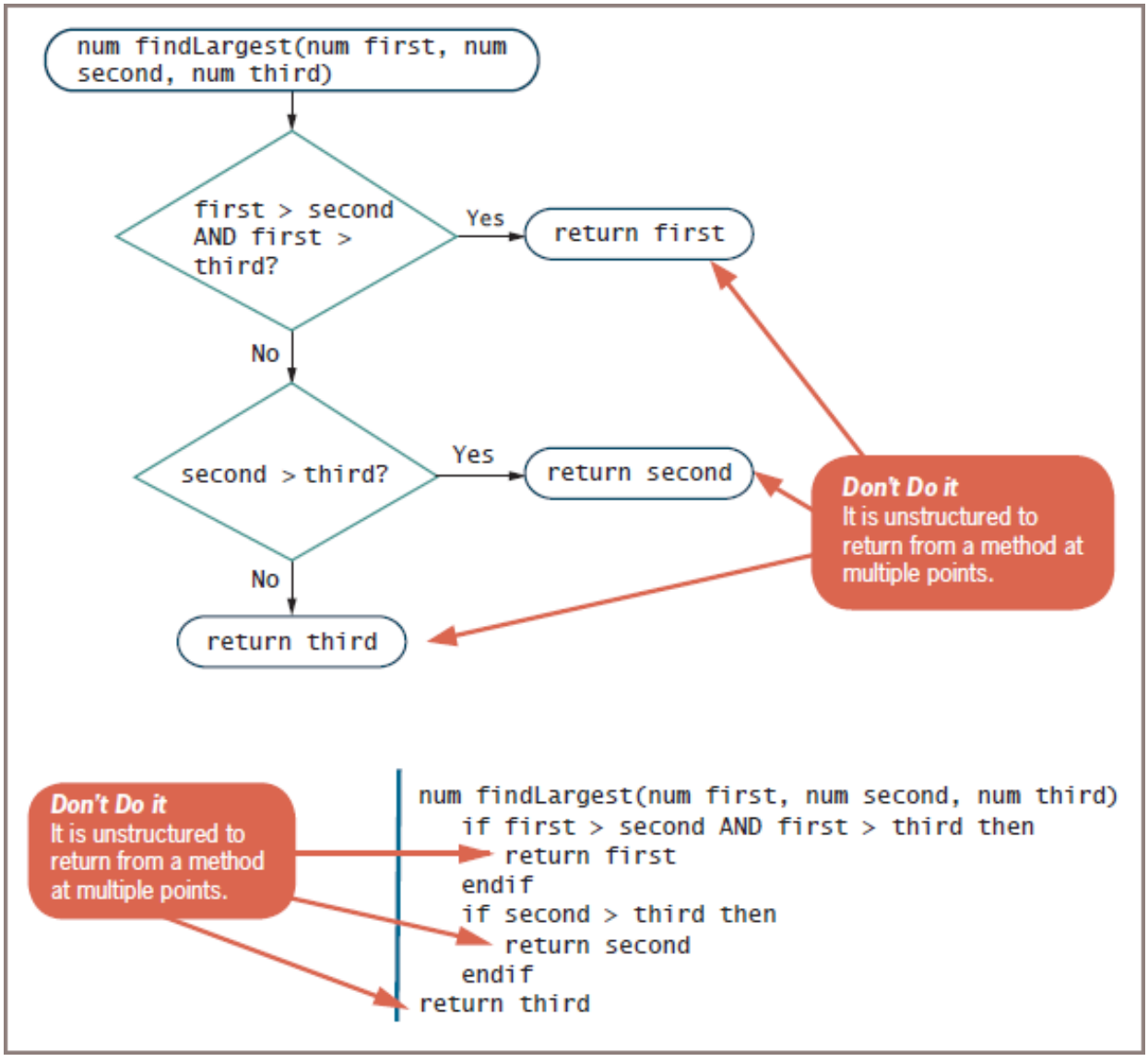
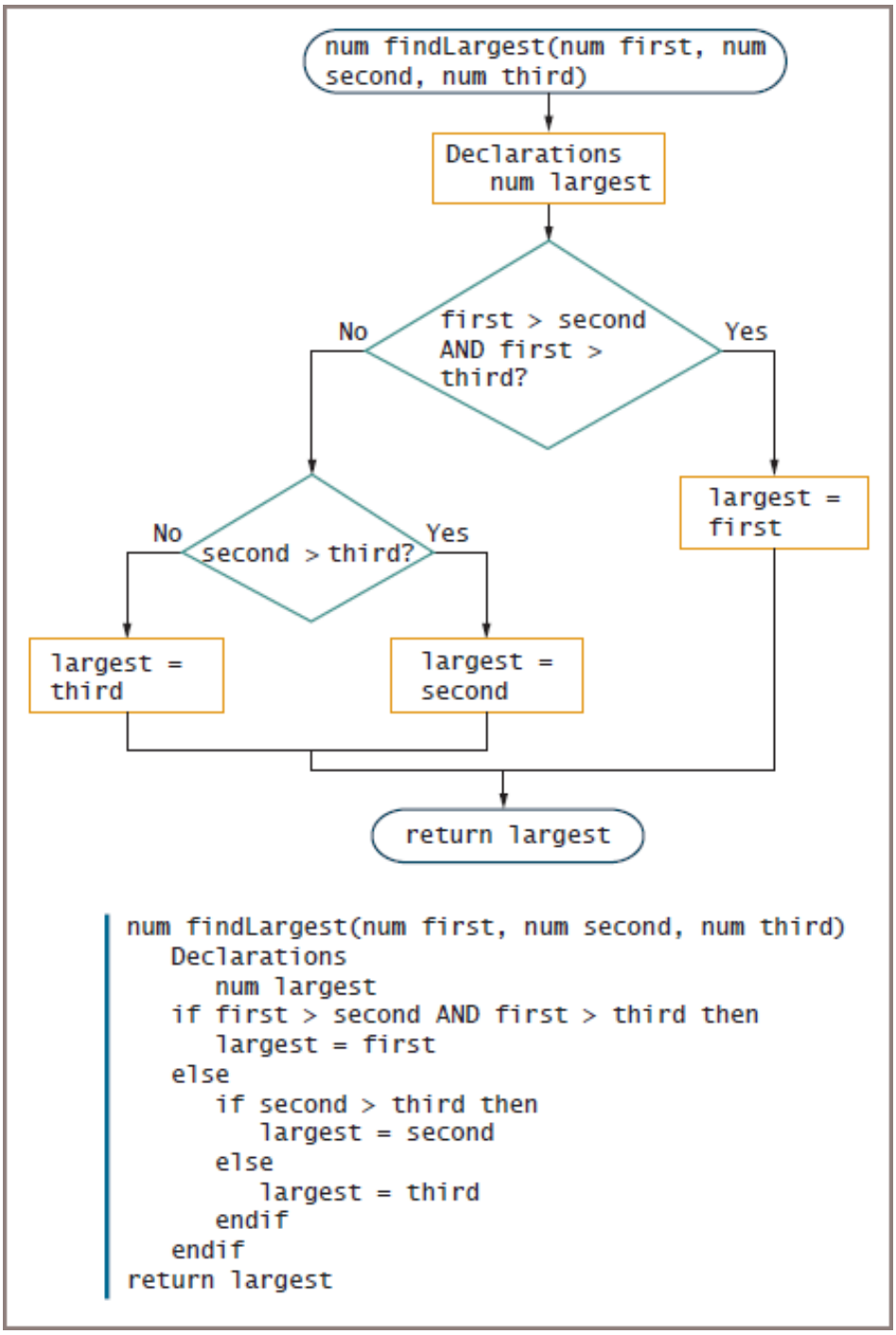


Figure 9-11

Recommeded, Structured Approach to Returning One of Several Values




```
num findLargest(num first, num second, num third)
   Declarations
      num largest
   if first > second AND first > third then
      largest = first
   else
      if second > third then
         largest = second
      else
         largest = third
      endif
   endif
return largest
```

When you want to use a method, you should know four things:

- What the method does in general, but not necessarily how it carries out tasks internally
- The method's name
- The method's required parameters, if any
- The method's return type, so that you can use any returned value appropriately

9-3a Using an IPO Chart

When designing methods to use within larger programs, some programmers find it helpful to use an **IPO chart** (A program development tool that delineates input, processing, and output tasks.) , a tool that identifies and categorizes each item needed within the method as pertaining to input, processing, or output. For example, consider a method that finds the smallest of three numeric values. When you think about designing this method, you can start by placing each of its components in one of the three processing categories, as shown in Figure 9-12.

Figure 9-12

IPO Chart for the Method that Finds the Smallest of Three Numeric Values

The IPO chart in Figure 9-12 provides an overview of the processing steps involved in the method. Like a flowchart or pseudocode, an IPO chart is just another tool to help you plan the logic of your programs. Many programmers create an IPO chart only for specific methods in their programs and as an alternative to flowcharting or writing pseudocode. IPO charts provide an overview of input to the method, the processing steps that must occur, and the result.

This book emphasizes creating flowcharts and pseudocode, but you can find many more examples of IPO charts on the Web.

Two Truths & A Lie

Creating Methods that Return a Value

1. The return type for a method can be any type, which includes numeric, character, and string, as well as other more specific types that exist in the programming language you are using.

   T F

2. A method's return type must be the same type as one of the method's parameters.

T F

3. You are not required to use a method's returned value.

   T F

9-4 Passing an Array to a Method

In Chapter 6, you learned that you can declare an array to create a list of elements, and that you can use any individual array element in the same manner you would use any single variable of the same type. For example, suppose that you declare a numeric array as follows:

You can subsequently output `someNums[0]` or perform arithmetic with `someNums[11]`, just as you would for any simple variable that is not part of an array. Similarly, you can pass a single array element to a method in exactly the same manner you would pass a variable or constant.

Consider the program shown in Figure 9-13. This program creates an array of four numeric values and then outputs them. Next, the program calls a method named `tripleTheValue()` four times, passing each of the array elements in turn. The method outputs the passed value, multiplies it by 3, and outputs it again. Finally, back in the calling program, the four numbers are output again. Figure 9-14 shows an execution of this program in a command-line environment.

Figure 9-13

`passarrayelement` Program

Figure 9-14

Output of `passarrayelement` Program

As you can see in Figure 9-14, the program displays the four original values, then passes each value to the `tripleTheValue()` method, where it is displayed, multiplied by 3, and displayed again. After the method executes four times, the logic returns to the main program where the four values are displayed again, showing that they are unchanged by the new assignments within `tripleTheValue()`. The `oneVal` variable is local to the `tripleTheValue()` method; therefore, any changes to it are not permanent and are not reflected in the array declared in the main program. Each `oneVal` variable in the `tripleTheValue()` method holds only a copy of the array element passed into the method, and the `oneVal` variable that holds each newly assigned, larger value exists only while the `tripleTheValue()` method is executing. In all respects, a single array element acts just like any single variable of the same type would.

Instead of passing a single array element to a method, you can pass an entire array as an argument. You can indicate that a method parameter must be an array by placing square brackets after the data type in the method's parameter list. When you pass an array to a method, changes you make to array elements within the method are permanent; that is, they are reflected in the original array that was sent to the method. Arrays, unlike simple built-in types, are **passed by reference** (Describes a method parameter that represents the item's memory address.) ; the method receives the actual memory address of the array and has access to the actual values in the array elements.

The name of an array represents a memory address, and the subscript used with an array name represents an offset from that address.

Simple nonarray variables usually are passed to methods by value. Many programming languages provide the means to pass variables by reference as well as by value. The syntax to accomplish this differs among the languages that allow it;

you will learn this technique when you study a specific language.

The program shown in Figure 9-15 creates an array of four numeric values. After the numbers are output, the entire array is passed to a method named `quadrupleTheValues()`. Within the method header, the parameter is declared as an array by using square brackets after the parameter type. Within the method, the numbers are output, which shows that they retain their values from the main program upon entering the method. Then the array values are multiplied by 4. Even though `quadrupleTheValues()` returns nothing to the calling program, when the program displays the array for the last time within the mainline logic, all of the values have been changed to their new quadrupled values. Figure 9-16 shows an execution of the program. Because arrays are passed by reference, the `quadrupleTheValues()` method "knows" the address of the array declared in the calling program and makes its changes directly to the original array that was declared in the calling program.

Figure 9-15

`Passentirearray` Program

Figure 9-16

Output of the `passentirearray` Program

When an array is a method parameter, the square brackets in the method header remain empty and do not hold a size. The array name that is passed is a memory address that indicates the start of the array. Depending on the language you are working in, you can control the values you use for a subscript to the array in different ways. In some languages, you might also want to pass a constant that indicates the array size to the method. In other languages, you can access the automatically created length field for the array. Either way, the array size itself is never implied when you use the array name. The array name only indicates the starting point from which subscripts will be used.

Two Truths & A Lie

Passing an Array to a Method

1. You can pass an entire array as a method's argument.

   T F

2. You can indicate that a method parameter must be an array by placing square brackets after the data type in the method's parameter list.

   T F

3. Arrays, unlike simple built-in types, are passed by value; the method receives a copy of the original array.

   T F

9-5 Overloading Methods

In most programming languages, some operators are overloaded. For example, a + between two values indicates addition, but a single + to the left of a value means the value is positive. The + sign has different meanings based on the arguments used with it.

Overloading a method is an example of **polymorphism** (The ability of a method to act appropriately depending on the context.) —the ability of a method to act appropriately according to the context. Literally, *polymorphism* means "many

forms.”

When you **overload a method** (To create multiple methods with the same name but different parameter lists.) , you write multiple methods with a shared name but different parameter lists. When you call an overloaded method, the language translator understands which version of the method to use based on the arguments used. For example, suppose that you create a method to output a message and the amount due on a customer bill, as shown in Figure 9-17. The method receives a numeric parameter that represents the customer’s balance and produces two lines of output. Assume that you also need a method that is similar to `printBill()`, except the new method applies a discount to the customer bill. One solution to this problem would be to write a new method with a different name—for example, `printBillWithDiscount()`. A downside to this approach is that a programmer who uses your methods must remember the names of each slightly different version. It is more natural for your method’s clients to use a single well-designed method name for the task of printing bills, but to be able to provide different arguments as appropriate. In this case, you can overload the `printBill()` method so that, in addition to the version that takes a single numeric argument, you can create a version that takes two numeric arguments—one that represents the balance and one that represents the discount rate. Figure 9-17 shows the two versions of the `printBill()` method.

Figure 9-17

Two Overloaded Versions of the `printbill()` Method

If both versions of `printBill()` are included in a program and you call the method using a single numeric argument, as in `printBill(custBalance)`, the first version of the method in Figure 9-17 executes. If you use two numeric arguments in the call, as in `printBill (custBalance, rate)`, the second version of the method executes.

If it suited your needs, you could provide more versions of the `printBill()` method, as shown in Figure 9-18. The first version accepts a numeric parameter that holds the customer's balance, and a string parameter that holds an additional message that can be customized for the bill recipient and displayed on the bill. For example, if a program makes a method call such as the following, this version of `printBill()` will execute:

Figure 9-18

Two Additional Overloaded Versions of the `printbill()` Method

The second version of the method in Figure 9-18 accepts three parameters, providing a balance, discount rate, and customized message. For example, the following method call would use this version of the method:

Overloading methods is never required in a program. Instead, you could create multiple methods with unique identifiers such as `printBill()` and

`printBillWithDiscountAndMessage()`. Overloading methods does not reduce your work when creating a program; you need to write each method individually. The advantage is provided to your method's clients; those who use your methods need to remember just one appropriate name for all related tasks.

In many programming languages, the `output` statement is actually an overloaded method that you call. Using a single name such as `output`, whether you want to output a number, a `string`, or any combination of the two, is convenient.

Even if you write two or more overloaded versions of a method, many program clients will use just one version. For example, suppose that you develop a bill-creating program that contains all four versions of the `printBill()` method just discussed, and then sell it to different companies. An organization that adopts your program and its methods might only want to use one or two versions of the method. You probably own many devices for which only some of the features are meaningful to you; for example, some people who own microwave ovens only use the *Popcorn* button or never use *Defrost*.

9-5a Avoiding Ambiguous Methods

When you overload a method, you run the risk of creating **ambiguous methods** (Methods that the compiler cannot distinguish because they have the same name and parameter types.) —a situation in which the compiler cannot determine which method to use. Every time you call a method, the compiler decides whether a suitable method exists; if so, the method executes, and if not, you receive an error message. For example, suppose that you write two versions of a `printBill()` method, as shown in the program in Figure 9-19. One version of the method is intended to accept a customer balance and a discount rate, and the other is intended to accept a customer balance and a discount amount expressed in dollars.

Figure 9-19

Program that Contains Ambiguous Method Call

Each of the two versions of `printBill()` in Figure 9-19 is a valid method on its own. However, when the two versions exist in the same program, a problem arises. When the main program calls `printBill()` using two numeric arguments, the compiler cannot determine which version to call. You might think that the version of the method with a parameter named `discountInDollars` would execute, because the method call uses the identifier `discountInDollars`. However, the compiler determines which version of a

method to call based on argument data types only, not their identifiers. Because both versions of the `printBill()` method could accept two numeric parameters, the compiler cannot determine which version to execute, so an error occurs and program execution stops.

An overloaded method is not ambiguous on its own—it becomes ambiguous only if you make a method call that matches multiple method signatures. In many languages, a program with potentially ambiguous methods will run without problems if you don't make any method calls that match more than one method.

Methods can be overloaded correctly by providing different parameter lists for methods with the same name. Methods with identical names that have identical parameter lists but different return types are not overloaded—they are ambiguous. For example, the following two method headers create ambiguity.

The compiler determines which version of a method to call based on parameter lists, not return types. When the method call `aMethod(17)` is made, the compiler will not know which of the two methods to execute because both possible choices take a numeric argument.

All the popular object-oriented programming languages support multiple numeric data types. For example, Java, C#, C++, and Visual Basic all support integer (whole number) data types that are different from floating-point (decimal place) data types. Many languages have even more specialized numeric types, such as signed and unsigned. Methods that accept different specific types are correctly overloaded.

Watch the video *Overloading Methods*.

Two Truths & A Lie

Overloading Methods

1. In programming, **overloading** (Supplying diverse meanings for a single identifier.) involves supplying diverse meanings for a single identifier.

   T F

2. When you overload a method, you write multiple methods with different names but identical parameter lists.

   T F

3. Methods can be overloaded correctly by providing different parameter lists for methods with the same name.

   T F

Chapter 9: Advanced Modularization Techniques: 9-6 Using Predefined Methods
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

9-6 Using Predefined Methods

All modern programming languages contain many methods that have already been written for programmers. Predefined methods might originate from several sources:

- Some prewritten methods are built into a language. For example, methods that perform input and output are usually predefined.
- When you work on a program in a team, each programmer might be assigned specific methods to create, and your methods will interact with methods written by others.
- If you work for a company, many standard methods may already have been written and you will be required to use them. For example, the company might have a standard method that displays its logo.

Predefined methods save you time and effort. For example, in most languages, displaying a message on the screen involves using a built-in method. When you want to display *Hello* on the command prompt screen in C#, you write the following:

In Java, you write:

In these statements, you can recognize `WriteLine()` and `println()` as method names because they are followed by parentheses; the parentheses hold an argument that represents the message to display. If these methods were not prewritten, you would have to know the low-level details of how to manipulate pixels on a screen to get the characters to display. Instead, by using the prewritten methods, you can concentrate on the higher-level task of displaying a useful and appropriate message.

In C#, the convention is to begin method names with an uppercase letter.

In Java, method names conventionally begin with a lowercase letter. The `WriteLine()` and `println()` methods follow their respective language's convention. The `WriteLine()` and `println()` methods are both overloaded in their respective languages. For example, if you pass a string to either method, the version of the method that accepts a string parameter executes, but if you pass a number, another version that accepts a numeric parameter executes.

Most programming languages also contain a variety of mathematical methods, such as those that compute a square root or the absolute value of a number. Other methods retrieve the current date and time from the operating system or select a random number to use in a game application. These methods were written as a convenience for you—computing a square root and generating random numbers are complicated tasks, so it is convenient to have methods already written, tested, and available when you need them. The names of the methods that perform these functions differ among programming languages, so you need to research the language's documentation to use them. For example, many of a language's methods are described in introductory programming language textbooks, and you can also find language documentation online.

Whether you want to use a predefined method or any other method, you should know the following four details:

- What the method does in general—for example, compute a square root.
- The method's name—for example, it might be `sqrt()`.
- The method's required parameters—for example, a square root method might require a single numeric parameter. There might be multiple overloaded versions of the method from which you can choose.
- The method's return type—for example, a square root method most likely returns a numeric value that is the square root of the argument passed to the method.

You do not need to know how the method is implemented—that is, how the instruction statements are written within it. Like all methods, you can use built-in methods without

worrying about their low-level implementation details.

Two Truths & A Lie

Using Predefined Methods

1. The name of a method that performs a specific function (such as generating a random number) is likely to be the same in various programming languages.

   T F

2. When you want to use a predefined method, you should know what the method does in general, along with its name, required parameters, and return type.

   T F

3. When you want to use a predefined method, you do not need to know how the method works internally to be able to use the method effectively.

   T F

9-7 Method Design Issues: Implementation Hiding, Cohesion, and Coupling

To design effective methods, you should consider several program qualities:

- You should employ implementation hiding; that is, a method's client should not need to understand a method's internal mechanisms.
- You should strive to increase cohesion.
- You should strive to reduce coupling.

9-7a Understanding Implementation Hiding

An important principle of modularization is the notion of **implementation hiding** (A programming principle that describes the encapsulation of method details.) , the encapsulation of method details. That is, when a program makes a request to a method, it doesn't know the details of how the method is executed. For example, when you make a restaurant reservation, you do not need to know how the reservation is actually recorded at the restaurant—perhaps it is written in a book, marked on a large chalkboard, or entered into a computerized database. The implementation details don't concern you as a patron, and if the restaurant changes its methods from one year to the next, the change does not affect your use of the reservation method—you still call and provide your name, a date, and a time. With well-written methods, using implementation hiding means that a method that calls another must know only the following:

- The name of the called method
- What type of information to send to the method
- What type of return data to expect from the method

In other words, the calling method needs to understand only the **interface to the method** (A method's return type, name, and arguments; the part of a method that a client sees and uses.) that is called. The interface is the only part of a method with which the method's **client** (A program or other method that uses a method.) (or method's caller) interacts. The program does *not* need to know how the method works internally. Additionally, if you substitute a new, improved method implementation but the interface to the method does not change, you won't need to make changes in any methods that call the altered method.

Programmers refer to hidden implementation details as existing in a **black box** (The analogy that programmers use to refer to the details of hidden methods.) —you can examine what goes in and what comes out, but not the details of how the method works inside.

9-7b Increasing Cohesion

When you begin to design computer programs, it is difficult to decide how much to put into a method. For example, a process that requires 40 instructions can be contained in a single 40- instruction method, two 20-instruction methods, five 8-instruction methods, or many other combinations. In most programming languages, any of these combinations is allowed; you can write a program that executes and produces correct results no matter how you divide the individual steps into methods. However, placing too many or too few instructions in a single method makes a program harder to follow and reduces flexibility.

To help determine the appropriate division of tasks among methods, you want to analyze each method's **cohesion** (A measure of how a method's internal statements are focused to accomplish the method's purpose.) , which refers to how the internal statements of a method serve to accomplish the method's purpose. In highly cohesive methods, all the operations are related, or "go together." Such methods are **functionally cohesive** (The extent to which all operations in a method contribute to the performance of only one task.) —all their operations contribute to the performance of a single task. Functionally cohesive methods usually are more reliable than those that have low cohesion; they are considered stronger, and they make programs easier to write, read, and maintain.

For example, consider a method that calculates gross pay. The method receives parameters that define a worker's pay rate and number of hours worked. The method computes gross pay and displays it. The cohesion of this method is high because each of its instructions contributes to one task—computing gross pay. If you can write a sentence describing what a method does using only two words—for example, *Compute gross*, *Cube value*, or *Display record*—the method is probably functionally cohesive.

You might work in a programming environment that has a rule such as *No method will be longer than can be printed on one page* or *No method will have more than 30 lines of code*. The rule maker is trying to achieve more cohesion, but such rules are arbitrary. A two-line method could have low cohesion and—although less likely—a 40-line method might have high cohesion. Because good, functionally cohesive methods perform only one task, they tend to be short. However, the issue is not size. If it takes 20 statements to perform one task within a method, the method is still cohesive.

Most programmers do not consciously make decisions about cohesiveness for each method they write. Rather, they develop a "feel" for what types of tasks belong together, and for which subsets of tasks should be diverted to their own methods.

Chapter 9: Advanced Modularization Techniques: 9-7c Reducing Coupling
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

9-7c Reducing Coupling

Coupling (A measure of the strength of the connection between two program methods.) is a measure of the strength of the connection between two program methods; it expresses the extent to which information is exchanged by methods. Coupling is either tight or loose, depending on how much one method relies on information from another. **Tight coupling** (A problem that occurs when methods excessively depend on each other; it makes programs more prone to errors.) , which occurs when methods depend on each other excessively, makes programs more prone to errors. With tight coupling, you have many data paths to keep track of, many chances for bad data to pass from one method to another, and many chances for one method to alter information needed by another method. **Loose coupling** (A relationship that occurs when methods do not depend on others.) occurs when methods do not depend on others. In general, you want to reduce coupling as much as possible because connections between methods make them more difficult to write, maintain, and reuse.

Imagine four cooks wandering in and out of a kitchen while preparing a stew. If each is allowed to add seasonings at will without the knowledge of the other cooks, you could end up with a culinary disaster. Similarly, if four payroll program methods can alter your gross pay without the "knowledge" of the other methods, you could end up with a financial disaster. A program in which several methods have access to your gross pay figure has methods that are tightly coupled. A superior program would control access to the payroll figure by limiting its passage to methods that need it.

You can evaluate whether coupling between methods is loose or tight by looking at how methods share data.

- Tight coupling occurs when methods have access to the same globally defined variables. When one method changes the value stored in a variable, other methods are affected. Because you should avoid tight coupling, all the examples in this book avoid using global variables. However, be aware that you might see them used in programs written by others.
- Loose coupling occurs when a copy of data that must be shared is passed from one method to another. That way, the sharing of data is always purposeful—variables must be explicitly passed to and from methods that use them. The loosest (best) methods pass single arguments rather than many variables or entire records, if possible.

Additionally, there is a time and a place for shortcuts. If you need a result from spreadsheet data in a hurry, you can type two values and take a sum rather than creating a formula with proper cell references. If a memo must go out in five minutes, you don't have time to change fonts or add clip art with your word processor. Similarly, if you need a quick programming result, you might very well use cryptic variable names, tight coupling, and minimal cohesion. When you create a professional application, however, you should keep professional guidelines in mind.

Two Truths & A Lie

Method Design Issues: Implementation Hiding, Cohesion, and Coupling

1. A calling method must know the interface to any method it calls.

   T F

2. You should try to avoid loose coupling, which occurs when methods do not depend on others.

   T F

3. Functional cohesion occurs when all operations in a method contribute to the performance of only one task.

```
T F
```


9-8 Understanding Recursion

Recursion (A programming event that occurs when a method is defined in terms of itself.) occurs when a method is defined in terms of itself. A method that calls itself is a **recursive method** (A method that calls itself.) . Some programming languages do not allow a method to call itself, but those that do can be used to create recursive methods that produce interesting effects.

Figure 9-20 shows a simple example of recursion. The program calls an `infinity()` method, which displays *Help!* and calls itself again (see the shaded statement). The second call to `infinity()` displays *Help!* and generates a third call. The result is a large number of repetitions of the `infinity()` method. The output is shown in Figure 9-21.

Figure 9-20

A Program That Calls a Recursive Method

Figure 9-21

Output of Program in Figure 9-20

Every time you call a method, the address to which the program should return at the

completion of the method is stored in a memory location called the **stack** (A memory location that holds the memory addresses to which method calls should return.) . When a method ends, the address is retrieved from the stack and the program returns to the location from which the method call was made. For example, suppose that a program calls `methodA()` and that `methodA()` calls `methodB()`. When the program calls `methodA()`, a return address is stored in the stack, and then `methodA()` begins execution. When `methodA()` calls `methodB()`, a return address in `methodA()` is stored in the stack and `methodB()` begins execution. When `methodB()` ends, the last entered address is retrieved from the stack and program control returns to complete `methodA()`. When `methodA()` ends, the remaining address is retrieved from the stack and program control returns to the main program method to complete it.

Like all computer memory, the stack has a finite size. When the program in Figure 9-20 calls the `infinity()` method, the stack receives so many return addresses that it eventually overflows. The recursive calls will end after an excessive number of repetitions and the program issues error messages.

Of course, there is no practical use for an infinitely recursive program. Just as you must be careful not to create endless loops, when you write useful recursive methods you must provide a way for the recursion to stop eventually.

Using recursion successfully requires a thorough understanding of looping. You learned about loops in Chapter 5. An everyday example of recursion is printed on shampoo bottles: *Lather*, *rinse*, *repeat*.

Figure 9-22 shows an application that uses recursion productively. The program calls a recursive method that computes the sum of every integer from 1 up to and including the method's argument value. For example, the sum of every integer up to and including 3 is 1 + 2 + 3, or 6, and the sum of every integer up to and including 4 is 1 + 2 + 3 + 4, or 10.

Figure 9-22

Program that Uses a Recursive `cumulativesum()` Method

When thinking about cumulative summing relationships, remember that the sum of all the integers up to and including any number is that number plus the sum of the integers for the next lowest number. In other words, consider the following:

- The sum of the digits from 1, up to and including 1, is simply 1.
- The sum of the digits from 1 through 2 is the previous sum, plus 2.
- The sum of the digits from 1 through 3 is the previous sum, plus 3.
- The sum of the digits 1 through 4 is the previous sum, plus 4.
- And so on.

The recursive `cumulativeSum()` method in Figure 9-22 uses this knowledge. For each `number`, its cumulative sum consists of itself plus the cumulative sum of all the previous lesser numbers.

The program in Figure 9-22 calls the `cumulativeSum()` method 10 times in a loop to show the cumulative sum of every integer from 1 through 10. Figure 9-23 shows the output.

Figure 9-23

Output of Program in Figure 9-22

If you examine Figures 9-22 and 9-23 together, you can see the following:

- When 1 is passed to the `cumulativeSum()` method, the `if` statement within the method determines that the argument is equal to 1, `returnVal` becomes 1, and 1 is returned for output.
- On the next pass through the loop, 2 is passed to the `cumulativeSum()` method. When the method receives 2 as an argument, the `if` statement within the method is false, and `returnVal` is set to 2 plus the value of `cumulativeSum(1)`. This second call to `cumulativeSum()` using 1 as an argument returns a 1, so when the method ends, it returns 2 + 1, or 3.
- On the third pass through the loop within the calling program, 3 is passed to the `cumulativeSum()` method. When the method receives 3 as an argument, the `if` statement within the method is false and the method returns 3 plus the value of `cumulativeSum(2)`. The value of this call is 2 plus `cumulativeSum(1)`. The value of `cumulativeSum(1)` is 1. Ultimately, `cumulativeSum(3)` is 3 + 2 + 1.

Many sophisticated programs that operate on lists of items use recursive processing. However, following the logic of a recursive method can be difficult, and programs that use recursion are sometimes error-prone and hard to debug. Because such programs also can be hard for others to maintain, some business organizations forbid their programmers from using recursive logic in company programs. Many of the problems solved by recursive methods can be solved in a more straightforward way. For example, examine the program in Figure 9-24. This program produces the same result as the previous recursive program, but in a more straightforward fashion.

Figure 9-24

Nonrecursive Program that Computes Cumulative Sums

A humorous illustration of recursion is found in this sentence: “In order to understand recursion, you must first understand recursion.” A humorous dictionary entry is “Recursion: See Recursion.” These examples contain an element of truth, but useful recursion algorithms always have a point at which the infinite loop is exited. In other words, the base case, or exit case, is always reached at some point.

Watch the video *Recursion*.

Two Truths & A Lie

Understanding Recursion

1. A method that calls itself is a recursive method.

   T F

2. Every time you call a method, the address to which the program should return at the completion of the method is stored in a memory location called the stack.

   T F

3. Following the logic of a recursive method is usually much easier than following the logic of an ordinary program, so recursion makes debugging easier.

   T F

9-9 Chapter Review

9-9a Chapter Summary

- A method is a program module that contains a series of statements that carry out a task. Any program can contain an unlimited number of methods, and each method can be called an unlimited number of times. A method must include a header, a body, and a `return` statement that marks the end of the method. Variables and constants are in scope within, or local to, only the method within which they are declared.

- When you pass a data item into a method, it is an argument to the method. When the method receives the data item, it is called a parameter. When you write the declaration for a method that can receive parameters, you must include the data type and a local name for each parameter within the method declaration's parentheses. You indicate that a method requires multiple arguments by listing their data types and local identifiers within the method header's parentheses. You can pass multiple arguments to a called method by listing the arguments within the method call and separating them with commas. When you call a method, the arguments you send to the method must match in order—both in number and in type—the parameters listed in the method declaration.

- When a method returns a value, the method must have a return type. A method's return type indicates the data type of the value that the method will send back to the location where the method call was made. The return type also is known as a method's type, and is indicated in front of the method name when the method is defined. When a method returns a value, you usually want to use the returned value in the calling method, although this is not required.

- You can pass a single array element to a method in exactly the same manner you would pass a variable or constant. Additionally, you can pass an entire array to a method. You can indicate that a method parameter must be an array by placing

square brackets after the data type in the method's parameter list. When you pass an array to a method, it is passed by reference; that is, the method receives the actual memory address of the array and has access to the actual values in the array elements.

- When you overload a method, you write multiple methods with a shared name but different parameter lists. The compiler understands your meaning based on the arguments you use when you call the method. Overloading a method introduces the risk of creating ambiguous methods—a situation in which the compiler cannot determine which method to use. Methods can be overloaded correctly by providing different parameter lists for methods with the same name.
- All modern programming languages contain many built-in, prewritten methods to save you time and effort.
- With well-written methods, the implementation is hidden. To use a method, you need only know the name of the called method, what type of information to send to the method, and what type of return data to expect from the method. When writing methods, you should strive to achieve high cohesion and loose coupling.
- Recursion occurs when a method is defined in terms of itself. Following the logic of a recursive method is difficult, and programs that use recursion are sometimes error-prone and hard to debug.

Chapter Review

9-9b Key Terms

method (A series of statements that carry out a task.)

method header (A program component that precedes a method's implementation; the header includes the method identifier and possibly other necessary information.)

method body (The set of all the statements in a method.)

implementation (The body of a method; the statements that carry out the tasks of a method.)

method return statement (A statement that marks the end of the method and identifies the point at which control returns to the calling method.)

local (Describes variables that are declared within the method that uses them.)

global (Describes variables that are known to an entire program.)

argument to the method (A value passed to a method in the method call.)

parameter to the method (A data item passed into a method from the outside.)

parameter list (All the data types and parameter names that appear in a method header.)

signature (A method's name and parameter list.)

passed by value (Describes a variable that has a copy of its value sent to a method and stored in a new memory location accessible to the method.)

actual parameters (The arguments in a method call.)

formal parameters (The variables in a method declaration that accept values from the actual parameters.)

return type (The data type for any value a method returns.)

void method (A method that returns no value.)

method's type (The data type of a method's return value.)

overhead (All the resources and time required by an operation.)

IPO chart (A program development tool that delineates input, processing, and output tasks.)

passed by reference (Describes a method parameter that represents the item's memory address.)

overloading (Supplying diverse meanings for a single identifier.)

polymorphism (The ability of a method to act appropriately depending on the context.)

overload a method (To create multiple methods with the same name but different parameter lists.)

ambiguous methods (Methods that the compiler cannot distinguish because they have the same name and parameter types.)

implementation hiding (A programming principle that describes the encapsulation of method details.)

interface to the method (A method's return type, name, and arguments; the part of a method that a client sees and uses.)

client (A program or other method that uses a method.)

black box (The analogy that programmers use to refer to the details of hidden methods.)

cohesion (A measure of how a method's internal statements are focused to accomplish

the method's purpose.)

functionally cohesive (The extent to which all operations in a method contribute to the performance of only one task.)

Coupling (A measure of the strength of the connection between two program methods.)

Tight coupling (A problem that occurs when methods excessively depend on each other; it makes programs more prone to errors.)

Loose coupling (A relationship that occurs when methods do not depend on others.)

Recursion (A programming event that occurs when a method is defined in terms of itself.)

recursive method (A method that calls itself.)

stack (A memory location that holds the memory addresses to which method calls should return.)

Chapter Review

9-9c Review Questions

1. Which of the following is true?
    a. A program can call one method at most.
    b. A program can contain a method that calls another method.
    c. A method can contain one or more other methods.
    d. All of the above are true.
2. Which of the following must every method have?
    a. a header
    b. a parameter list
    c. a return value
    d. all of the above

3. Which of the following is most closely related to the concept of *local*?

    a. abstract

    b. object-oriented

    c. in scope

    d. program level

4. Although the terms *parameter* and *argument* are closely related, the difference is that *argument* refers to ____ .

    a. a passed constant

    b. a value in a method call

    c. a formal parameter

    d. a variable that is local to a method

5. A method's interface is its ____ .

    a. signature

    b. return type

    c. identifier

    d. parameter list

6. When you write the declaration for a method that can receive a parameter, which of the following must be included in the method declaration?

    a. the name of the argument that will be used to call the method

    b. a local name for the parameter

    c. the return value for the method

    d. two of the above

7. When you use a variable name in a method call, it ____ as the variable in the method header.

    a. can have the same name

    b. cannot have the same name

c. must have the same name

d. cannot have the same data type

8. Assume that you have written a method with the header `void myMethod(num a, string b)`. Which of the following is a correct method call?

   a. `myMethod(12)`

   b. `myMethod(12, "Hello")`

   c. `myMethod("Goodbye")`

   d. It is impossible to tell.

9. Assume that you have written a method with the header `num yourMethod(string name, num code)`. The method's type is ____ .

   a. `num`

   b. `string`

   c. `num` and `string`

   d. `void`

10. Assume that you have written a method with the header `string myMethod(num score, string grade)`. Also assume that you have declared a numeric variable named `test`. Which of the following is a correct method call?

    a. `myMethod()`

    b. `myMethod(test)`

    c. `myMethod(test, test)`

    d. `myMethod(test,"A")`

11. The value used in a method's `return` statement must ____ .

    a. be numeric

    b. be a variable

    c. match the data type used before the method name in the header

    d. two of the above

12. When a method receives a copy of the value stored in an argument used in the method call, it means the variable was ____ .

    a. unnamed

    b. passed by value

    c. passed by reference

    d. assigned its original value when it was declared

13. A void method ____ .

    a. contains no statements

    b. requires no parameters

    c. returns nothing

    d. has no name

14. When an array is passed to a method, it is ____ .

    a. passed by reference

    b. passed by value

    c. unnamed in the method

    d. unalterable in the method

15. When you overload a method, you write multiple methods with the same ____ .

    a. name

    b. parameter list

    c. number of parameters

    d. return type

16. A program contains a method with the header `num calculateTaxes(num amount, string name)`. Which of the following methods can coexist in the same program with no possible ambiguity?

    a. `num calculateTaxes(string name, num amount)`

    b. `string calculateTaxes(num money, string taxpayer)`

c. `num calculateTaxes(num annualPay, string taxpayerId)`

d. All of these can coexist without ambiguity.

17. Methods in the same program with identical names and identical parameter lists are ____ .

    a. overloaded

    b. overworked

    c. overwhelmed

    d. ambiguous

18. Methods in different programs with identical names and identical parameter lists are ____ .

    a. overloaded

    b. illegal

    c. both of the above

    d. none of the above

19. The notion of ____ most closely describes the way a calling method is not aware of the statements within a called method.

    a. abstraction

    b. object-oriented

    c. implementation hiding

    d. encapsulation

20. Programmers should strive to ____ .

    a. increase coupling

    b. increase cohesion

    c. both of the above

    d. neither a nor b

Chapter Review

9-9d Exercises

1. Create an IPO chart for each of the following methods:
    a. The method that calculates the amount owed on a restaurant check, including tip
    b. The method that calculates your yearly education-related expenses
    c. The method that calculates your annual housing expenses, including rent or mortgage payment and utilities
2. Create the logic for a program that continuously prompts the user for a number of dollars until the user enters 0. Pass each entered amount to a conversion method that displays a breakdown of the passed amount into the fewest bills; in other words, the method calculates the number of 20s, 10s, 5s, and 1s needed.
3. Create the logic for a program that calculates and displays the amount of money you would have if you invested $5000 at 2 percent simple interest for one year. Create a separate method to do the calculation and return the result to be displayed.
4. Create the logic for a program that accepts input values for the projected cost of a vacation and the number of months until vacation. Pass both values to a method that displays the amount you must save per month to achieve your goal.
5. a. Create the logic for a program that performs arithmetic functions. Design the program to contain two numeric variables, and prompt the user for values for the variables. Pass both variables to methods named `sum()` and `difference()`. Create the logic for the methods `sum()` and `difference()`; they compute the sum of and difference between the values of two arguments, respectively. Each method should perform the appropriate computation and display the results.
    b. Add a method named `product()` to the program in Exercise 5a. The

`product()` method should compute the result when multiplying two numbers, but not display the answer. Instead, the method should return the answer to the calling program, which displays the answer.

6. Create the logic for a program that continuously prompts a user for a numeric value until the user enters 0. The application passes the value in turn to a method that computes the sum of all the whole numbers from 1 up to and including the entered number, and to a method that computes the product of all the whole numbers up to and including the entered number.

7. Create the logic for a program that calls a method that computes the final price for a sales transaction. The program contains variables that hold the price of an item, the salesperson's commission expressed as a percentage, and the customer discount expressed as a percentage. Create a `calculatePrice()` method that determines the final price and returns the value to the calling method. The `calculatePrice()` method requires three arguments: product price, salesperson commission rate, and customer discount rate. A product's final price is the original price plus the commission amount minus the discount amount. The customer discount is taken as a percentage of the total price after the salesperson commission has been added to the original price.

8. Create the logic for a program that continuously prompts the user for two numeric values that represent the dimensions of a room in feet. Include two overloaded methods that compute the room's area. One method takes two numeric parameters and calculates the area by multiplying the parameters. The other takes a single numeric parameter, which is squared to calculate area. Each method displays its calculated result. Accept input and respond as follows:

    - When the user enters zero for the first value, end the program.
    - If the user enters a negative number for either value, continue to reprompt the user until the value is not negative.
    - If both numbers entered are greater than 0, call the method version that accepts two parameters and pass it both values.
    - If the second value is zero, call the version of the method that accepts just one parameter and pass it the nonzero value.

9. a. Plan the logic for an insurance company program to determine policy premiums. The program continuously prompts the user for an insurance policy number. When the user enters an appropriate sentinel value, end the program. Call a method that prompts each user for the

type of policy needed—health or auto. While the user's response does not indicate health or auto, continue to prompt the user. When the value is valid, return it from the method. Pass the user's response to a new method where the premium is set and returned—$550 for a health policy or $225 for an auto policy. Display the results for each policy.

b. Modify Exercise 9a so that the premium-setting method calls one of two additional methods—one that determines the health premium or one that determines the auto premium. The health insurance method asks users whether they smoke; the premium is $550 for smokers and $345 for nonsmokers. The auto insurance method asks users to enter the number of traffic tickets they have received in the last three years. The premium is $225 for drivers with three or more tickets, $190 for those with one or two tickets, and $110 for those with no tickets. Each of these two methods returns the premium amount to the calling method, which returns the amount to be displayed.

10. Create the logic for a program that calculates the due date for a bill. Prompt the user for the month, day, and year a bill is received, and pass the data to a method that displays slashes between the parts of the date—for example, *6/24/2013*. Then pass the parts of the date to a method that calculates the bill's due date, which is 10 days after receipt. (A due date might be in the next month or even the next year.) This method displays the due date with slashes by calling the display method.

11. Create the logic for a program that computes hotel guest rates at Cornwall's Country Inn. Include two overloaded methods named `computeRate()`. One version accepts a number of days and calculates the rate at $99.99 per day. The other accepts a number of days and a code for a meal plan. If the code is *A*, three meals per day are included, and the price is $169.00 per day. If the code is *C*, breakfast is included, and the price is $112.00 per day. All other codes are invalid. Each method returns the rate to the calling program where it is displayed. The main program asks the user for the number of days in a stay and whether meals should be included; then, based on the user's response, the program either calls the first method or prompts for a meal plan code and calls the second method.

12. Create the logic for a program that prompts a user for five numbers and stores them in an array. Pass the array to a method that reverses the order of the numbers. Display the reversed numbers in the main program.

13. Create the logic for a program that prompts a user for six numbers and stores them in an array. Pass the array to a method that calculates the arithmetic

average of the numbers and returns the value to the calling program. Display each number and how far it is from the arithmetic average. Continue to prompt the user for additional sets of six numbers until the user wants to quit.

14. The Information Services Department at the Springfield Library has created methods with the following signatures:

Table 9-1

Library Methods

Signature	Description
`num getNumber(num high, num low)`	Prompts the user for a number, and continues to prompt until the number falls between designated high and low limits; returns a valid number
`string getCharacter()`	Prompts the user for a character string and returns the entered string
`num lookUpISBN(string title)`	Accepts the title of a book and returns the ISBN; returns a 0 if the book cannot be found
`string lookUpTitle(num isbn)`	Accepts the ISBN of a book and returns a title; returns a space character if the book cannot be found
`string isBookAvailable(num isbn)`	Accepts an ISBN, searches the library database, and returns `"Y"` or `"N"` indicating whether the book is currently available

a. Design an interactive program that does the following, using the

prewritten methods whenever they are appropriate.

- Prompt the user for and read a library card number, which must be between 1000 and 9999.
- Prompt the user for and read a search option—1 to search for a book by ISBN, 2 to search for a book by title, and 3 to quit. If the entry is invalid, repeat the request.
- While the user does not enter 3, prompt for an ISBN or title based on the user's previous selection. If the user enters an ISBN, get and display the book's title and ask the user to enter a `"Y"` or `"N"` to confirm whether the title is correct.
- If the user has entered a valid ISBN or a title that matches a valid ISBN, check whether the book is available, and display an appropriate message for the user.
- The user can continue to search for books until he or she enters 3 as the search option.

b. Develop the logic that implements each of the methods in Exercise 14a.

15. Each of the programs in Figure 9-25 uses a recursive method. Try to determine the output in each case.

Figure 9-25

Problems for Exercise 15

Find the Bugs

16. Your downloadable files for Chapter 9 include DEBUG09-01.txt, DEBUG09-

02.txt, and DEBUG09-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

17. In the Game Zone sections of Chapters 6 and 8, you designed the logic for a quiz that contains questions about a topic of your choice. Now, modify the program so it contains an array of five multiple-choice quiz questions related to the topic of your choice. Each question contains four answer choices. Also create a parallel array that holds the correct answer to each question—A, B, C, or D. In turn, pass each question to a method that displays the question and accepts the player's answer. If the player does not enter a valid answer choice, force the player to reenter the choice. Return the user's valid (but not necessarily correct) answer to the main program. After the user's answer is returned to the main program, pass it and the correct answer to a method that determines whether the values are equal and displays an appropriate message. After the user answers all five questions, display the number of correct and incorrect answers that the user chose.

18. In the Game Zone section of Chapter 6, you designed the logic for the game Hangman, in which the user guesses letters in a hidden word. Improve the game to store an array of 10 words. One at a time, pass each word to a method that allows the user to guess letters continuously until the game is solved. The method returns the number of guesses it took to complete the word. Store the number in an array before returning to the method for the next word. After all 10 words have been guessed, display a summary of the number of guesses required for each word as well as the average number of guesses per word.

Up for Discussion

19. One advantage to writing a program that is subdivided into methods is that such a structure allows different programmers to write separate methods, thus dividing the work. Would you prefer to write a large program by yourself, or to work on a team in which each programmer produces one or more methods? Why?

20. In this chapter, you learned that hidden implementations are often said to exist in a black box. What are the advantages and disadvantages to this approach in both programming and real life?