



Programming Logic and Design

Ninth Edition

Chapter 9

Advanced Modularization Techniques



Objectives

In this chapter, you will learn about:

- The parts of a method
- 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



The Parts of a Method

- **Method**

- A program module that contains a series of statements that carry out a task
- Invoke or call a method from another program or method
- Any program can contain an unlimited number of methods
- Each method can be called an unlimited number of times
- Calling program or method is called the method's **client**

The Parts of a Method (continued)

- Method must include
 - **Method header** (also called the declaration or definition)
 - **Method body**
 - Contains the **implementation** (Statements that carry out the tasks)
 - **Method return statement** (Returns control to the calling method)
- Variables and constants
 - **Local**: declared in a method
 - **Global**: known to all program modules

Using Methods with No Parameters

```

C:\WINDOWS\system32\cmd.exe
1 - English or 2 - Espanol >> 2
Please enter your weight in pounds >> 150
Your weight on the moon would be 24.9
Press any key to continue . . .
  
```

Figure 9-2 Output of moon weight calculator program in Figure 9-1

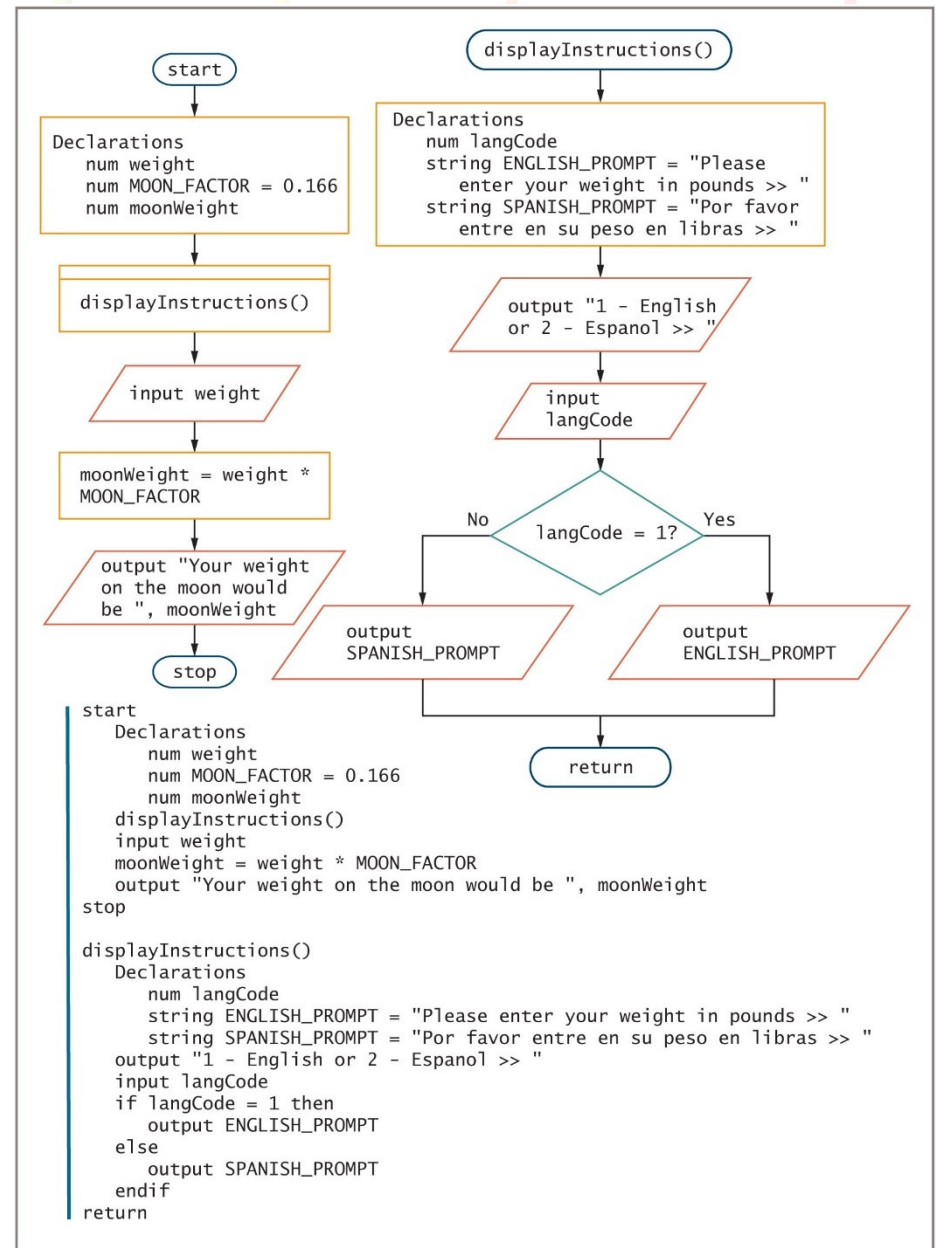


Figure 9-1 A program that calculates the user's weight on the moon

Using Methods with No Parameters

(continued -1)

- When methods must share data
 - Pass the data into and return the data out of methods
- When you call a method from a program, you must know four things:
 - What the method does
 - Name of the called method
 - Type of information to send to the method, if any
 - Type of return data to expect from the method, if any

Creating Methods that Require Parameters

- **Argument to the method**
 - Pass a data item into a method from a calling program
- **Parameter to the method**
 - Method receives the data item
- When a method receives a parameter, you must provide a **parameter list** that includes:
 - The type of the parameter
 - The local name for the parameter
- **Signature**
 - Method's name and parameter list

Creating Methods that Require Parameters

(continued -1)

- Improve the moon weight program by making the final output more user-friendly
- Several approaches
 - Rewrite the program without including any methods
 - Retain the `displayInstructions()` method, but make the `langCode` variable global
 - Retain the `displayInstructions()` method as is, but add a section to the main program that also asks the user for a preferred language

Creating Methods that Require Parameters

(continued -2)

- **Passed by value**
 - A copy of a value is sent to the method and stored in a new memory location accessible to the method
- Each time a method executes, parameter variables listed in the method header are redeclared

Creating Methods that Require Parameters

(continued -3)

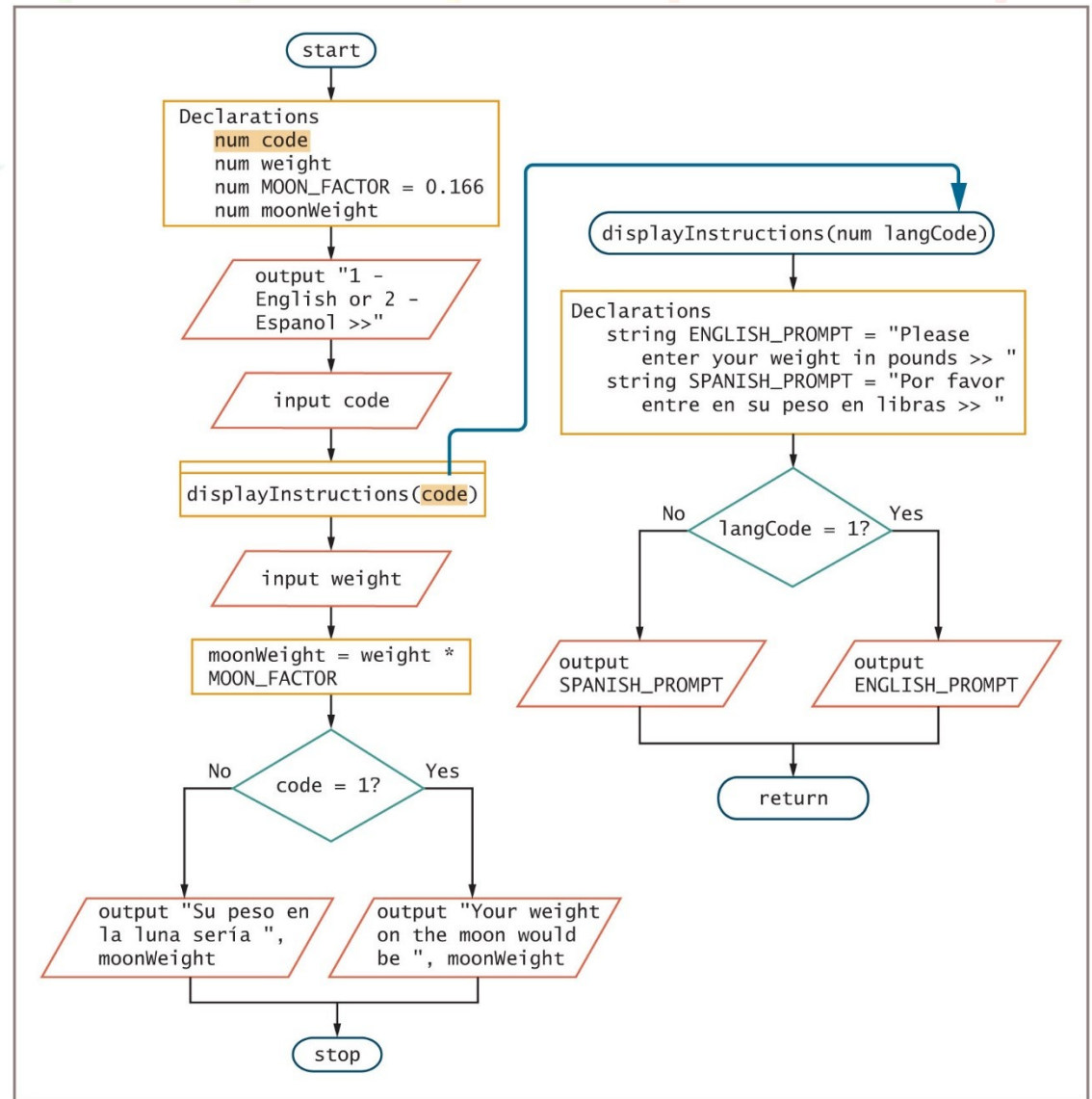


Figure 9-3 Moon weight program that passes an argument to a method (continues)

Creating Methods that Require Parameters

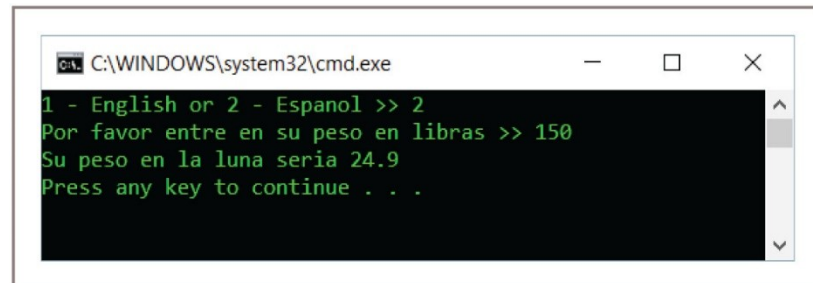
(continued -4)

(continued)

```
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 seria ", 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 9-3 Moon weight program that passes an argument to a method

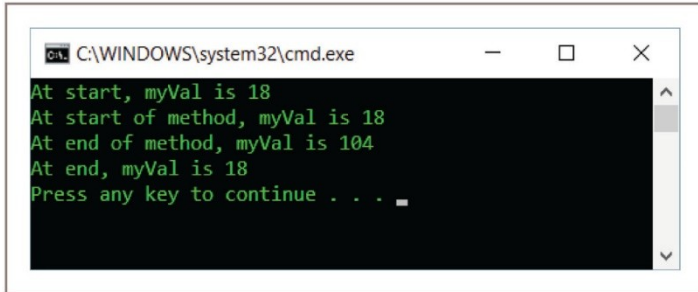


```
C:\WINDOWS\system32\cmd.exe
1 - English or 2 - Espanol >> 2
Por favor entre en su peso en libras >> 150
Su peso en la luna seria 24.9
Press any key to continue . . .
```

Figure 9-4 Typical execution of moon weight program in Figure 9-3

Creating Methods that Require Parameter

S (continued -5)



```
C:\WINDOWS\system32\cmd.exe
At start, myVal is 18
At start of method, myVal is 18
At end of method, myVal is 104
At end, myVal is 18
Press any key to continue . . . _
```

Figure 9-6 Execution of the program in Figure 9-5

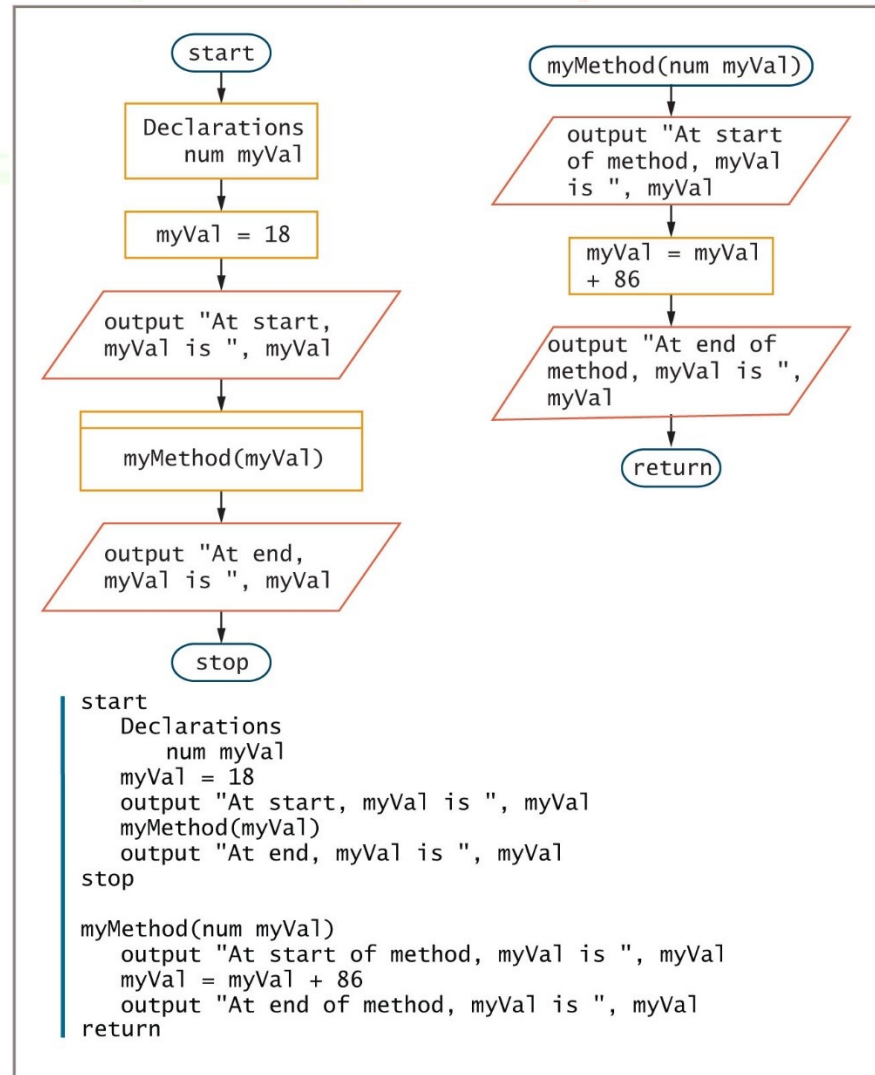


Figure 9-5 A program that calls a method in which the argument and parameter have the same identifier

Creating Methods that Require Multiple Parameters

- Methods can require more than one parameter
 - List the arguments within the method call, separated by commas
 - List a data type and local identifier for each parameter within the method header's parentheses
 - Separate each declaration with a comma
 - The data type must be repeated with each parameter
 - Arguments sent are called actual parameters
 - Variables that accept the parameters in the method are called formal parameters

Creating Methods that Require Multiple Parameters

(continued -1)

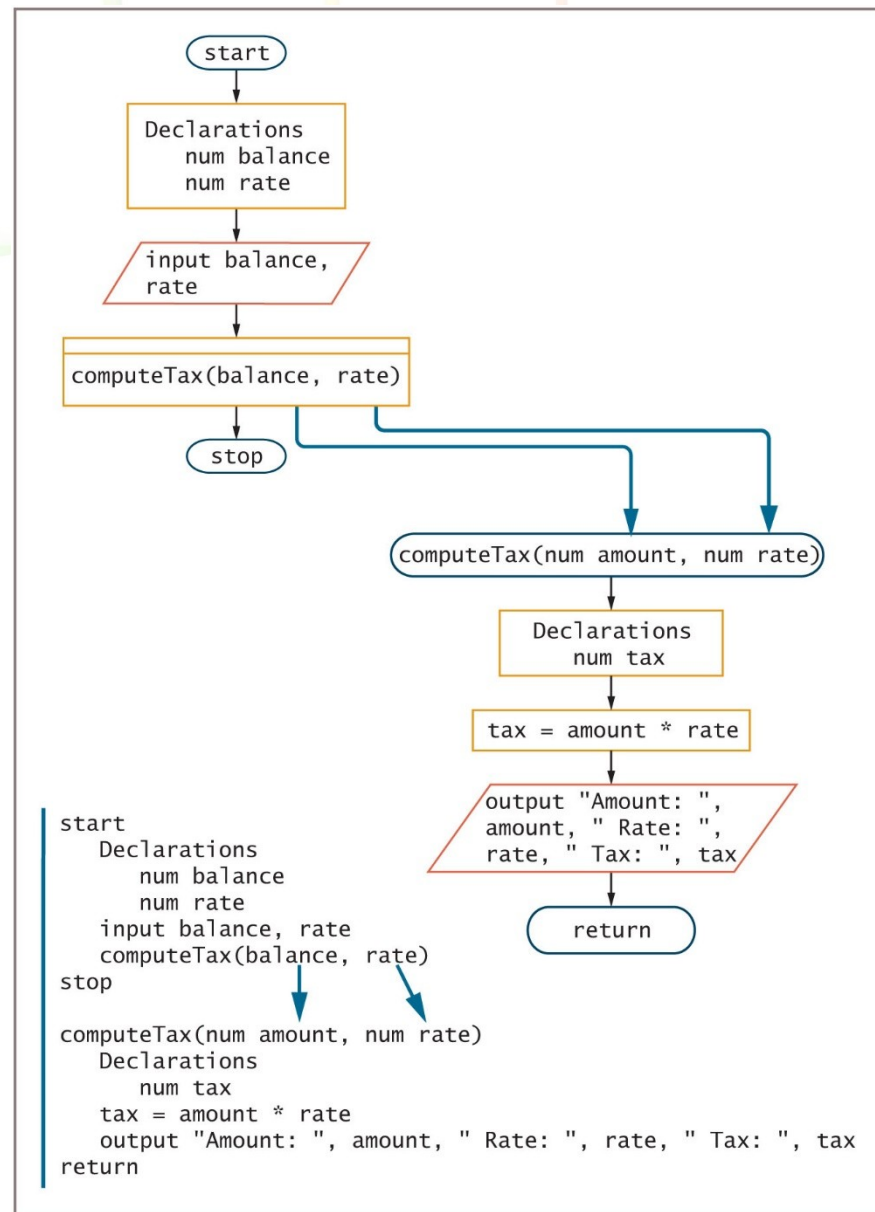


Figure 9-7 A program that calls a `computeTax()` method that requires two parameters

Creating Methods that Return a Value

- A variable declared within a method ceases to exist when the method ends
 - Goes out of scope
- To retain a value that exists when a method ends, return the value from the method back to the calling method
- When a method returns a value, the method must have a **return type** that matches the data type of the value that is returned

Creating Methods that Return a Value

(continued -1)

- Return type for a method
 - Indicates the data type of the value that the method will send back
 - Can be any type
 - Also called **method's type**
 - Listed in front of the method name when the method is defined
- Method can also return nothing
 - Return type `void`
 - **Void method**

Creating Methods that Return a Value

(continued -2)

- Example: `num getHoursWorked()`
 - Returns a numeric value
- Usually, you want to use the returned value in the calling method
 - Not required
 - Store in variable or use directly without storing
 - output `"Hours worked is ", getHoursWorked()`

Creating Methods that Return a Value

(continued -3)

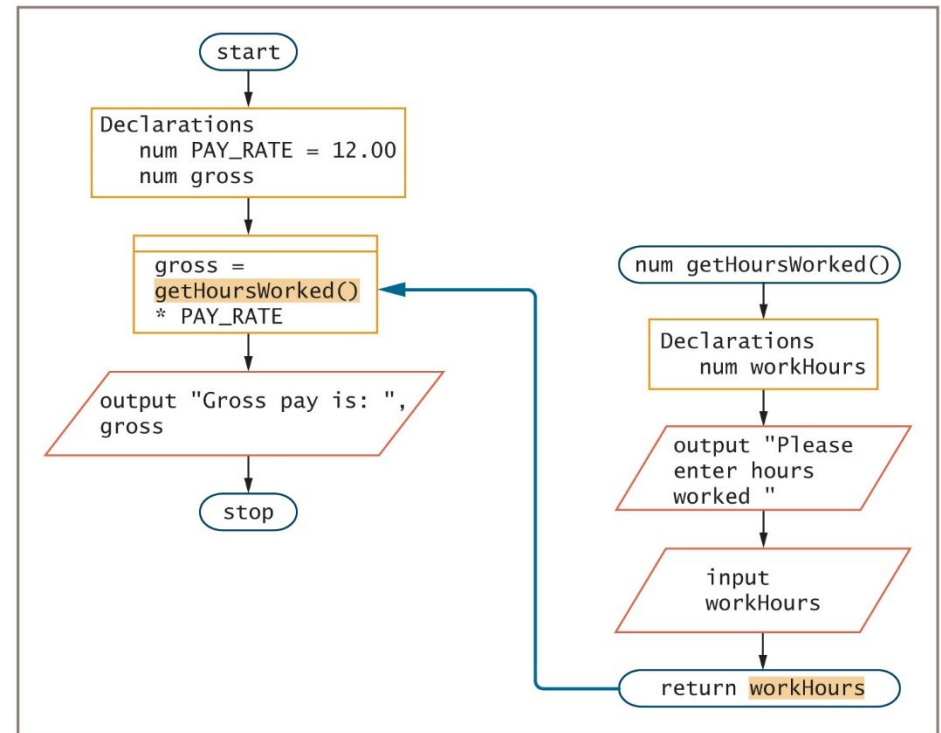
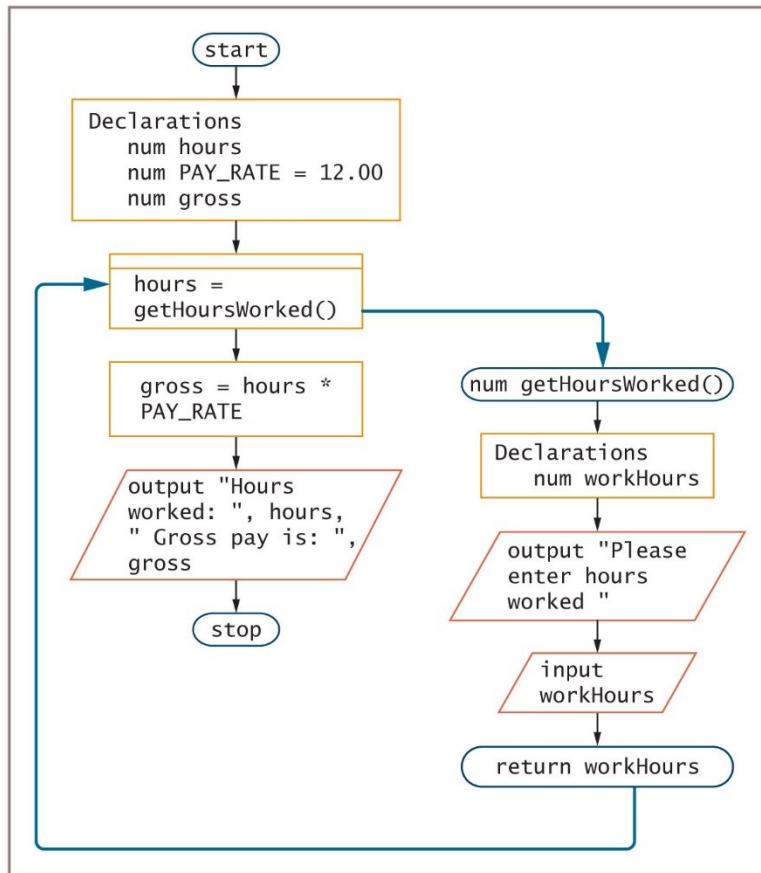


Figure 9-9 A program that uses a method's returned value without storing it

Figure 9-8 A payroll program that calls a method that returns a value

Creating Methods that Return a Value

(continued -4)

- Technically, you are allowed to include multiple return statements in a method
 - It's not recommended
 - Violates structured logic

Creating Methods that Return a Value

(continued -5)

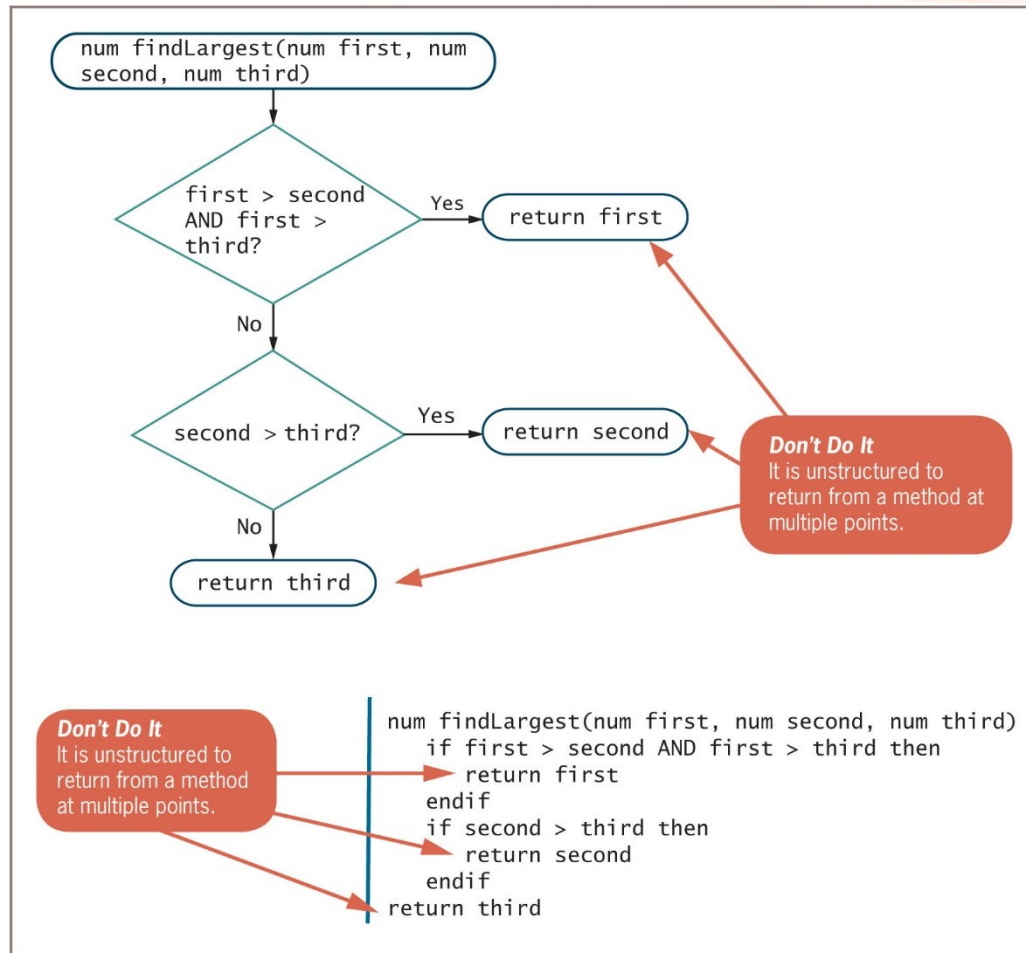


Figure 9-10 Unstructured approach to returning one of several values

Creating Methods that Return a Value

(continued -6)

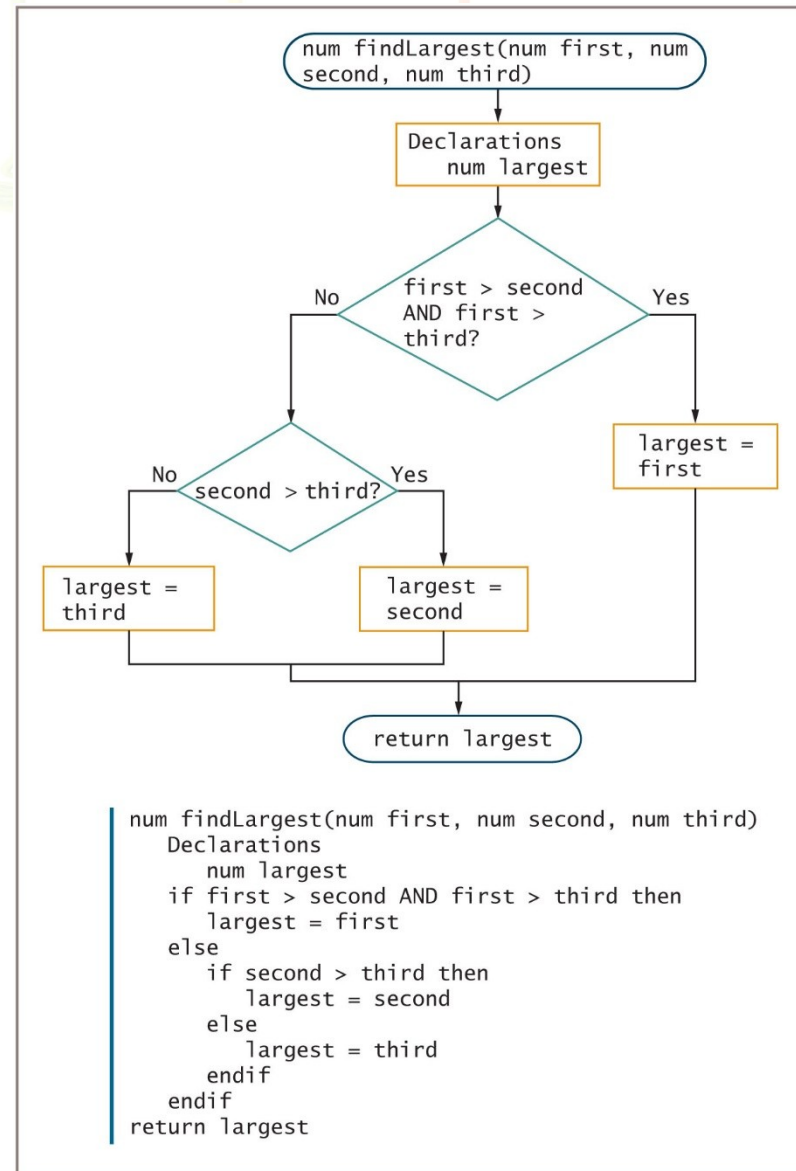


Figure 9-11 Recommended, structured approach to returning one of several values

Creating Methods that Return a Value

(continued -7)

- To use a method, you should know:
 - 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
- **Overhead** refers to the extra resources and time required by an operation, such as calling a method

Using an IPO Chart

- **IPO chart**
 - A tool that identifies and categorizes each item in a method by input, processing, and output
- A method that finds the smallest of three numeric values

Input	Processing	Output
First value second value Third value	If the firstvalue is smaller than each of the other two,save it as the smallest value; otherwise, if the second value is smaller than the third, save it as the smallest value; otherwise, save the third value as the smallest value	smallest value

Figure 9-12 IPO chart for the method that finds the smallest of three numeric values



Using an IPO Chart (continued -1)

- Many programmers create an IPO chart only for specific methods in their programs
- Provide an overview of:
 - Input to the method
 - Processing steps that must occur
 - Result



Passing an Array to a Method

- Pass a single array element to a method
 - Same manner you would pass a variable or constant
- Pass an entire array as an argument
- Indicate that a method parameter must be an array
 - Place square brackets after the data type in the method's parameter list
- Arrays are **passed by reference**
 - the method receives the actual memory address of the array and has access to the actual values in the array elements

Passing an Array to a Method

(continued - 1)

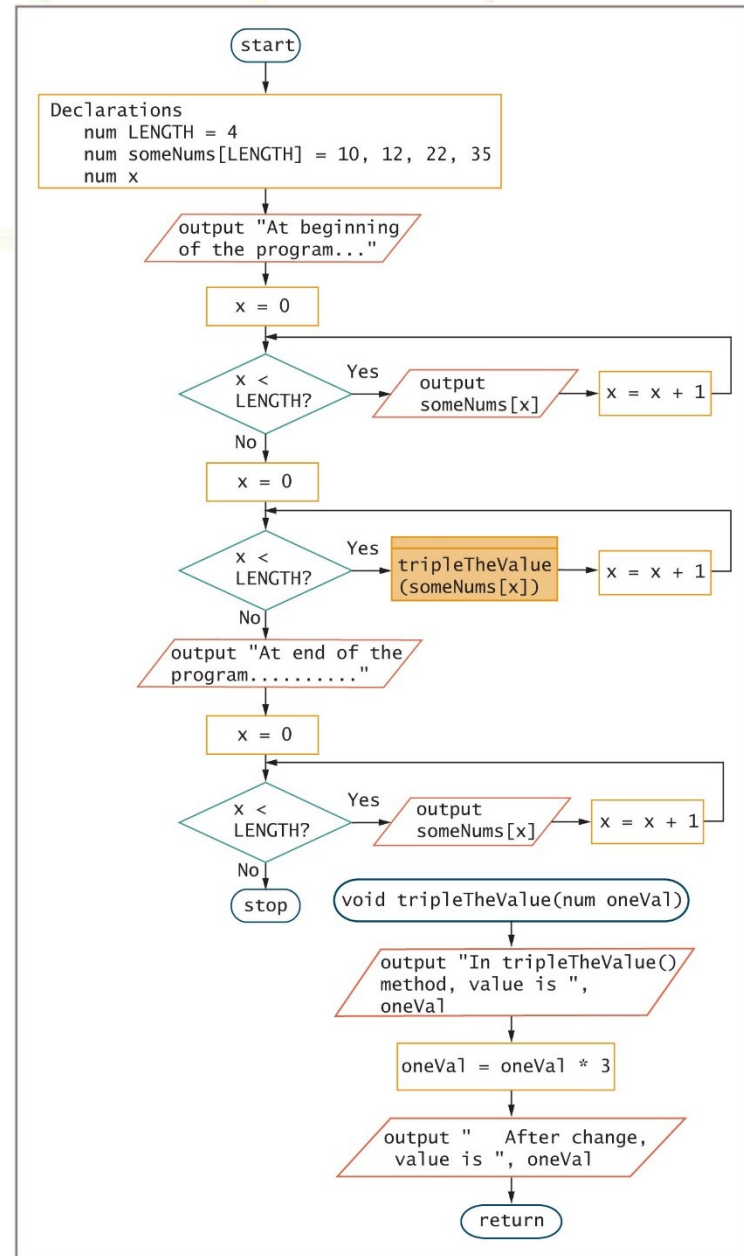


Figure 9-13 PassArrayElement program (continues)

Passing an Array to a Method

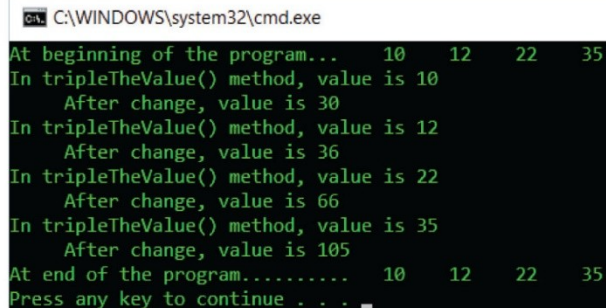
(continued -2)

(continued)

```
start
  Declarations
    num LENGTH = 4
    num someNums[LENGTH] = 10, 12, 22, 35
    num x
  output "At beginning of the program..."
  x = 0
  while x < LENGTH
    output someNums[x]
    x = x + 1
  endwhile
  x = 0
  while x < LENGTH
    tripleTheValue(someNums[x])
    x = x + 1
  endwhile
  output "At end of the program....."
  x = 0
  while x < LENGTH
    output someNums[x]
    x = x + 1
  endwhile
stop

void tripleTheValue(num oneVal)
  output "In tripleTheValue() method, value is ", oneVal
  oneVal = oneVal * 3
  output "    After change, value is ", oneVal
return
```

Figure 9-13 PassArrayElement program



```
C:\WINDOWS\system32\cmd.exe
At beginning of the program... 10 12 22 35
In tripleTheValue() method, value is 10
    After change, value is 30
In tripleTheValue() method, value is 12
    After change, value is 36
In tripleTheValue() method, value is 22
    After change, value is 66
In tripleTheValue() method, value is 35
    After change, value is 105
At end of the program..... 10 12 22 35
Press any key to continue . . .
```

Figure 9-14 Output of the PassArrayElement program

Passing an Array to a Method

(continued -3)

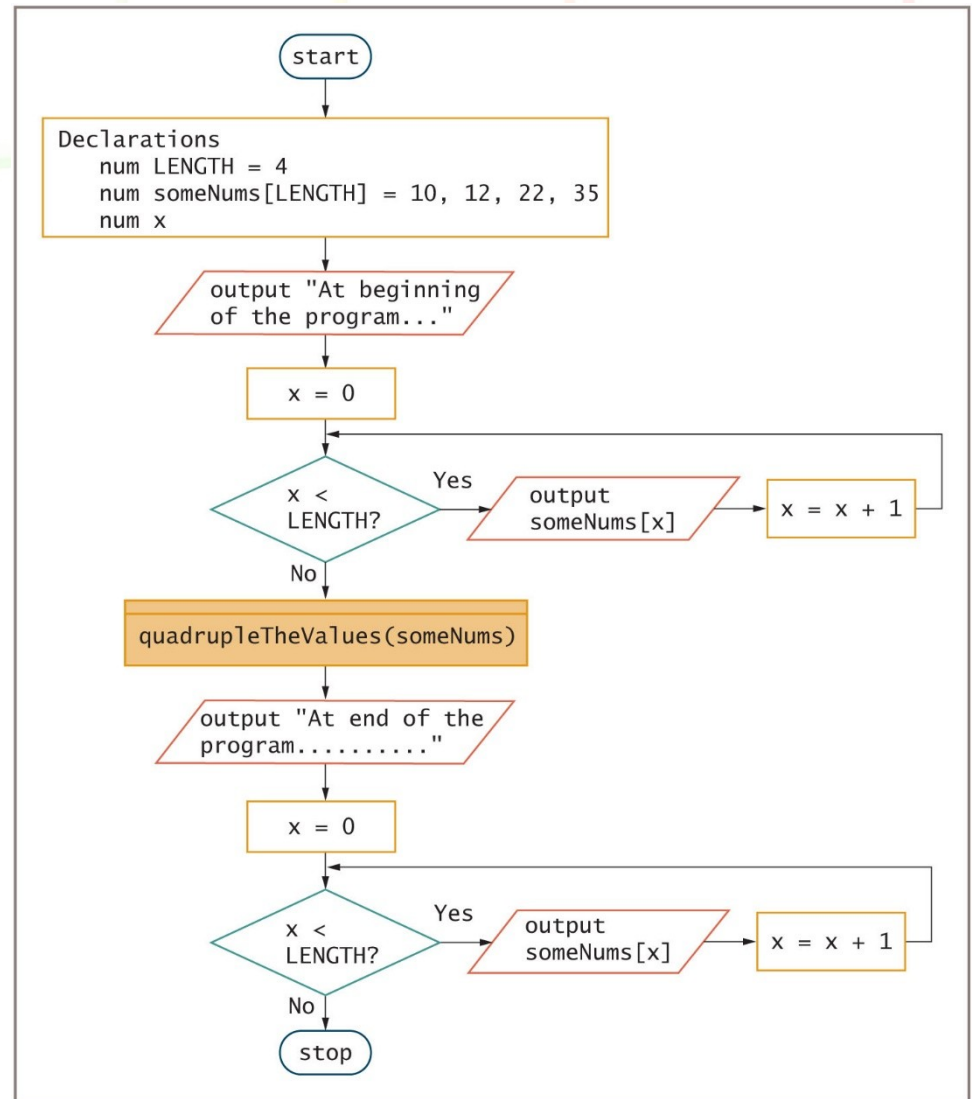


Figure 9-15 PassEntireArray program (continues)

Passing an Array to a Method

(continued -4)

(continued)

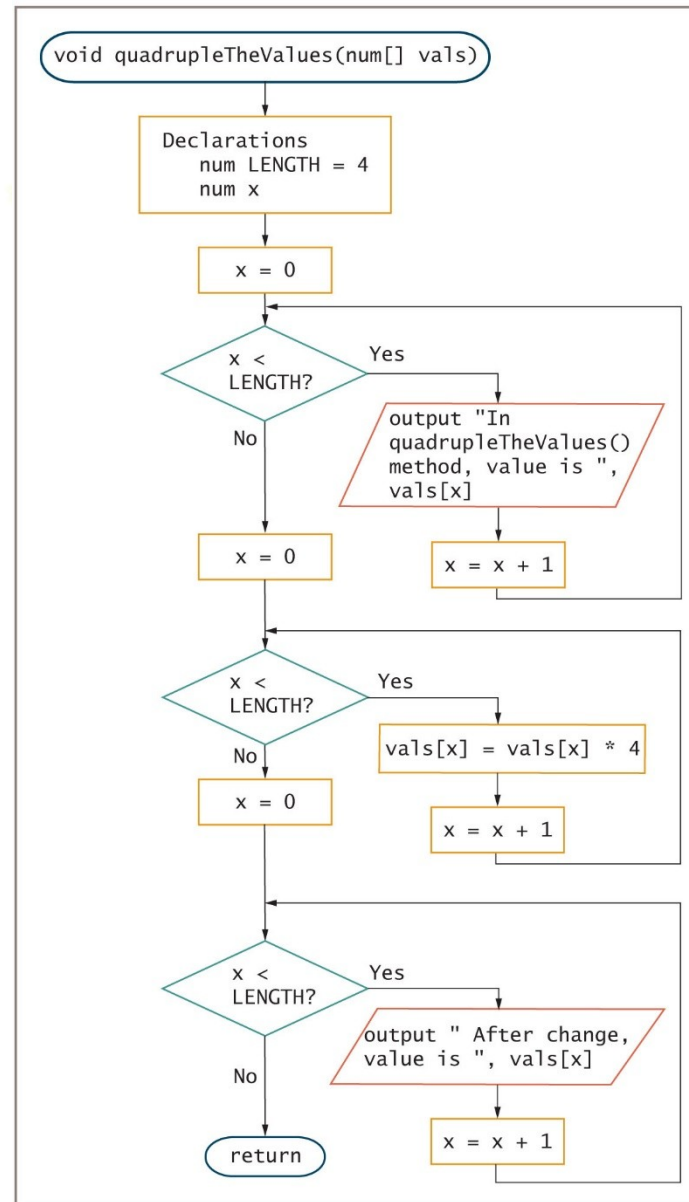


Figure 9-15 PassEntireArray program (continues)

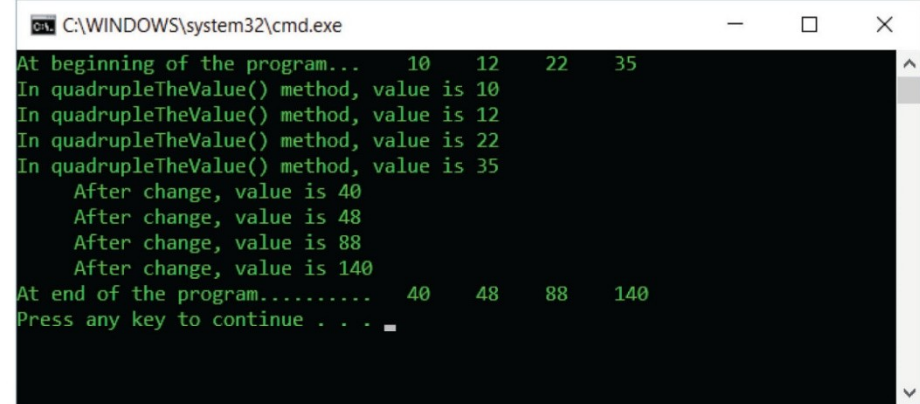
(continued)

```
start
  Declarations
    num LENGTH = 4
    num someNums[LENGTH] = 10, 12, 22, 35
    num x
  output "At beginning of the program..."
  x = 0
  while x < LENGTH
    output someNums[x]
    x = x + 1
  endwhile
  quadrupleTheValues(someNums)
  output "At end of the program....."
  x = 0
  while x < LENGTH
    output someNums[x]
    x = x + 1
  endwhile
stop

void quadrupleTheValues(num[] vals)
  Declarations
    num LENGTH = 4
    num x
  x = 0
  while x < LENGTH
    output "In quadrupleTheValues() method, value is ", vals[x]
    x = x + 1
  endwhile
  x = 0
  while x < LENGTH
    vals[x] = vals[x] * 4
    x = x + 1
  endwhile
  x = 0
  while x < LENGTH
    output "      After change, value is ", vals[x]
    x = x + 1
  endwhile
  return
```

Figure 9-15 PassEntireArray program

Passing an Array to a Method (continued - 5)



```
C:\WINDOWS\system32\cmd.exe

At beginning of the program... 10 12 22 35
In quadrupleTheValue() method, value is 10
In quadrupleTheValue() method, value is 12
In quadrupleTheValue() method, value is 22
In quadrupleTheValue() method, value is 35
      After change, value is 40
      After change, value is 48
      After change, value is 88
      After change, value is 140
At end of the program..... 40 48 88 140
Press any key to continue . . .
```

Figure 9-16 Output of the PassEntireArray program



Overloading Methods

- **Overloading**

- Involves supplying diverse meanings for a single identifier
- Similar to English language—the word *break* can be overloaded to mean:
 - Break a window
 - Break bread
 - Break the bank
 - Take a break

- **Overload a method**

- Write multiple methods with a shared name but different parameter lists

Overloading Methods (continued -1)

- Call an overloaded method
 - Language translator understands which version of the method to use based on the arguments used
- **Polymorphism**
 - Ability of a method to act appropriately according to the context
 - Literally, polymorphism means “many forms”

Overloading Methods (continued -2)

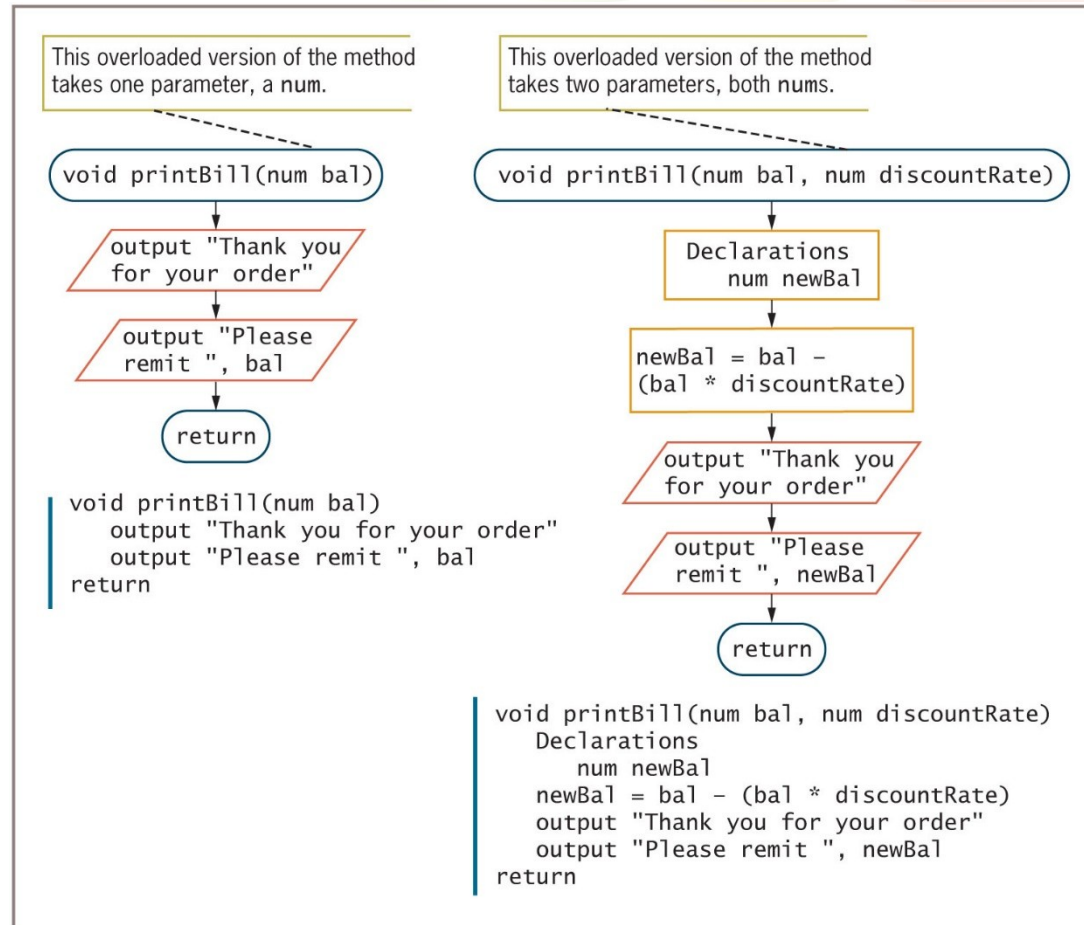


Figure 9-17 Two overloaded versions of the printBill() method

Overloading Methods (continued -3)

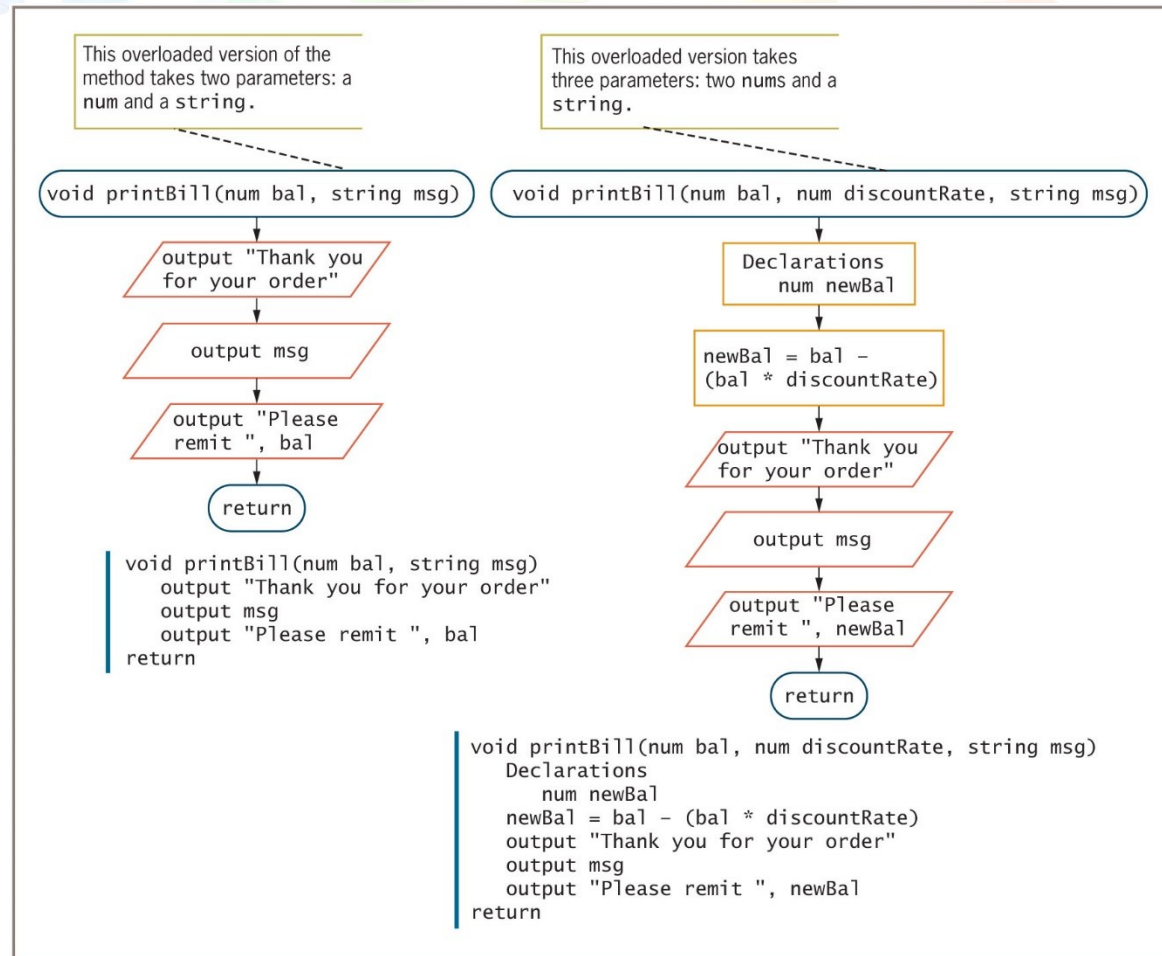


Figure 9-18 Two additional overloaded versions of the printBill() method



Overloading Methods

(continued -4)

- Overloading methods is never required in a program
 - Could create multiple methods with unique identifiers
- Advantage is provided to your method's clients
 - Those who use your methods need to remember just one appropriate name for all related tasks



Avoiding Ambiguous Methods

- **Ambiguous methods**
 - Situations in which the compiler cannot determine which method to use
- Every time you call a method
 - Compiler decides whether a suitable method exists
 - If so, the method executes
 - If not, you receive an error message

Avoiding Ambiguous Methods

1)

(continued -

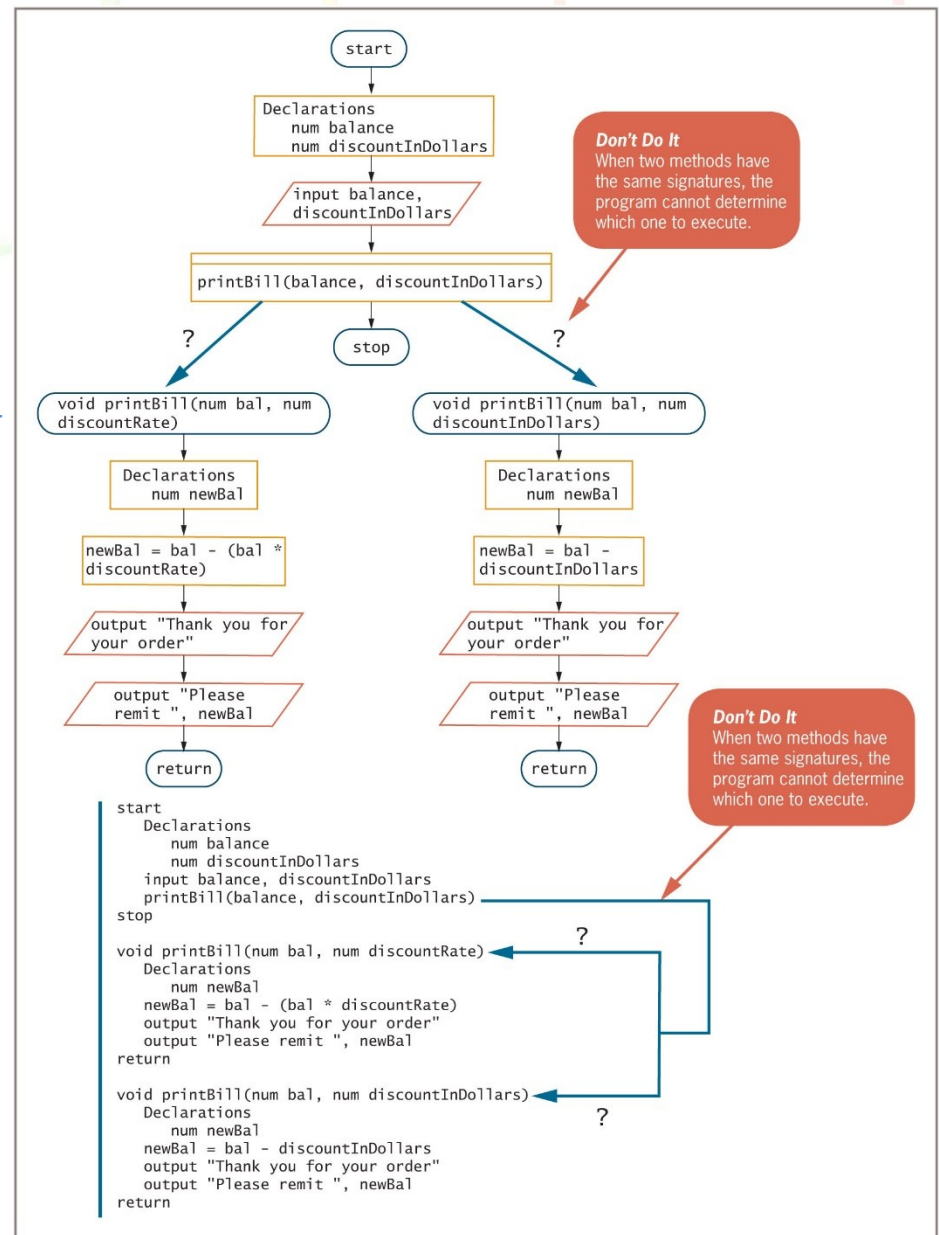


Figure 9-19 Program that contains ambiguous method call

Avoiding Ambiguous Methods

(continued -2)

- Overload method correctly
 - Different parameter lists for methods with the same name
- Ambiguous, not overloaded
 - Methods with identical names that have identical parameter lists but different return types
 - Example

```
string aMethod(num x)
num aMethod(num y)
```



Using Predefined Methods

- Modern programming languages contain many methods that have already been written
- Sources
 - Built-in methods
 - Program team members
 - Company-wide methods and standards
- Save time and effort
- Output methods
- Mathematical methods

Using Predefined Methods

(continued -1)

- To use predefined methods, you should know:
 - What the method does in general
 - Name
 - Required parameters
 - Return type
- You do not need to know:
 - How method is implemented



Method Design Issues: Implementation Hiding, Cohesion, and Coupling

- Consider several program qualities
 - Employ implementation hiding
 - Clients don't need to understand internal mechanisms
 - Strive to increase cohesion
 - Strive to reduce coupling

Understanding Implementation Hiding

- **Implementation hiding**
 - Encapsulation of method details
- When a program makes a request to a method, it does not know the details of how the method is executed
- A method that calls another must know:
 - The name of the called method
 - The type of information to send to the method
 - The type of return data to expect from the method

Understanding Implementation Hiding (continued -1)

- **Interface to the method**
 - The only part of a method with which the method's client interacts
- Substitute a new method implementation
 - As long as the interface does not change, you do not need to make changes in any methods that call the altered method
- A hidden implementation is often called a **black box**
 - You can examine what goes in and out but not how it works



Increasing Cohesion

- It is difficult to decide how much to put into a method
- **Cohesion**
 - How the internal statements of a method serve to accomplish the method's purpose
- Highly cohesive methods
 - All the operations are related
 - **Functionally cohesive**
 - Usually more reliable than those that have low cohesion



Reducing Coupling

- **Coupling**
 - A measure of the strength of the connection between two program methods
- **Tight coupling**
 - Occurs when methods excessively depend on each other
 - Makes programs more prone to errors
 - Methods have access to the same globally defined variables
- **Loose coupling**
 - Occurs when methods do not depend on others
 - Data is passed from one method to another



Understanding Recursion

- **Recursion**
 - Occurs when a method is defined in terms of itself
- **Recursive method**
 - Calls itself
- Every time you call a method
 - The address to which the program should return is stored in a memory location called the **stack**
 - When a method ends
 - Address is retrieved from the stack
 - Program returns to the location from which the method call was made

(continued -1)

```
start
    infinity()
stop

infinity()
    output "Help! "
    infinity()
return
```

Figure 9-20 A program that calls a recursive method

[illegible]

Figure 9-21 Output of the program in Figure 9-20

Understanding Recursion (continued -2)

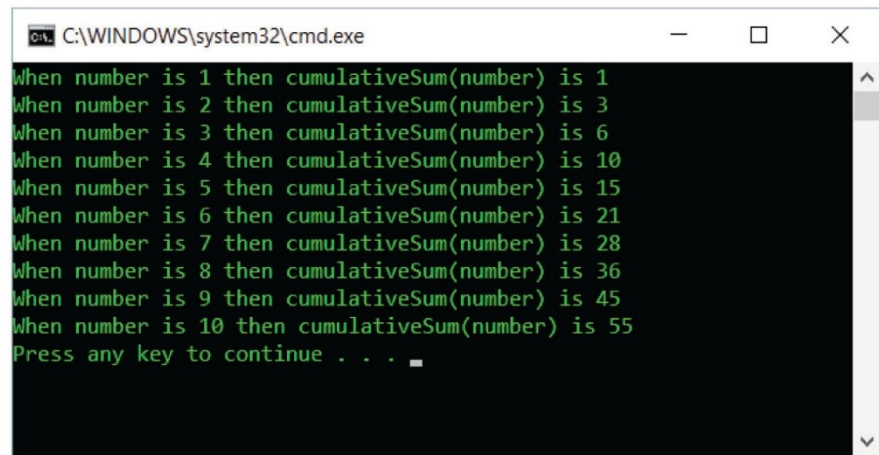
- The stack holds the address where the program should return at the completion of the method
 - Has a finite size
 - Will overflow if there are too many recursive calls
- **Recursive cases**
 - input values that cause a method to recur
- **Base case or terminating case**
 - input values that makes the recursion stop

Understanding Recursion (continued -3)

```
start
  Declarations
    num LIMIT = 10
    num number
  number = 1
  while number <= LIMIT
    output "When number is ", number,
      " then cumulativeSum(number) is ",
      cumulativeSum(number)
    number = number + 1
  endwhile
return

num cumulativeSum(num number)
  Declarations
    num returnVal
  if number = 1 then
    returnVal = number
  else
    returnVal = number + cumulativeSum(number - 1)
  endif
  return returnVal
```

Figure 9-22 Program that uses a recursive cumulativeSum() method



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed in green text on a black background. It shows the cumulative sum of numbers from 1 to 10. The output is as follows:

```
When number is 1 then cumulativeSum(number) is 1
When number is 2 then cumulativeSum(number) is 3
When number is 3 then cumulativeSum(number) is 6
When number is 4 then cumulativeSum(number) is 10
When number is 5 then cumulativeSum(number) is 15
When number is 6 then cumulativeSum(number) is 21
When number is 7 then cumulativeSum(number) is 28
When number is 8 then cumulativeSum(number) is 36
When number is 9 then cumulativeSum(number) is 45
When number is 10 then cumulativeSum(number) is 55
Press any key to continue . . .
```

Figure 9-23 Output of the program in Figure 9-22

Understanding Recursion (continued -4)

```
start
  Declarations
    num number
    num total
    num LIMIT = 10
  total = 0
  number = 1
  while number <= LIMIT
    total = total + number
    output "When number is ", number,
      " then the cumulative sum of 1 through",
      number, " is ", total
    number = number + 1
  endwhile
stop
```

Figure 9-24 Nonrecursive program that computes cumulative sums



Summary

- A method is a program module that contains a series of statements that carry out a task
 - Must include a header body and return statement
 - A program can contain an unlimited number of methods
 - Methods can be called an unlimited number of times
- Data passed to a method is called an argument
- Data received by a method is called a



Summary (continued)

- Single array elements or entire arrays can be passed
- Overloading allows you to create methods with different parameter lists under one name
- All modern languages contain prewritten methods
- Implementation is hidden in well-written methods
- Recursion occurs when methods call themselves; logic becomes difficult to follow