Visual Basic Language Companion for

Starting Out with Programming Logic and Design, 3rd Edition

By Tony Gaddis

Table of Contents

Introduction

Welcome to the Visual Basic Language Companion for *Starting Out with Programming Logic and Design, 3rd Edition*, by Tony Gaddis. You can use this guide as a reference for the Visual Basic Programming Language as you work through the textbook. Each chapter in this guide corresponds to the same numbered chapter in the textbook. As you work through a chapter in the textbook, you can refer to the corresponding chapter in this guide to see how the chapter's topics are implemented in the Visual Basic programming language. In this book you will also find Visual Basic versions of many of the pseudocode programs that are presented in the textbook.

Chapter 1 Introduction to Visual Basic

To program in Visual Basic, you will need to have one of the following products installed on your computer:

- Microsoft Visual Studio
- Microsoft Visual Basic Express Edition

Starting Visual Studio

There's a good chance that Microsoft Visual Studio has been installed in your school's computer lab. This is a powerful programming environment that allows you to create applications, not only in Visual Basic, but other languages such C# and C++. If Visual Studio is installed on the computer that you are using, you can find it by clicking the Windows *Start* button, then selecting *All Programs*. You will see a program group for *Microsoft Visual Studio 2010*. Open the program group and execute *Microsoft Visual Studio 2010*.

Downloading and Installing Visual Basic Express Edition

If you don't have Visual Studio installed on your personal computer, you can download and install Visual Basic Express Edition for free from this Web site:

```
www.microsoft.com/express/vb
```

This Web page will allow you to download the current version of Visual Basic Express Edition. The installation is very simple. Just click the link to download and install, and follow the instructions. The time required for the installation to complete will depend on the speed of your Internet connection.

Starting Visual Basic Express Edition

Once Visual Basic Express Edition has been installed, you can start it by clicking the Windows *Start* button, then selecting *All Programs*. You will see *Microsoft Visual Basic 2010 Express Edition* on the program menu.

Starting a New Project

Each time you want to begin writing a new program, you have to start a new project. In the following tutorial you will start a new Visual Basic project, write a line of code, save the project, and then run it.

Tutorial: Starting, Saving, and Running a Visual Basic Project

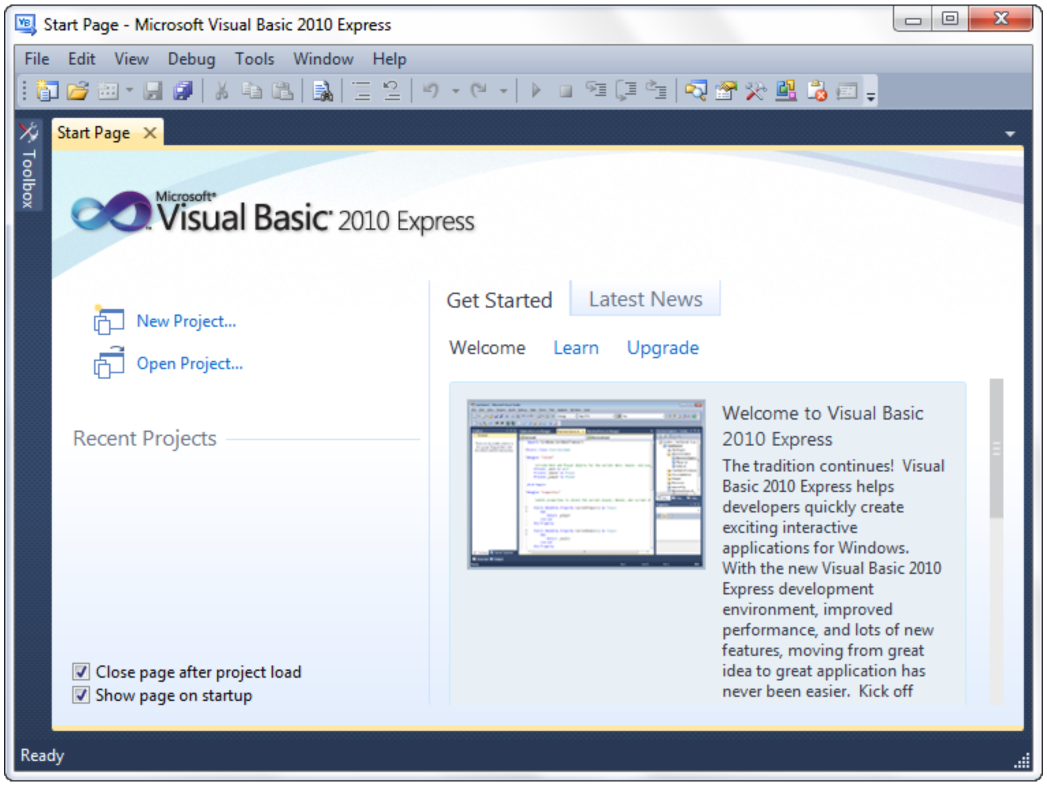
Step 1: Start either Visual Studio or Visual Basic Express Edition. Figure 1-3 shows Visual Basic 2010 Express Edition running. (Visual Studio will look similar.) The screen shown in the figure is known as the *Start Page*. To start a new project, click *File* on the menu, and then click *New Project*… This will display the *New Project* window, shown in Figure 1-4.

Note: If you are using Visual Studio, you will see an area titled *Project Types* at the left side of the New Project window. Make sure *Visual Basic* is selected.

Step 2: In this book, most of the programs that we demonstrate are Visual Basic console applications. (The only exception is in Chapter 15, where we use Windows Forms applications.) As shown in Figure 1-4 make sure *Console Application* is selected in the section titled Visual Studio installed templates.

At the bottom of the *New Project* window you see a *Name* text box. This is where you enter the name of your project. This will be automatically filled in with a default name. In Figure 1-18 the default name is *ConsoleApplication1*. It is recommended that you change to project name to something more meaningful. Once you have changed the project name, click the *OK* button.

Figure 1-3 Visual Basic 2010 Express Edition

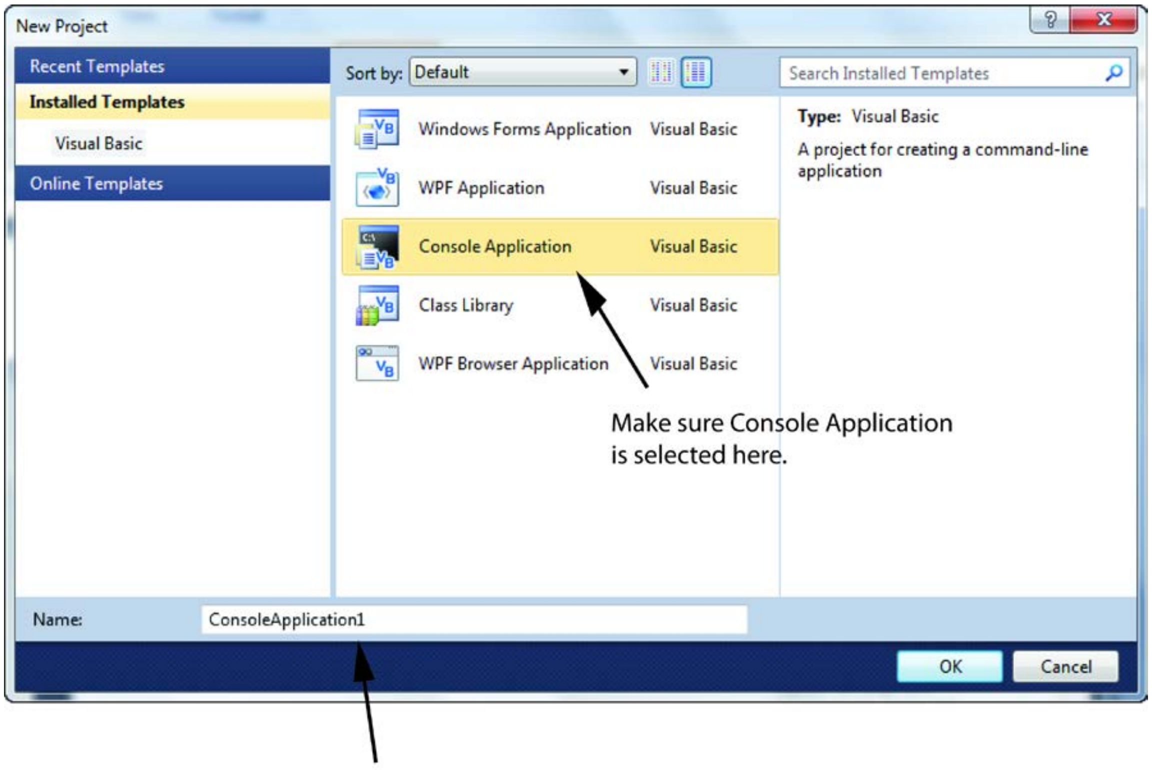


Welcome to Visual Basic 2010 Express

The tradition continues! Visual Basic 2010 Express helps developers quickly create exciting interactive applications for Windows. With the new Visual Basic 2010 Express development environment, improved performance, and lots of new features, moving from great idea to great application has never been easier. Kick off

[✓] Close page after project load
[✓] Show page on startup

Figure 1-4 The *New Project* window



Sort by: Default

Windows Forms Application	Visual Basic
WPF Application	Visual Basic
Console Application	Visual Basic
Class Library	Visual Basic
WPF Browser Application	Visual Basic

Type: Visual Basic

A project for creating a command-line application

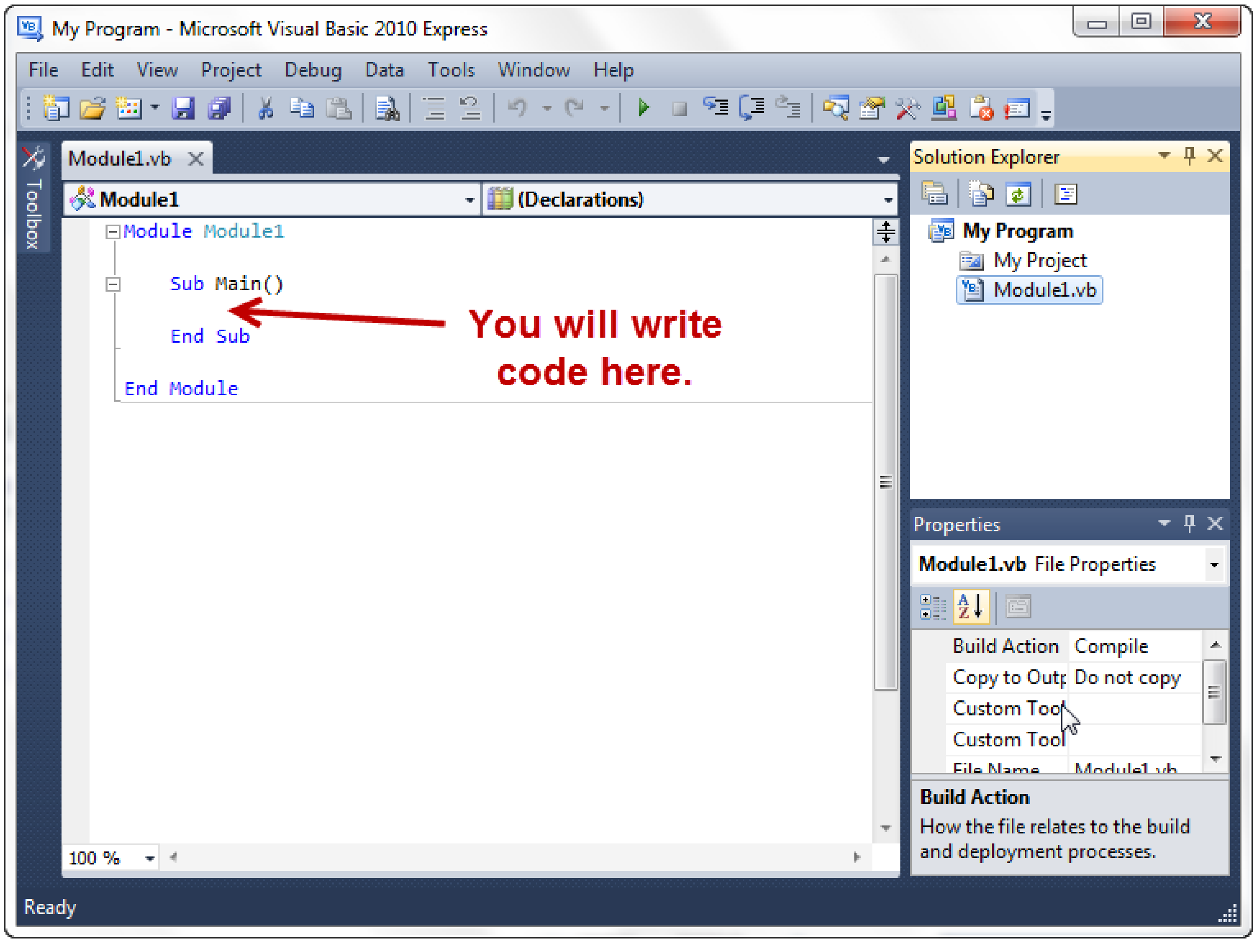
Name: ConsoleApplication1

Change the application's name.

Note: If you create a lot of Visual Basic projects, you will find that default names such as *ConsoleApplication1* do not help you remember what each project does. Therefore, you should always change the name of a new project to something that describes the project's purpose.

Step 3: It might take a moment for the project to be created. Once it is, the Visual Basic environment should appear similar to Figure 1-5. The large window that occupies most of the screen is the code editor. This is where you write Visual Basic programming statements.

Figure 1-5 The Visual Basic Environment



```
Module Module1

    Sub Main()

    End Sub

End Module
```

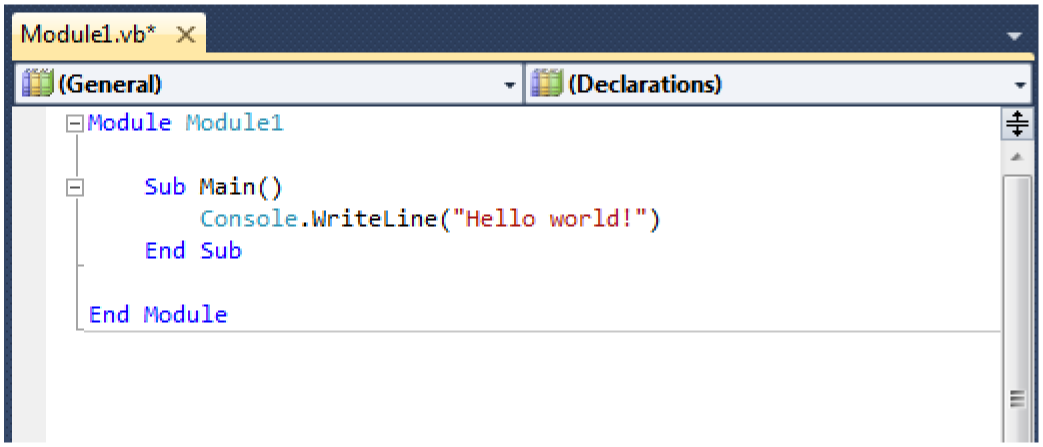
My Program - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Toolbox
Module1.vb
Module1
(Declarations)
You will write code here.
100 %
Solution Explorer
My Program
My Project
Module1.vb
Properties
Module1.vb File Properties
Build Action Compile
Copy to Outp Do not copy
Custom Tool
Custom Tool
File Name Module1.vb
Build Action
How the file relates to the build and deployment processes.
Ready

Let's try the code editor. Notice that some code has already been provided in the window. Between the lines that reads `Sub Main()`and `End Sub`, type the following statement:

```
Console.WriteLine("Hello world!")
```

After typing this statement, the code editor should appear as shown in Figure 1-6. Congratulations! You've written your first Visual Basic program!

Figure 1-6 The code window

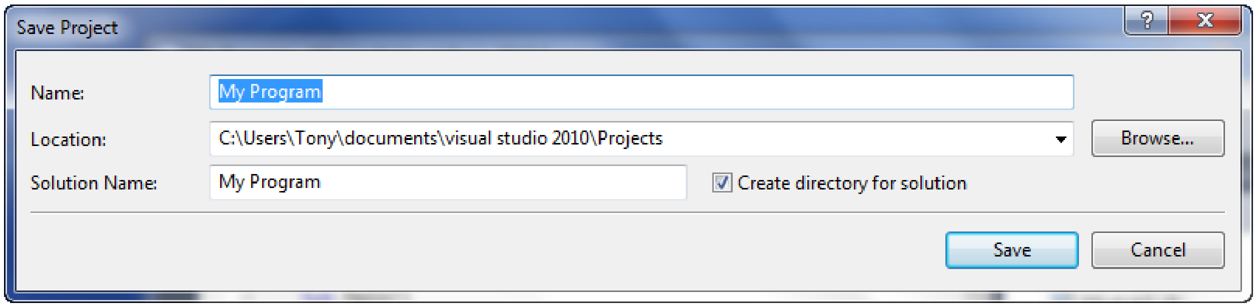


```
Module Module1

    Sub Main()
        Console.WriteLine("Hello world!")
    End Sub

End Module
```

Step 4: Before going any further, you should save the project. Click *File* on the menu bar, then click *Save All*. The *Save Project* window will appear, as shown in Figure 1-7. The *Name* text box shows the project name that you entered when you created the project. The *Location* text box shows where a folder will be created on your system to hold the project. If you wish to change the location, click the *Browse* button and select the desired drive and folder.

Click the *Save* button to save the project.

Figure 1-7 The *Save Project* dialog box



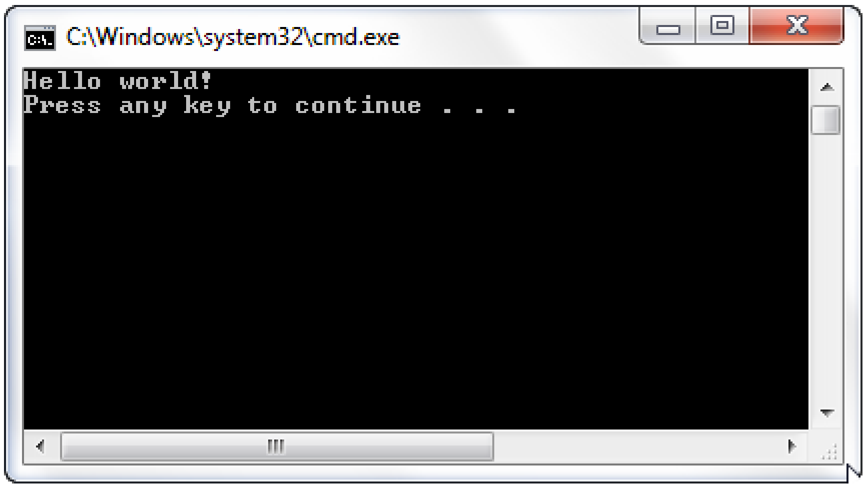
Name: My Program
Location: C:\Users\Tony\documents\visual studio 2010\Projects
Solution Name: My Program
[✓] Create directory for solution

Step 5: Now let's run the program. Press *Ctrl+F5* on the keyboard. If you typed the statement correctly back in Step 3, the window shown in Figure 1-8 should appear. This is the output of the program. Press any key on the keyboard to close the window.

If the window shown in Figure 1-8 does not appear, make sure the contents of the code editor appears exactly as shown in Figure 1-6. Correct any mistakes, and press *Ctrl+F5* on the keyboard to run the program.

Step 6: When you are finished, click *File* on the menu bar, and then click *Exit* to exit Visual Basic.

Figure 1-8 Program output



```
Hello world!
Press any key to continue . . .
```

Chapter 2 Input, Processing, and Output

Before we look at any Visual Basic code, there are two simple concepts that you must learn:

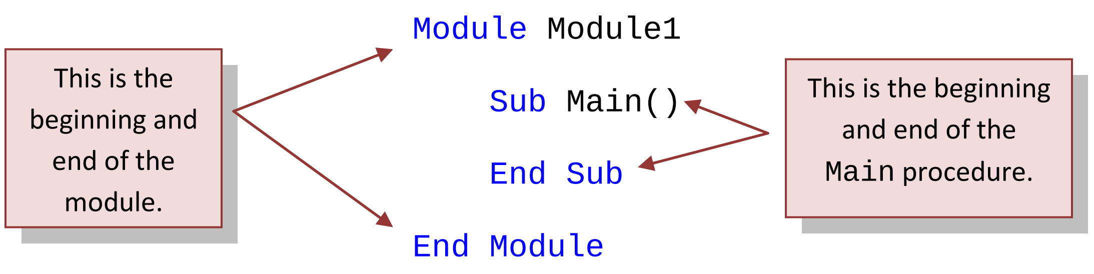
- In Visual Basic, a module is a container that holds code. In almost all of the examples in this book, a module holds all of a program's code.
- A sub procedure (which is also known simply as a procedure) is another type of container that holds code. Almost all of the programs that you will see in this book have a sub procedure named `Main`. When a Visual Basic console program executes, it automatically begins running the code that is inside the `Main` sub procedure.

When you start a new Visual Basic Console project, the following code automatically appears in the code editor:

```
Module Module1

    Sub Main()

    End Sub

End Module
```

This code is required for your program, so don't erase it or modify it. In a nutshell, this code sets up a module for the program, and inside the module it creates a `Main` sub procedure. Let's look at the details.

The first line, which reads `Module Module1`, is the beginning of a module declaration. The last line, which reads `End Module`, is the end of the module. Between these two lines the Main sub procedure appears. The line that reads `Sub Main()`is the beginning of the `Main` sub procedure, and the line that reads `End Sub` is the end of the `Main` sub procedure. This is illustrated here:



```
Module Module1

    Sub Main()

    End Sub

End Module
```

This is the beginning and end of the module.
This is the beginning and end of the Main procedure.

In this chapter, when you write a program, all of the code that you will write will appear inside the `Main` sub procedure, as illustrated here:

```
Module Module1

    Sub Main()

    End Sub

End Module
```

The code that you write will go here.

Displaying Screen Output

To display text on the screen in a Visual Basic console program you use the following statements:

- `Console.WriteLine()`
- `Console.Write()`

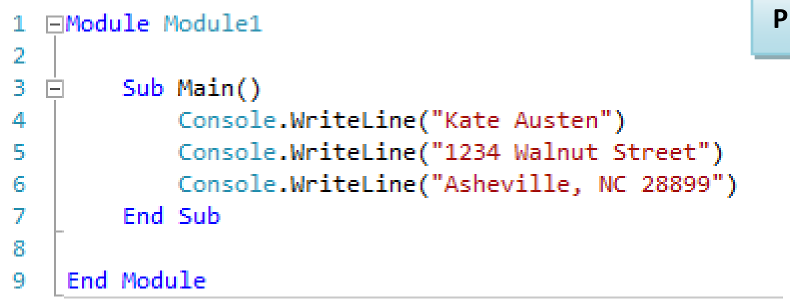
First, let's look at the `Console.WriteLine()`statement. The purpose of this statement is to display a line of output. Notice that the statement ends with a set of parentheses. The text that you want to display is written as a string inside the parentheses. Program 2-1 shows an example. (This is the Visual Basic version of pseudocode Program 2-1 in your textbook.)

First, a note about the line numbers that you see in the program. These are *NOT* part of the program! They are helpful, though, when we need to discuss parts of the code. We can simply refer to specific line numbers and explain what's happening in those lines. For that reason we will show line numbers in all of our program listings. When you are writing a program, however, do not type line numbers in your code. Doing so will cause a mountain of errors when you compile the program!

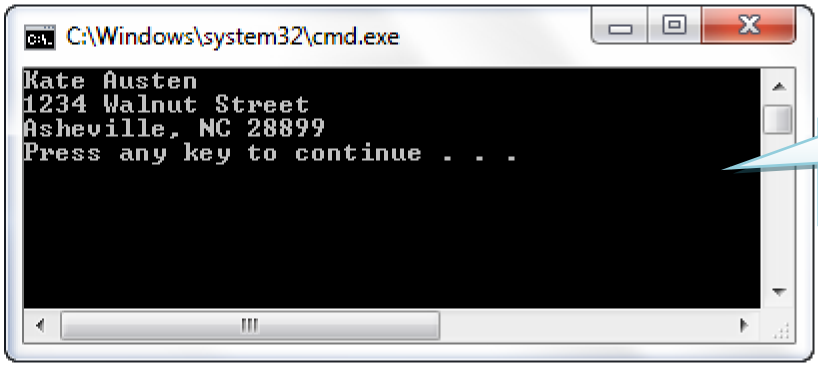
Program 2-2 has three `Console.WriteLine()`statements, appearing in lines 4, 5, and 6. (I told you those line numbers would be useful!) Line 4 displays the text `Kate Austen`, line 5 displays the text `123 Dharma Lane`, and line 6 displays the text `Asheville, NC 28899`. The program's output is also shown.

Program 2-1

This program is the VB version of **Program 2-1** in your textbook!



```
Module Module1

    Sub Main()
        Console.WriteLine("Kate Austen")
        Console.WriteLine("1234 Walnut Street")
        Console.WriteLine("Asheville, NC 28899")
    End Sub

End Module
```



```
Kate Austen
1234 Walnut Street
Asheville, NC 28899
Press any key to continue . . .
```

This is the program's output.

Notice that the output of the `Console.WriteLine()`statements appear on separate lines. When the `Console.WriteLine()`statement displays output, it advances the output cursor (the location where the next item of output will appear) to the next line. That means the `Console.WriteLine()`statement displays its output, and then the next thing that is displayed will appear on the following line.

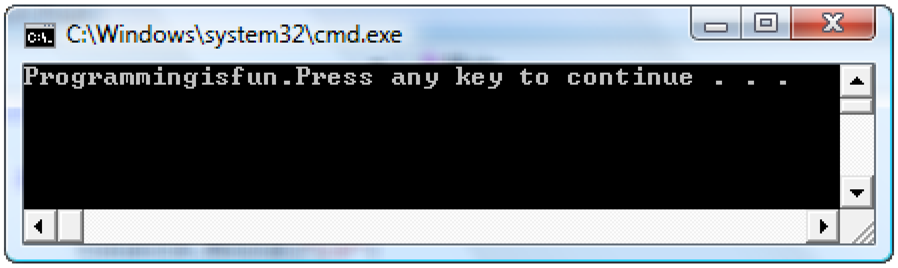
The `Console.Write()`statement displays output, but it does not advance the output cursor to the next line. Program 2-2 shows an example. The program's output is also shown.

Oops! It appears from the program output that something went wrong. All of the words are jammed together into one long series of characters. If we want spaces to appear between the words, we have to explicitly display them. Program 2-3 shows how we have to insert spaces into the strings that we are displaying, if we want the words to be separated on the screen. Notice that in line 4 we have inserted a space in the string, after the letter `g`, in line 5 we have inserted a space in the string after the letter `s`, and in line 6 we have inserted a space in the string after the period.

Program 2-2

```
Module Module1

    Sub Main()
        Console.Write("Programming")
        Console.Write("is")
        Console.Write("fun.")
    End Sub

End Module
```

This is the program's output.



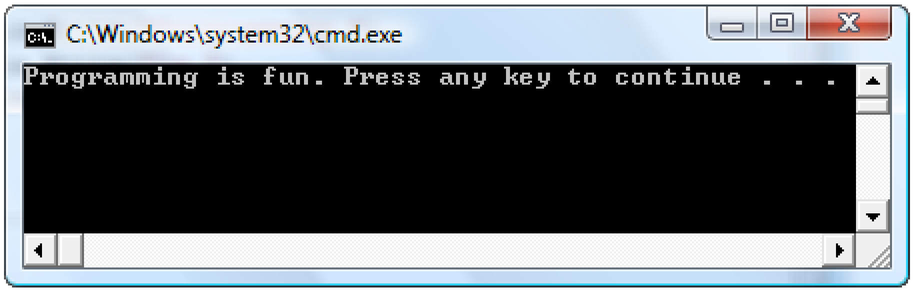
```
Programmingisfun.Press any key to continue . . .
```

Program 2-3

Notice these spaces, inside the strings.

```
Module Module1

    Sub Main()
        Console.Write("Programming ")
        Console.Write("is ")
        Console.Write("fun. ")
    End Sub

End Module
```

This is the program's output.



```
Programming is fun. Press any key to continue . . .
```

Variables

In Visual Basic, variables must be declared before they can be used in a program. A variable declaration statement is written in the following general format:

```
Dim VariableName As DataType
```

Variable declarations begin with the word `Dim`. (The word `Dim` is short for *dimension*, and has historically been used to start variable declarations in many different versions of the BASIC programming language.) In the general format, *`VariableName`* is the name of the variable that you are declaring and *`DataType`* is the name of a Visual Basic data type. For example, the key word `Integer` is the name of the integer data type in Visual Basic, so the following statement declares an `Integer` variable named `number`.

```
Dim number As Integer
```

Table 2-1 lists the Visual Basic data types that we will use in this book, gives their memory size in bytes, and describes the type of data that each can hold. Note that there are many more data types available in Visual Basic. These, however, are the ones we will be using.

Table 2-1 A Few of the Visual Basic Data Types

Data Type	Size	What It Can Hold
`Integer`	4 bytes	Integers in the range of –2,147,483,648 to +2,147,483,647
`Double`	8 bytes	Floating-point numbers in the following ranges: $-1.79769313486231570\times10^{308}$ through $-4.94065645841246544\times10^{-324}$ for negative values; $4.94065645841246544\text{E}\times10^{-324}$ through $1.79769313486231570\text{E}\times10^{308}$ for positive values
`String`	Varies	Strings of text.
`Boolean`	Varies	True or False

Here are some other examples of variable declarations:

```
Dim speed As Integer
Dim distance As Double
Dim name As String
```

Several variables of the same data type can be declared with the same declaration statement. For example, the following statement declares three `Integer` variables named `width`, `height`, and `length`.

```
Dim width, height, length As Integer
```

You can also initialize variables with starting values when you declare them. The following statement declares an `Integer` variable named `hours`, initialized with the starting value 40:

```
Dim hours As Integer = 40
```

Variable Names

You may choose your own variable names in Visual Basic, as long as you follow these rules:

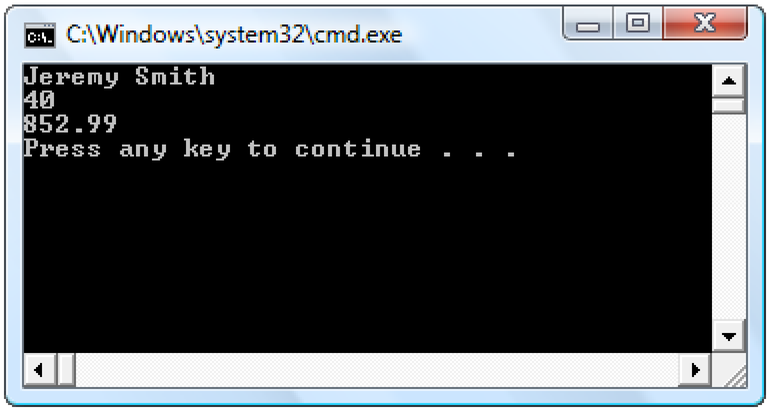
- The first character must be a letter or an underscore character. (We do not recommend that you start a variable name with an underscore, but if you do, the name must also contain at least one letter or numeric digit.)
- After the first character, you may use letters, numeric digits, and underscore characters. (You cannot use spaces, periods, or other punctuation characters in a variable name.)
- Variable names cannot be longer than 1023 characters.
- Variable names cannot be Visual Basic key words. Key words have reserved meanings in Visual Basic, and their use as variable names would confuse the compiler.

Program 2-4 shows an example with three variable declarations. Line 4 declares a `String` variable named `name`, initialized with the string "Jeremy Smith". Line 5 declares an `Integer` variable named `hours` initialized with the value 40. Line 6 declares a `Double` variable named `pay`, initialized with the value 852.99. Notice that in lines 8 through 10 we use `Console.WriteLine` to display the contents of each variable.

Program 2-4

```
Module Module1

    Sub Main()
        Dim name As String = "Jeremy Smith"
        Dim hours As Integer = 40
        Dim pay As Double = 852.99

        Console.WriteLine(name)
        Console.WriteLine(hours)
        Console.WriteLine(pay)
    End Sub

End Module
```

This is the program's output.



```
Jeremy Smith
40
852.99
Press any key to continue . . .
```

Assignment Statements

You assign a value to an existing variable with an assignment statement, which is written in the following general format:

`variable = value`

In the general format, variable is the name of the variable that is receiving the assignment, and value is the value that is being assigned. (This is like assignment statements in your textbook, but without the word `Set`.)

When writing an assignment statement, make sure the variable that is receiving the value appears on the left side of the = operator. Also, make sure that the value appearing on the right side of the = operator is of a data type that is compatible with the variable on the left side. For example, you shouldn't try to assign a floating-point value to an `Integer` variable, or a string to a `Double`.

Reading Keyboard Input

First, let's talk about reading string input. In a Visual Basic console program, you use `Console.ReadLine()` to read a string that has been typed on the keyboard, and you usually assign that input to a `String` variable. Here is the general format:

```
variable = Console.ReadLine()
```

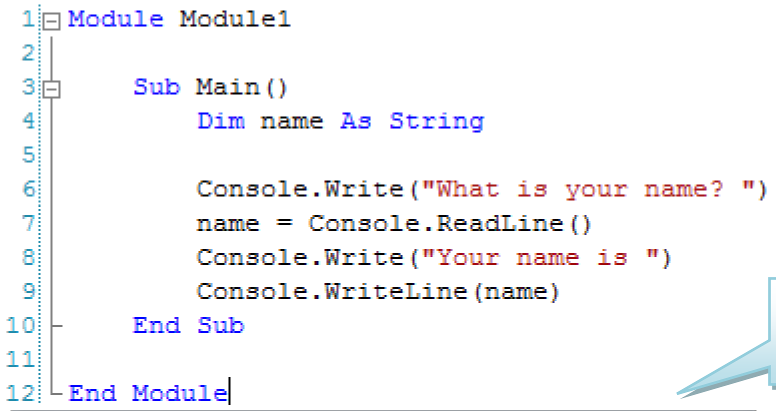
In the general format, variable is a `String` variable. When the statement executes, the program will pause for the user to type input on the keyboard. After the user presses the Enter key, the input that was typed on the keyboard will be assigned to the *`variable`*. Here is an example of actual code:

```
Dim name As String
Console.Write("What is your name? ")
name = Console.ReadLine()
```

The first line declares a `String` variable named `name`. The second statement displays the message "What is your name? ". The third statement waits for the user to enter something on the keyboard. When the user presses Enter, the input that was typed on the keyboard is assigned to the `name` variable. For example, if the user types *Jesse* and then presses Enter, the string "Jesse" will be assigned to the `name` variable.

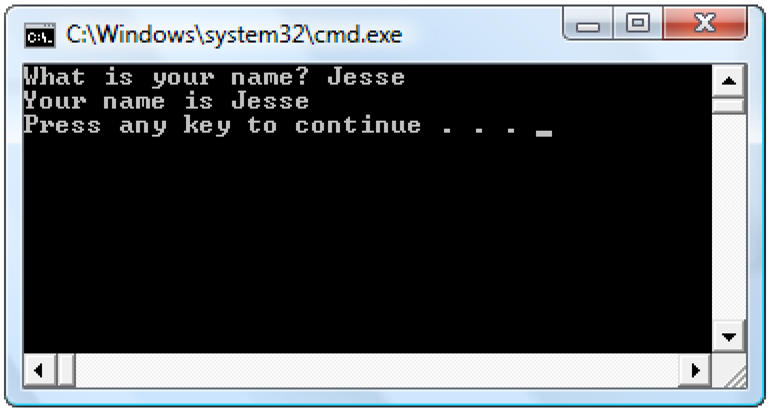
Program 2-5 shows a complete example.

Program 2-5



```
Module Module1

    Sub Main()
        Dim name As String

        Console.Write("What is your name? ")
        name = Console.ReadLine()
        Console.Write("Your name is ")
        Console.WriteLine(name)
    End Sub

End Module
```

This is the program's output.



```
What is your name? Jesse
Your name is Jesse
Press any key to continue . . . _
```

Reading Numeric Input

`Console.ReadLine()` always reads keyboard input as a string, even if the user enters a number. For example, suppose we are using `Console.ReadLine()` to read input, and the user enters the number 72. The input will be returned from `Console.ReadLine()` as the string "72". This can be a problem if you want to use the user's input in a math operation because you cannot perform math on strings. In such a case, you must use a *data type conversion function* to convert the input to a numeric value. To convert a string to an `Integer`, we will use the `CInt` function, and to convert a string to a `Double` we will use the `CDbl` function. Here is an example that demonstrates the `CInt` function:

```
Dim number As Integer
Console.Write("Enter an integer number: ")
number = CInt(Console.ReadLine())
```

The first line declares an `Integer` variable named `number`. The second statement displays the message "Enter an integer number: ". The third statement does the following:

- `Console.ReadLine()` waits for the user to enter something on the keyboard. When the user presses Enter, the input that was typed on the keyboard is returned as a string.
- The `CInt` function converts the string that was returned from `Console.ReadLine()` to an `Integer`.
- The `Integer` that is returned from the `CInt` function is assigned to the `number` variable.

For example, if the user enters 72 and then presses Enter, the `Integer` value 72 will be assigned to the `number` variable.

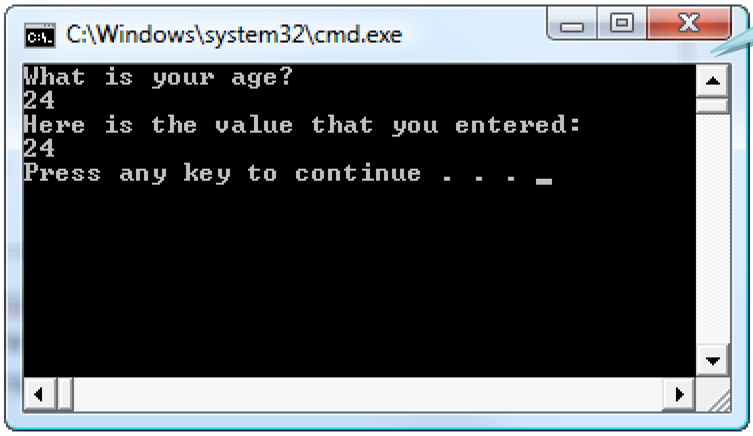
Let's look at a complete program that uses the `CInt` function. Program 2-6 is the Visual Basic version of pseudocode Program 2-2 in your textbook.

Program 2-6

This program is the VB version of **Program 2-2** in your textbook!

```
Module Module1

    Sub Main()
        Dim age As Integer

        Console.WriteLine("What is your age?")
        age = CInt(Console.ReadLine())
        Console.WriteLine("Here is the value that you entered:")
        Console.WriteLine(age)
    End Sub

End Module
```

This is the program's output.



```
What is your age?
24
Here is the value that you entered:
24
Press any key to continue . . .
```

Let's take a closer look at the code:

- Line 4 declares an `Integer` variable named `age`.
- Line 6 displays the string "`What is your age?`"
- Line 7 uses `Console.ReadLine()` to read a string from the keyboard. The `CInt` function converts that input to an `Integer`, and the resulting `Integer` is assigned to the `age` variable.
- Line 8 displays the string "`Here is the value that you entered:`"
- Line 9 displays the value of the `age` variable.

Here's an example that demonstrates the `CDbl` function, which converts a string to a `Double`:

```
Dim number As Double
Console.Write("Enter a floating-point number: ")
number = CDbl(Console.ReadLine())
```

The first line declares a `Double` variable named `number`. The second statement displays the message "Enter a floating-point number: ". The third statement does the following:

- `Console.ReadLine()` waits for the user to enter something on the keyboard. When the user presses Enter, the input that was typed on the keyboard is returned as a string.
- The `CDbl` function converts the string that was returned from `Console.ReadLine()` to a `Double`.
- The `Double` that is returned from the `CDbl` function is assigned to the `number` variable.

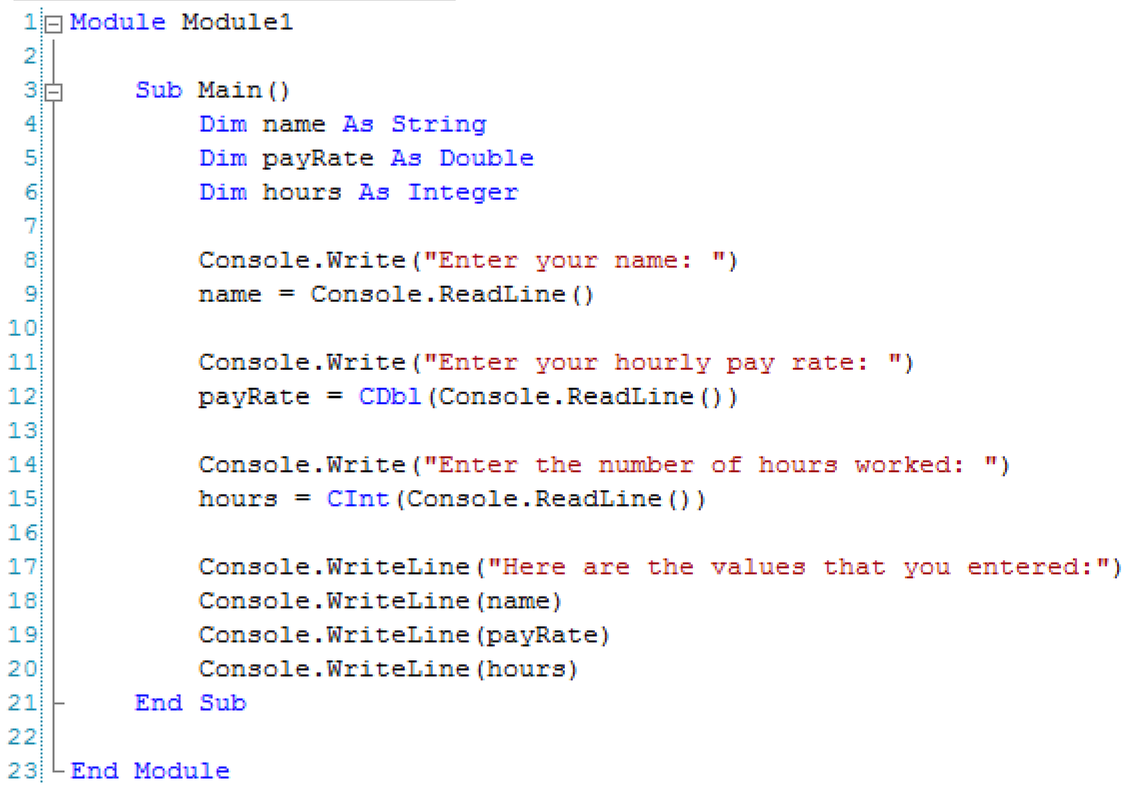
For example, if the user enters 3.5 and then presses Enter, the `Double` value 3.5 will be assigned to the `number` variable.

Program 2-7 demonstrates how to read `Strings`, `Doubles`, and `Integers` from the keyboard:

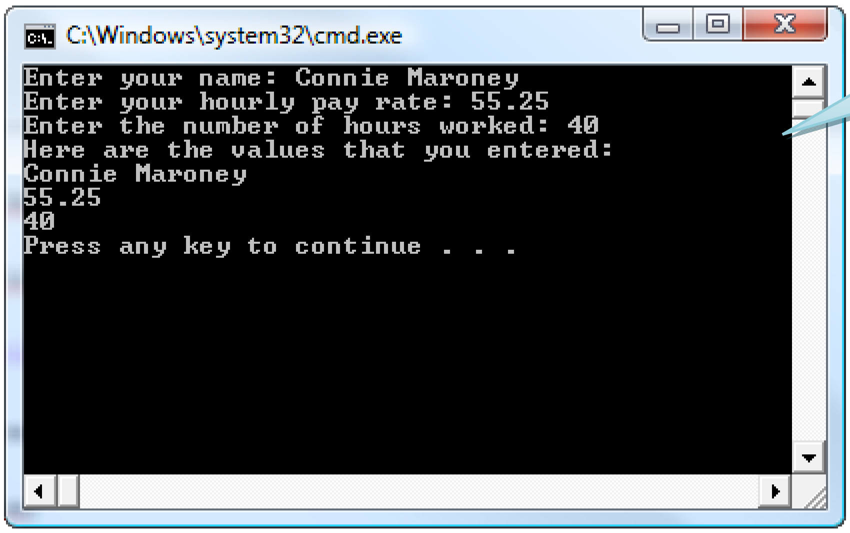
- Line 4 declares a `String` variable named `name`, line 5 declares a `Double` variable named `payRate`, and line 6 declares an `Integer` variable named `hours`.
- Line 9 uses `Console.ReadLine()` to read a string from the keyboard, and assigns the string to the `name` variable.

- Line 12 uses `Console.ReadLine()` to read a string from the keyboard, then uses `CDbl` to convert that input to a `Double`, and assigns the resulting `Double` to the `payRate` variable.
- Line 15 uses `Console.ReadLine()` to read a string from the keyboard, then uses `CInt` to convert that input to an `Integer`, and assigns the resulting `Integer` to the `hours` variable.

Program 2-7



```
 1 Module Module1
 2
 3     Sub Main()
 4         Dim name As String
 5         Dim payRate As Double
 6         Dim hours As Integer
 7
 8         Console.Write("Enter your name: ")
 9         name = Console.ReadLine()
10
11         Console.Write("Enter your hourly pay rate: ")
12         payRate = CDbl(Console.ReadLine())
13
14         Console.Write("Enter the number of hours worked: ")
15         hours = CInt(Console.ReadLine())
16
17         Console.WriteLine("Here are the values that you entered:")
18         Console.WriteLine(name)
19         Console.WriteLine(payRate)
20         Console.WriteLine(hours)
21     End Sub
22
23 End Module
```



```
Enter your name: Connie Maroney
Enter your hourly pay rate: 55.25
Enter the number of hours worked: 40
Here are the values that you entered:
Connie Maroney
55.25
40
Press any key to continue . . .
```

This is the program's output.

Displaying Multiple Items with the & Operator

The & operator is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
Console.WriteLine("This is " & "one string.")
```

This statement will display:

```
This is one string.
```

The & operator produces a string that is the combination of the two strings used as its operands. You can also use the & operator to concatenate the contents of a variable to a string. The following code shows an example:

```
Dim number As Integer = 5
Console.WriteLine("The value is " & number)
```

The second line uses the & operator to concatenate the contents of the `number` variable with the string `"The value is "`. Although `number` is not a string, the & operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

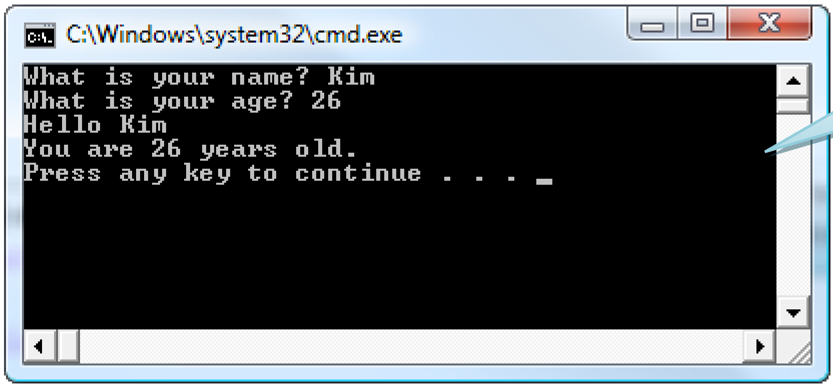
```
The value is 5
```

Program 2-8 shows an example. (This is the Visual Basic version of the pseudocode Program 2-4 in your textbook.)

Program 2-8

This program is the VB version of **Program 2-4** in your textbook!

```
Module Module1

    Sub Main()
        Dim age As Integer
        Dim name As String

        Console.Write("What is your name? ")
        name = Console.ReadLine()

        Console.Write("What is your age? ")
        age = CInt(Console.ReadLine())

        Console.WriteLine("Hello " & name)
        Console.WriteLine("You are " & age & " years old.")
    End Sub

End Module
```



```
What is your name? Kim
What is your age? 26
Hello Kim
You are 26 years old.
Press any key to continue . . . _
```

This is the program's output.

Performing Calculations

Table 2-3 shows the Visual Basic arithmetic operators, which are nearly the same as those presented in Table 2-1 in your textbook.

Table 2-3 Visual Basic's Arithmetic Operators

Symbol	Operation	Description
`+`	Addition	Adds two numbers
`-`	Subtraction	Subtracts one number from another
`*`	Multiplication	Multiplies two numbers
`/`	Division (floating-point)	Divides one number by another and gives the quotient as a `Double`
`\`	Division (Integer)	Divides one number by another and gives the quotient as an `Integer`. (If the quotient has a fractional part, the fractional part is cut off, or *truncated*.)
`MOD`	Modulus	Divides one integer by another and gives the remainder, as an integer
`^`	Exponent	Raises a number to a power and gives the result as a `Double`

Here are some examples of statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

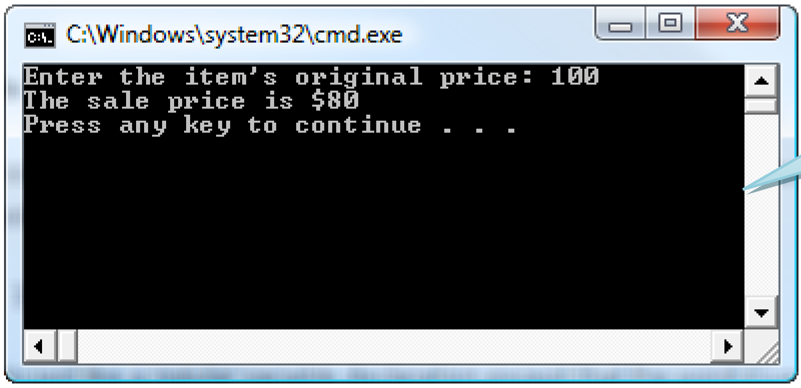
```
total = price + tax
sale = price - discount
population = population * 2
half = number / 2
midpoint = value \ 2
leftOver = 17 MOD 3
area = length ^ 2.0
```

Program 2-9 shows an example program that performs mathematical calculations (This program is the Visual Basic version of pseudocode Program 2-9 in your textbook.)

Program 2-9

This program is the VB version of **Program 2-9** in your textbook!

```
Module Module1

    Sub Main()
        Dim originalPrice, discount, salePrice As Double

        Console.Write("Enter the item's original price: ")
        originalPrice = CDbl(Console.ReadLine())

        discount = originalPrice * 0.2
        salePrice = originalPrice - discount

        Console.WriteLine("The sale price is $" & salePrice)
    End Sub

End Module
```



```
Enter the item's original price: 100
The sale price is $80
Press any key to continue . . .
```

This is the program's output.

Named Constants

You create named constants in Visual Basic by using the `Const` key word in a variable declaration. The word `Const` is written instead of the word `Dim`. Here is an example:

```
Const INTEREST_RATE As Double = 0.069;
```

This statement looks just like a regular variable declaration except that the word `Const` is used instead of `Dim`, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the `Const` key word, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a `Const` variable.

Documenting a Program with Comments

To write a line comment in Visual Basic you simply place an apostrophe (') where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Here is an example:

```
' This program calculates an employee's gross pay.
```

Sometimes programmers write comments prior to a statement, to document what that statement does. Here is an example:

```
' Calculate the gross pay.
grossPay = hours * payRate
```

And sometimes programmers write a comment at the end of a line, to document what that line of code does. Here is an example:

```
grossPay = hours * payRate    ' Calculate the gross pay.
```

Line Continuation

Some programming statements are so long that you have to scroll the code editor horizontally to read the entire line. This can make the code difficult to read on the screen. When this is the case, you can break the statement into several lines of code to make it easier to read.

When you are typing a statement and you reach the point where you want to continue it on the next line, simply type a space, followed by an underscore character, and press then Enter. Here is an example:

```
totalCost = unitsA * price + _
            unitsB * price + _
            unitsC * price
```

This is the same as:

```
totalCost = unitsA * price + unitsB * price + unitsC * price
```

When you want to continue a statement on the next line, remember the following rules:

- A space must appear before the underscore.
- The underscore must be the last character on the line that is being continued.
- You cannot break up a keyword, string literal, or a name (such as a variable name or a control name).

Note: If you are using Visual Basic 2010 or a later version, the underscore character is not required to continue a line of code. Simply press the Enter key at any space character that is not inside of a string literal, and the statement will continue on the next line. In this booklet we will use the underscore anytime we need to continue a line because this technique is compatible with all versions of Visual Basic.

Chapter 3 Procedures

Chapter 3 in your textbook discusses modules as named groups of statements that perform specific tasks in a program. In Visual Basic, we use *sub procedures* for that purpose. Sub procedures are commonly referred to simply as *procedures*, and from this point forward, we will use that term.

In this chapter we will discuss how to define and call Visual Basic procedures, declare local variables in a procedure, and pass arguments to a procedure. We will also discuss the declaration of global variables and global constants.

Note: In Visual Basic, the term module is used differently, than it used in your textbook. A Visual Basic module is a container that holds procedures and other pieces of code. Try not to get confused about this point. When you are reading the textbook and you see the term *module*, think *procedure* in Visual Basic.

Defining and Calling Procedures

To create a procedure you must write its *definition*. Here is an example of a procedure definition:

```
Sub ShowMessage()
    Console.WriteLine("Hello world")
End Sub
```

The first line of a procedure definition starts with the word `Sub`, followed by the procedure's name, followed by a set of parentheses. We refer to this line as the procedure header. In this example, the name of the procedure is `ShowMessage`.

Beginning at the line after the procedure header, one or more statements will appear. These statements are the procedure body, and they are performed when the procedure is executed. In this example, there is one statement in the body of the `ShowMessage` procedure. It uses `Console.WriteLine` to display the string *Hello World* on the screen. The last line of the definition, after the body, reads `End Sub`. This marks the end of the procedure definition.

Calling a Procedure

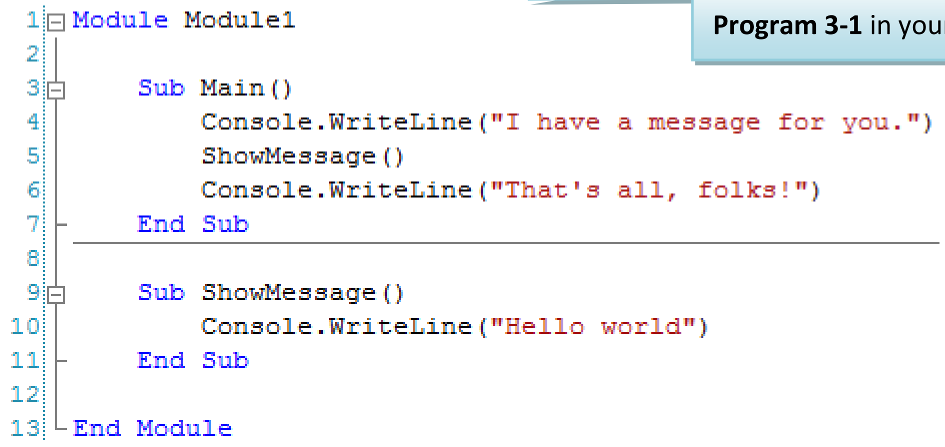
A procedure executes when it is called. The `Main` procedure is automatically called when a Visual Basic console program starts, but other procedures are executed by procedure call statements. When a procedure is called, the program branches to that procedure and executes the statements in its body. Here is an example of a procedure call statement that calls the `ShowMessage` procedure we previously examined:

```
ShowMessage()
```

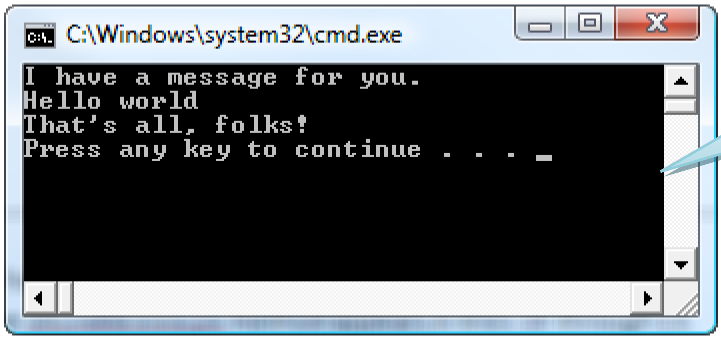
The statement is simply the name of the procedure followed by a set of parentheses. Program 3-1 shows a Visual Basic program that demonstrates the `ShowMessage` procedure. This is the Visual Basic version of pseudocode Program 3-1 in your textbook.

Program 3-1

This program is the VB version of **Program 3-1** in your textbook!



```
 1  Module Module1
 2
 3      Sub Main()
 4          Console.WriteLine("I have a message for you.")
 5          ShowMessage()
 6          Console.WriteLine("That's all, folks!")
 7      End Sub
 8
 9      Sub ShowMessage()
10          Console.WriteLine("Hello world")
11      End Sub
12
13  End Module
```



```
I have a message for you.
Hello world
That's all, folks!
Press any key to continue . . . _
```

This is the program's output.

The module that contains this program's code has two procedures: `Main` and `ShowMessage`. The `Main` procedure appears in lines 3 through 7, and the `ShowMessage` procedure appears in lines 9 through 11. When the program runs, the `Main` procedure executes. The statement in line 4 displays "I have a message for you." Then the statement in line 5 calls the `ShowMessage` procedure. This causes the program to branch to the `ShowMessage` procedure and execute the statement that appears in line 10. This displays "Hello world". The program then branches back to the `Main` procedure and resumes execution at line 6. This displays "That's all, folks!"

Local Variables

Variables that are declared inside a procedure are known as local variables. They are called *local* because they are local to the procedure in which they are declared. Statements outside a procedure cannot access that procedure's local variables. Because a procedure's local variables are hidden from other procedures, the other procedures may have their own local variables with the same name.

Passing Arguments to Procedures

If you want to be able to pass an argument into a procedure, you must declare a parameter variable in that procedure's header. The parameter variable will receive the argument that is passed when the procedure is called. Here is the definition of a procedure that uses a parameter:

```
    Sub DisplayValue(ByVal num As Integer)
        Console.WriteLine("The value is " & num)
    End Sub
```

Notice in the procedure header, inside the parentheses, these words appear:

```
ByVal num As Integer
```

This is the declaration of an `Integer` parameter named `num`. The `ByVal` key word is used instead of the word `Dim`. `ByVal` indicates that any argument passed into the parameter will be passed *by value*. This means that the parameter variable `num` will hold only a copy of the argument that is passed into it. Any changes that are made to the parameter variable will not affect the original argument that was passed into it.

Here is an example of a call to the `DisplayValue` procedure, passing 5 as an argument:

```
DisplayValue(5)
```

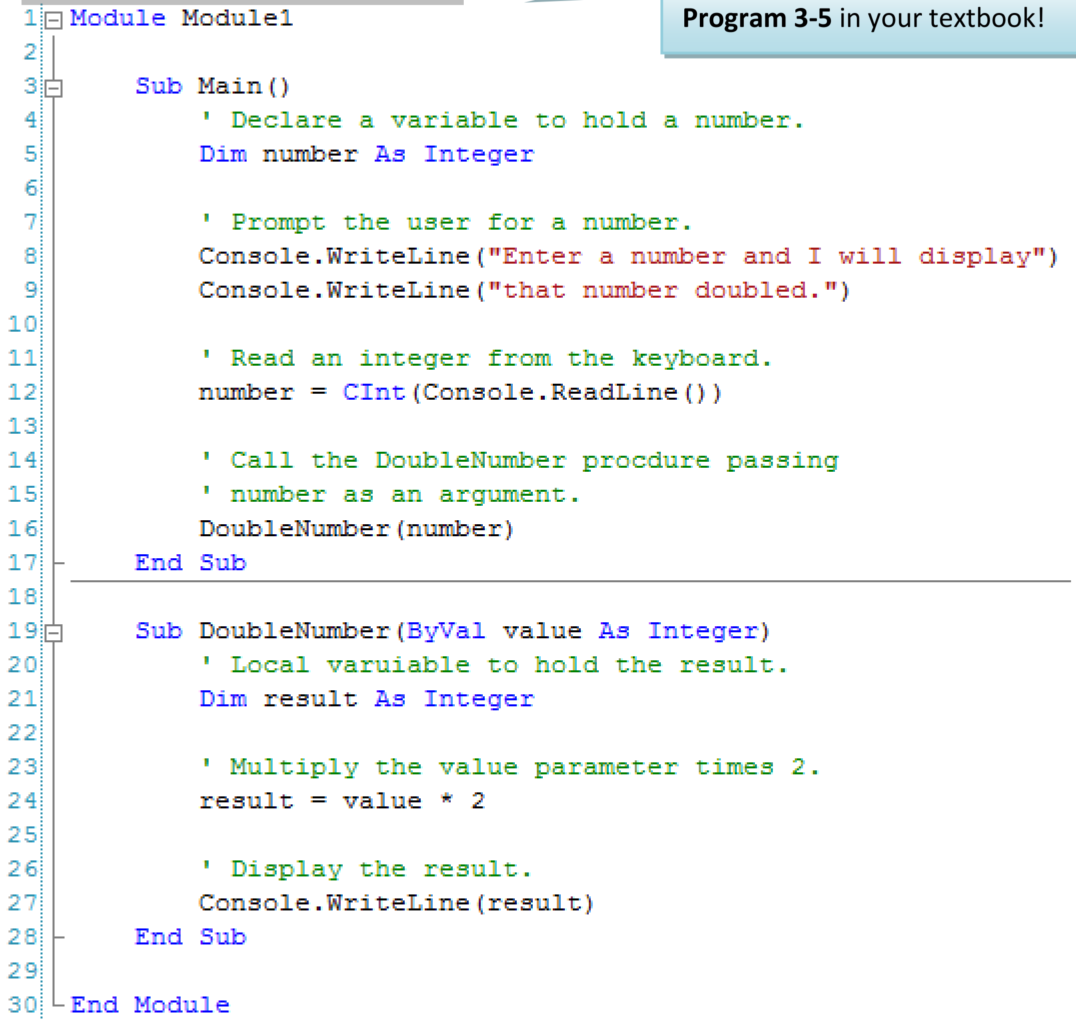
This statement executes the `DisplayValue` procedure. The argument that is listed inside the parentheses is copied into the procedure's parameter variable, `num`.

Program 3-2 shows the Visual Basic version of pseudocode Program 3-5 in your textbook. When the program runs, it prompts the user to enter a number. Line 12 reads an integer from the keyboard and assigns it to the `number` variable. Line 16 calls the `DoubleNumber` procedure, passing the `number` variable as an argument.

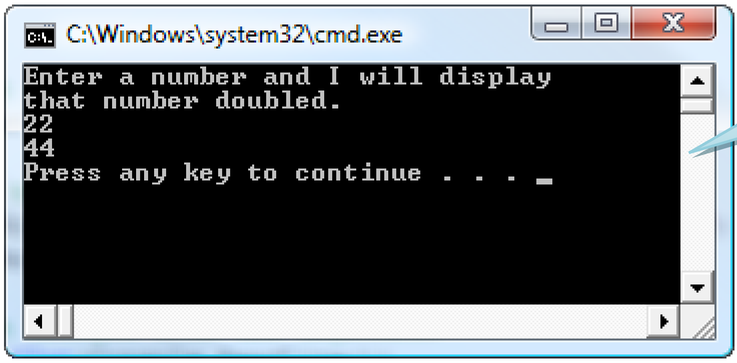
The `DoubleNumber` procedure is defined in lines 19 through 28. The procedure has an `Intgeger` parameter variable named `value`. A local `Intgeger` variable named `result` is declared in line 21, and in line 24 the `value` parameter is multiplied by 2 and the result is assigned to the `result` variable. In line 27 the value of the `result` variable is displayed.

Program 3-2

This program is the VB version of **Program 3-5** in your textbook!



```
 1  Module Module1
 2
 3      Sub Main()
 4          ' Declare a variable to hold a number.
 5          Dim number As Integer
 6
 7          ' Prompt the user for a number.
 8          Console.WriteLine("Enter a number and I will display")
 9          Console.WriteLine("that number doubled.")
10
11          ' Read an integer from the keyboard.
12          number = CInt(Console.ReadLine())
13
14          ' Call the DoubleNumber procdure passing
15          ' number as an argument.
16          DoubleNumber(number)
17      End Sub
18
19      Sub DoubleNumber(ByVal value As Integer)
20          ' Local varuiable to hold the result.
21          Dim result As Integer
22
23          ' Multiply the value parameter times 2.
24          result = value * 2
25
26          ' Display the result.
27          Console.WriteLine(result)
28      End Sub
29
30  End Module
```



```
Enter a number and I will display
that number doubled.
22
44
Press any key to continue . . .
```

This is the program's output.

Passing Multiple Arguments

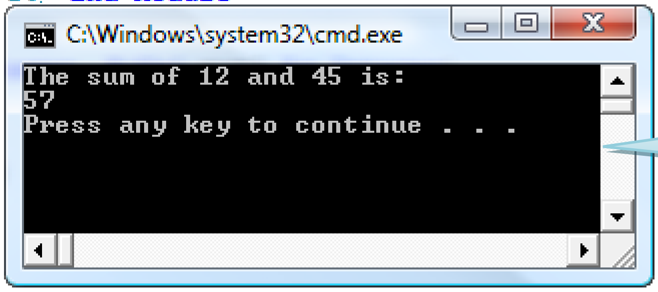
Often it is useful to pass more than one argument to a procedure. When you define a procedure, you must declare a parameter variable for each argument that you want passed into the procedure. Program 3-3 shows an example. This is the Visual Basic version of pseudocode Program 3-6 in your textbook.

This program is the VB version of **Program 3-6** in your textbook!

Program 3-3

```
Module Module1

    Sub Main()
        Console.WriteLine("The sum of 12 and 45 is:")
        ShowSum(12, 45)
    End Sub

    Sub ShowSum(ByVal num1 As Integer, ByVal num2 As Integer)
        Dim result As Integer
        result = num1 + num2
        Console.WriteLine(result)
    End Sub

End Module
```

num1 and num2 parameters



```
The sum of 12 and 45 is:
57
Press any key to continue . . .
```

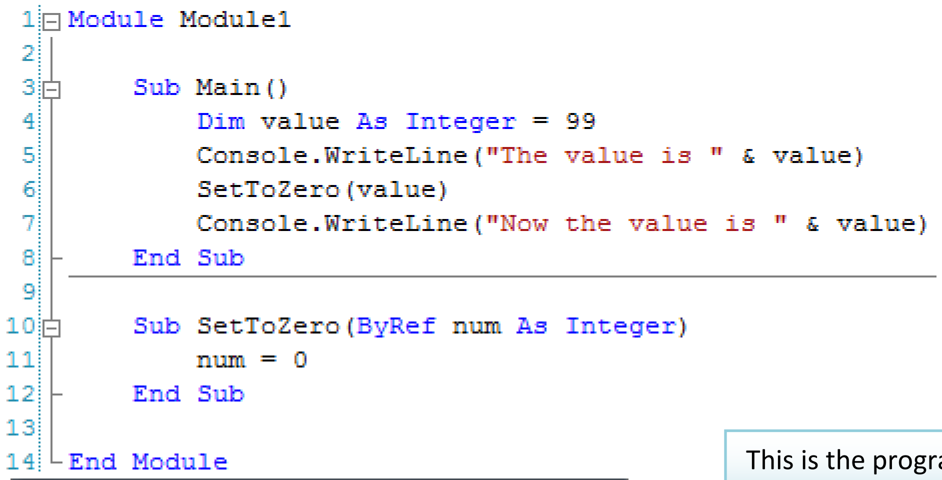
This is the program's output.

Passing Arguments by Reference

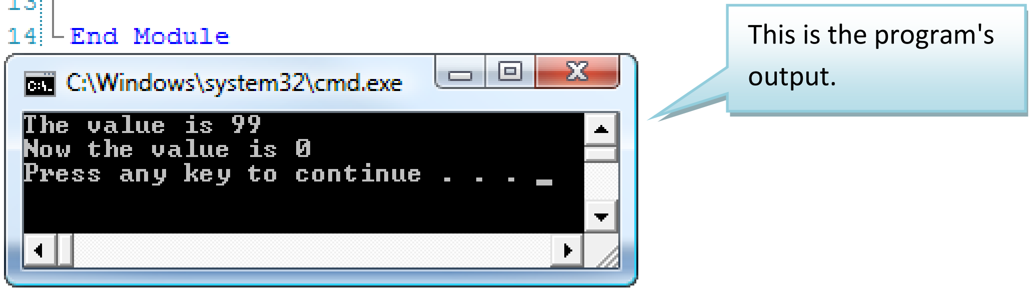
When an argument is passed by reference, it means that the procedure has access to the argument and make changes to it. If you use the word `ByRef` in a parameter variable declaration (instead of the word `ByVal`), then any argument passed into the parameter will be passed by reference.

Program 3-4 shows an example. In the program, an `Integer` argument is passed by reference to the `SetToZero` procedure. The `SetToZero` procedures sets its parameter variable to 0, which also sets the original variable that was passed as an argument to 0.

Program 3-4



```
Module Module1

    Sub Main()
        Dim value As Integer = 99
        Console.WriteLine("The value is " & value)
        SetToZero(value)
        Console.WriteLine("Now the value is " & value)
    End Sub

    Sub SetToZero(ByRef num As Integer)
        num = 0
    End Sub

End Module
```



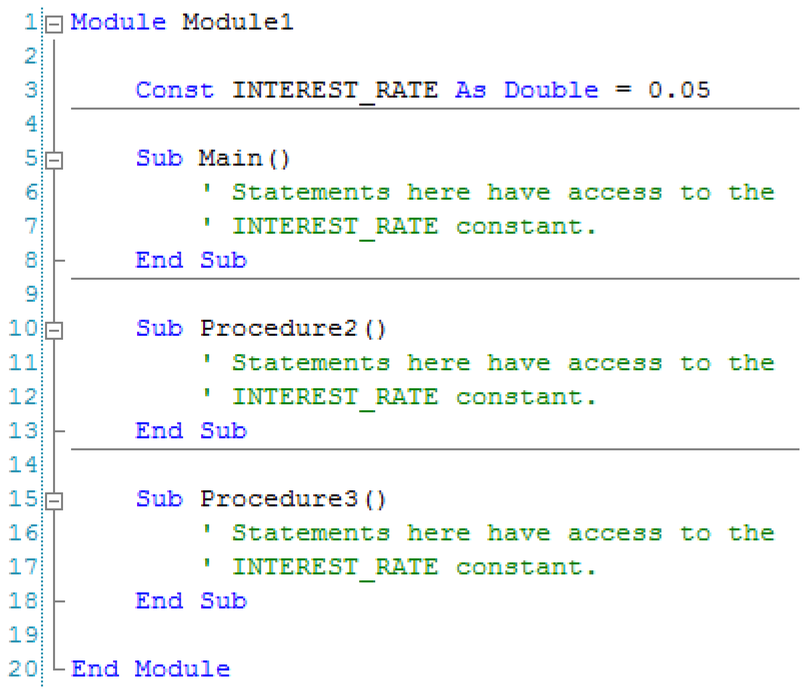
```
The value is 99
Now the value is 0
Press any key to continue . . .
```

Global Variables and Global Constants

To declare a global variable or constant in a Visual Basic Console program, you write the declaration inside the program module, but outside of all procedures. As a result, all of the procedures in the module have access to the variable or constant.

Chapter 3 in your textbook warns against the use of global variables because they make programs difficult to debug. Global constants are permissible, however, because statements in the program cannot change their value. Program 3-5 demonstrates how to declare such a constant. Notice that in line 3 we have declared a constant named `INTEREST_RATE` . The declaration is inside the program module, but it is not inside any of the procedures. As a result, the constant is available to all of the procedures in the program module.

Program 3-5



```
 1 Module Module1
 2
 3     Const INTEREST_RATE As Double = 0.05
 4
 5     Sub Main()
 6         ' Statements here have access to the
 7         ' INTEREST_RATE constant.
 8     End Sub
 9
10     Sub Procedure2()
11         ' Statements here have access to the
12         ' INTEREST_RATE constant.
13     End Sub
14
15     Sub Procedure3()
16         ' Statements here have access to the
17         ' INTEREST_RATE constant.
18     End Sub
19
20 End Module
```

Chapter 4 Decision Structures and Boolean Logic

Relational Operators and the `If-Then` Statement

Visual Basic's relational operators, shown in Table 4-1, are similar to those discussed in your textbook. Notice that the *equal to* operator is only one = sign, and that the *not equal to* operator is <>.

Table 4-1 Relational Operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

The relational operators are used to create Boolean expressions, and are commonly used with `If-Then` statements. Here is the general format of the `If-Then` statement in Visual Basic:

```
If BooleanExpression Then
   statement
   statement
   etc
End If
```

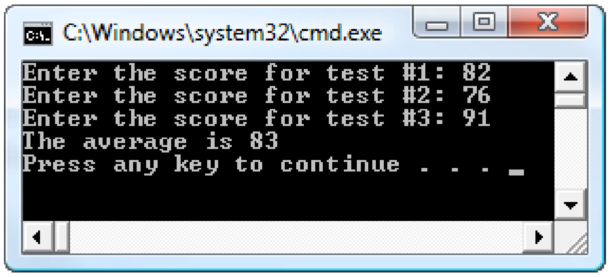
The general format of the statement is exactly like the general format used for pseudocode in your textbook. Program 4-1 demonstrates the `If-Then` statement. This is the Visual Basic version of pseudocode Program 4-1 in your textbook.

Program 4-1

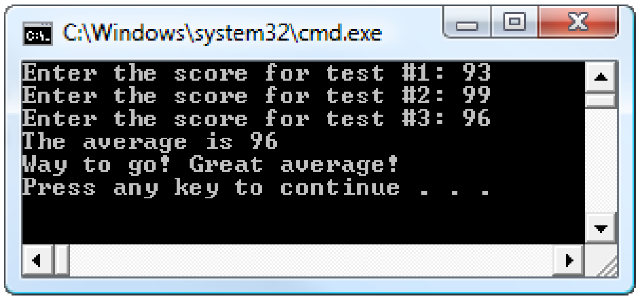
This program is the VB version of **Program 4-1** in your textbook!

```
Module Module1

    Sub Main()
        ' Declare variables
        Dim test1, test2, test3, average As Double

        ' Get test 1.
        Console.Write("Enter the score for test #1: ")
        test1 = CDbl(Console.ReadLine())

        ' Get test 2.
        Console.Write("Enter the score for test #2: ")
        test2 = CDbl(Console.ReadLine())

        ' Get test 3.
        Console.Write("Enter the score for test #3: ")
        test3 = CDbl(Console.ReadLine())

        ' Calculate the average score.
        average = (test1 + test2 + test3) / 3

        ' Display the average.
        Console.WriteLine("The average is " & average)

        ' If the average is greater than 95
        ' congratulate the user.
        If average > 95 Then
            Console.WriteLine("Way to go! Great average!")
        End If
    End Sub

End Module
```

This is the program's output.


C:\Windows\system32\cmd.exe

```
Enter the score for test #1: 82
Enter the score for test #2: 76
Enter the score for test #3: 91
The average is 83
Press any key to continue . . . _
```


C:\Windows\system32\cmd.exe

```
Enter the score for test #1: 93
Enter the score for test #2: 99
Enter the score for test #3: 96
The average is 96
Way to go! Great average!
Press any key to continue . . .
```

Dual Alternative Decision Structures

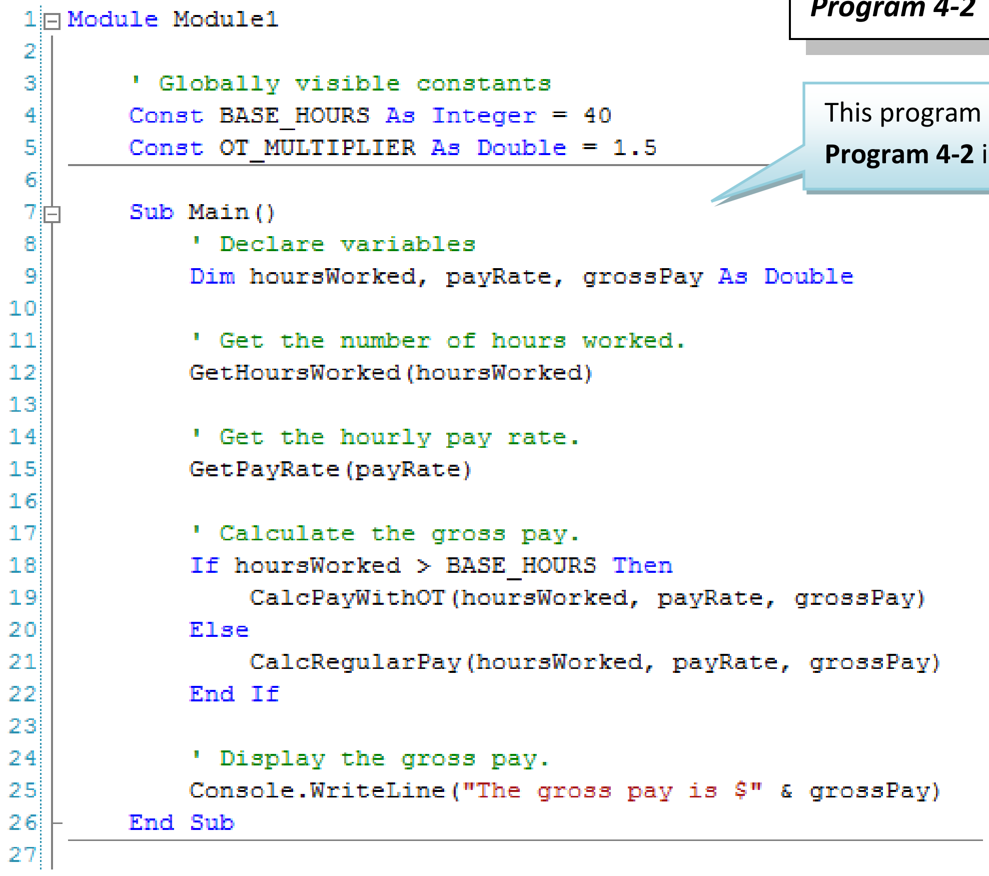
You use the `If-Then-Else` statement in Visual Basic to create a dual alternative decision structure. This is the format of the `If-Then-Else` statement:

```
If BooleanExpression Then
    statement
    statement
    etc
Else
    statement
    statement
    etc
End If
```

The general format of the statement is exactly like the general format used for pseudocode in your textbook. Program 4-2 demonstrates the `If-Then-Else` statement. This is the Visual Basic version of pseudocode Program 4-2 in your textbook.

Program 4-2

This program is the VB version of **Program 4-2** in your textbook!



```
Module Module1

    ' Globally visible constants
    Const BASE_HOURS As Integer = 40
    Const OT_MULTIPLIER As Double = 1.5

    Sub Main()
        ' Declare variables
        Dim hoursWorked, payRate, grossPay As Double

        ' Get the number of hours worked.
        GetHoursWorked(hoursWorked)

        ' Get the hourly pay rate.
        GetPayRate(payRate)

        ' Calculate the gross pay.
        If hoursWorked > BASE_HOURS Then
            CalcPayWithOT(hoursWorked, payRate, grossPay)
        Else
            CalcRegularPay(hoursWorked, payRate, grossPay)
        End If

        ' Display the gross pay.
        Console.WriteLine("The gross pay is $" & grossPay)
    End Sub

```

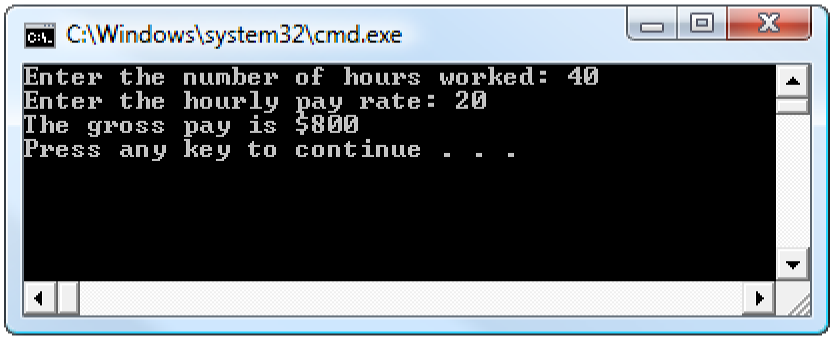
Program continued on next page…

```
    ' The GetHoursWorked procedure gets the number
    ' of hours worled and stores it in the hours parameter.

    Sub GetHoursWorked(ByRef hours As Double)
        Console.Write("Enter the number of hours worked: ")
        hours = CDbl(Console.ReadLine())
    End Sub

    ' The GetPayRate procedure gets the hourly
    ' pay rate and stores it in the rate parameter.

    Sub GetPayRate(ByRef rate As Double)
        Console.Write("Enter the hourly pay rate: ")
        rate = CDbl(Console.ReadLine())
    End Sub

    ' The CalcPayWithOT procedure calculates pay
    ' with overtime. The gross pay is stored
    ' in the gross parameter.

    Sub CalcPayWithOT(ByVal hours As Double, ByVal rate As Double, _
                      ByRef gross As Double)
        ' Local variables
        Dim overtimeHours, overtimePay As Double

        ' Calculate the number of overtime hours.
        overtimeHours = hours - BASE_HOURS

        ' Calculate the overtime pay.
        overtimePay = overtimeHours * rate * OT_MULTIPLIER

        ' Calculate the gross pay.
        gross = BASE_HOURS * rate + overtimePay
    End Sub

```

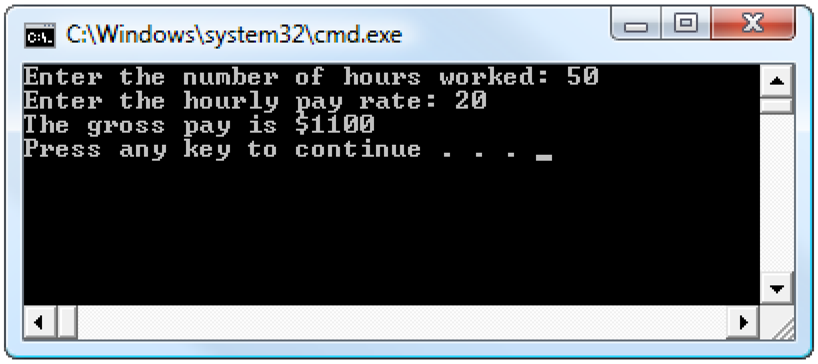
Program continued on next page…

```
    ' The CalcRegularPay procedure calculates
    ' pay with no overtime and stores it in
    ' the gross parameter.

    Sub CalcRegularPay(ByVal hours As Double, ByVal rate As Double, _
                       ByRef gross As Double)
        gross = hours * rate
    End Sub
End Module
```

This is the program's output.



```
Enter the number of hours worked: 40
Enter the hourly pay rate: 20
The gross pay is $800
Press any key to continue . . .
```



```
Enter the number of hours worked: 50
Enter the hourly pay rate: 20
The gross pay is $1100
Press any key to continue . . .
```

Comparing Strings

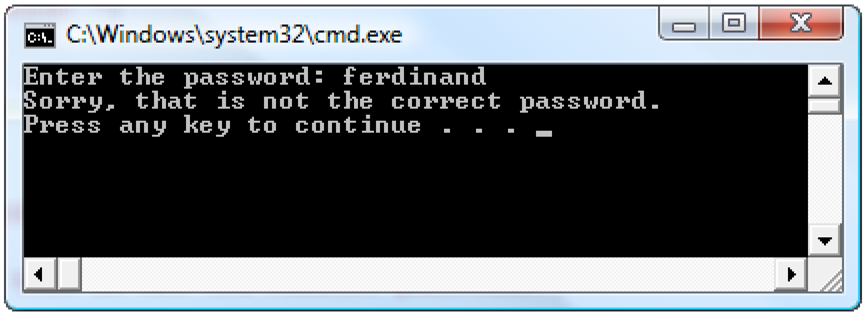
String comparisons in Visual Basic work as described in your textbook, Program 4-3 shows an example. This is the Visual Basic version of pseudocode Program 4-3 in your textbook.

Program 4-3

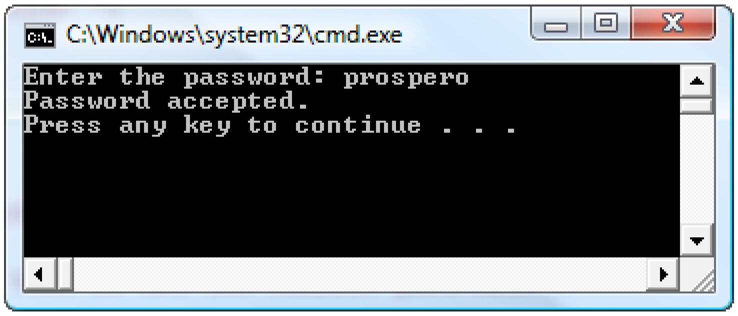
This program is the VB version of **Program 4-3** in your textbook!

```
Module Module1

    Sub Main()
        ' A variable to hold a password.
        Dim password As String

        ' Prompt the user to enter the password.
        Console.Write("Enter the password: ")
        password = Console.ReadLine()

        ' Determine whether the correct password
        ' was entered.
        If password = "prospero" Then
            Console.WriteLine("Password accepted.")
        Else
            Console.WriteLine("Sorry, that is not the correct password.")
        End If
    End Sub

End Module
```

This is the program's output.



```
Enter the password: ferdinand
Sorry, that is not the correct password.
Press any key to continue . . .
```



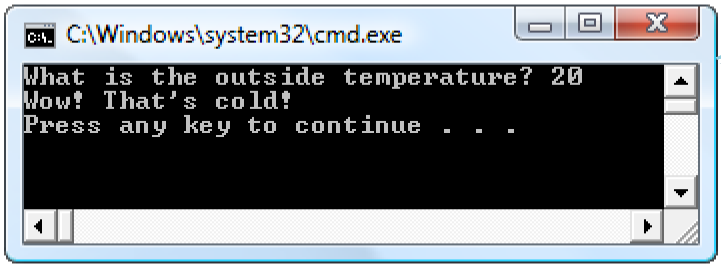
```
Enter the password: prospero
Password accepted.
Press any key to continue . . .
```

Nested Decision Structures

Program 4-4 shows an example of nested decision structures.

Program 4-4

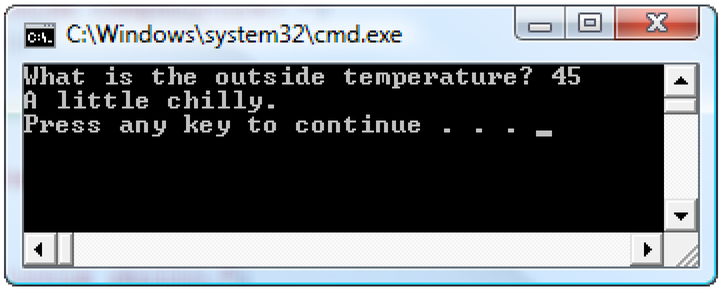
```
Module Module1

    Sub Main()
        ' A variable to hold the temperature.
        Dim temp As Integer

        ' Prompt the user to enter the temperature.
        Console.Write("What is the outside temperature? ")
        temp = CInt(Console.ReadLine())

        ' Determine the type of weather we're having.
        If temp < 30 Then
            Console.WriteLine("Wow! That's cold!")
        Else
            If temp < 50 Then
                Console.WriteLine("A little chilly.")
            Else
                If temp < 80 Then
                    Console.WriteLine("Nice and warm.")
                Else
                    Console.WriteLine("Whew! it's hot!")
                End If
            End If
        End If
    End Sub

End Module
```



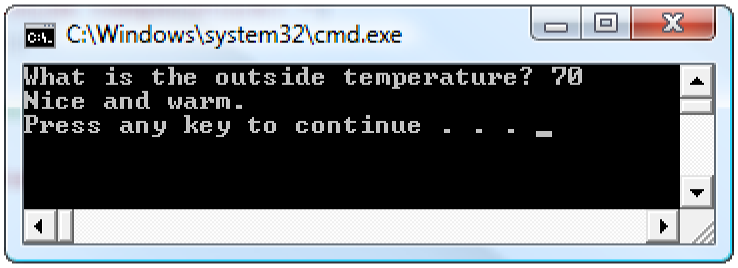
```
What is the outside temperature? 20
Wow! That's cold!
Press any key to continue . . .
```

This is the program's output.

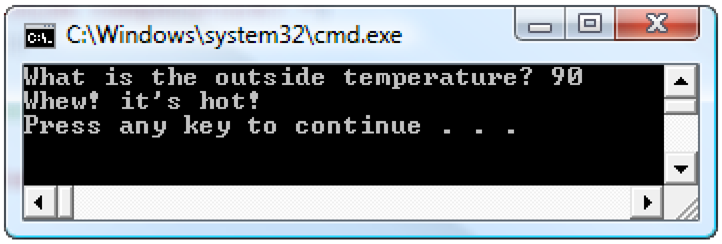
Program output continued on next page…



```
What is the outside temperature? 45
A little chilly.
Press any key to continue . . .
```



```
What is the outside temperature? 70
Nice and warm.
Press any key to continue . . .
```



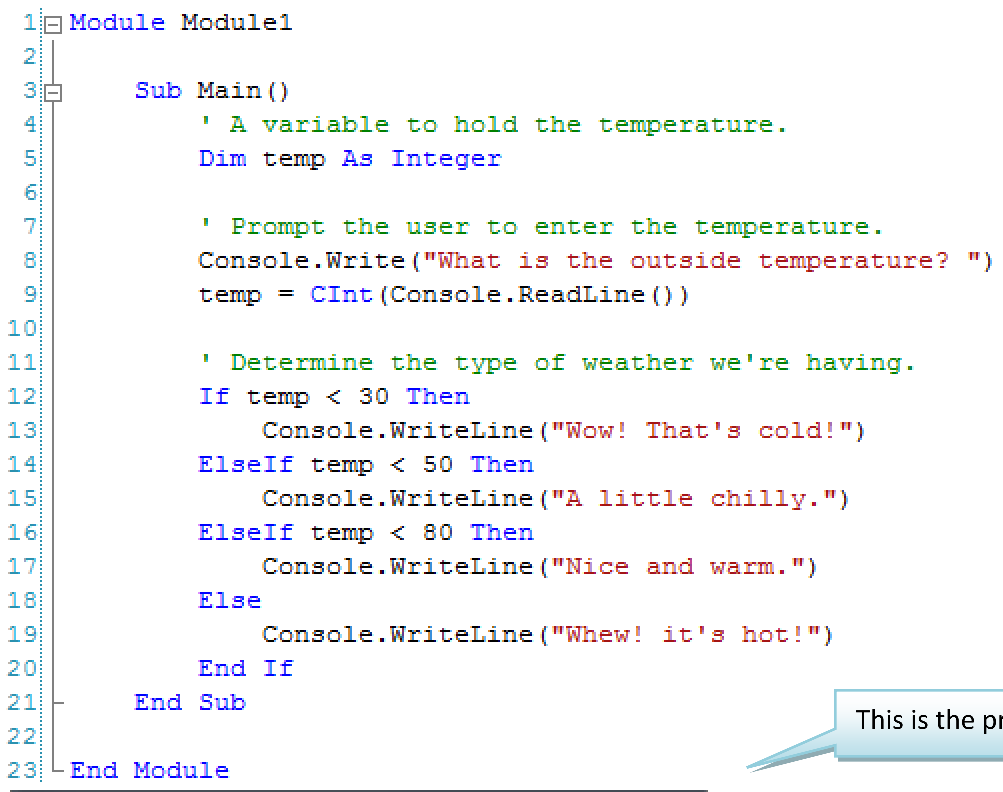
```
What is the outside temperature? 90
Whew! it's hot!
Press any key to continue . . .
```

The `If-Then-ElseIf` Statement

As discussed in your textbook, nested decision logic can sometimes be written with the `If-Then-ElseIf` statement. Note that in Visual Basic, the word `ElseIf` is written with no space separating the words `Else` and `If`.

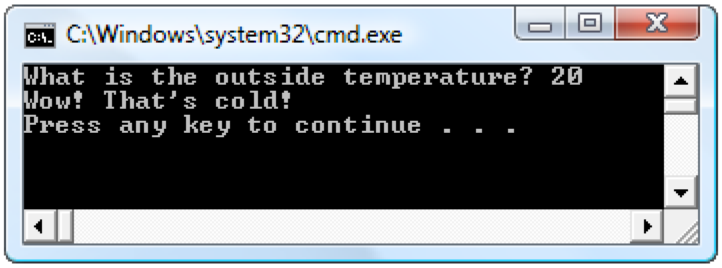
Program 4-5 shows an example. This is a modified version of Program 4-4, and its output is the same as Program 4-4.

Program 4-5

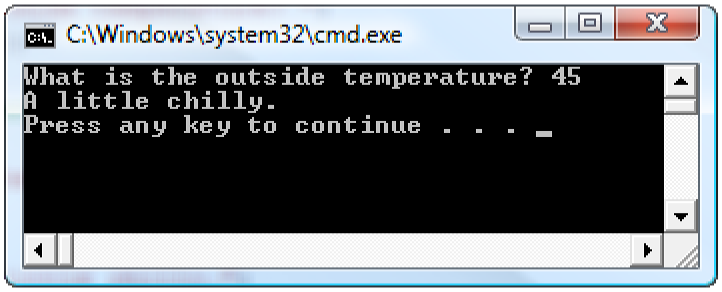


```
Module Module1

    Sub Main()
        ' A variable to hold the temperature.
        Dim temp As Integer

        ' Prompt the user to enter the temperature.
        Console.Write("What is the outside temperature? ")
        temp = CInt(Console.ReadLine())

        ' Determine the type of weather we're having.
        If temp < 30 Then
            Console.WriteLine("Wow! That's cold!")
        ElseIf temp < 50 Then
            Console.WriteLine("A little chilly.")
        ElseIf temp < 80 Then
            Console.WriteLine("Nice and warm.")
        Else
            Console.WriteLine("Whew! it's hot!")
        End If
    End Sub

End Module
```

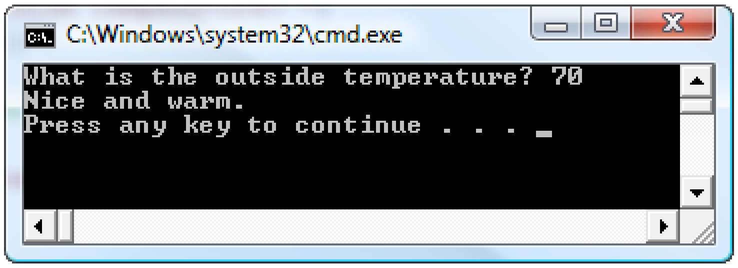
This is the program's output.



```
What is the outside temperature? 20
Wow! That's cold!
Press any key to continue . . .
```



```
What is the outside temperature? 45
A little chilly.
Press any key to continue . . . _
```



```
What is the outside temperature? 70
Nice and warm.
Press any key to continue . . . _
```

The Case Structure (`Select Case` Statement)

In Visual Basic, case structures are written as `Select Case` statements. Here is the general format of the `Select Case` statement:

```
Select Case (testExpression)        ←    This is a variable or expression.

    Case value_1
        statement                  }     These statements are executed if the
        statement                  }     testExpression is equal to value_1.
        etc.                       }

    Case value_2
        statement                  }     These statements are executed if the
        statement                  }     testExpression is equal to value_2.
        etc.                       }

   Insert as many Case sections as necessary

    Case value_N
        statement                  }     These statements are executed if the
        statement                  }     testExpression is equal to value_N.
        etc.                       }

    Case Else
        statement                  }     These statements are executed if the
        statement                  }     testExpression is not equal to any
        etc.                       }     of the values listed after the Case
                                         statements.
End Select                          ←    This is the end of the statement.
```

Note that in your textbook, the pseudocode `Case` statements end with a colon, but in Visual Basic they do not. Also note that Visual Basic uses a `Case Else` statement at the end of the structure instead of a `Default` section.

For example, the following code performs the same operation as the flowchart in Figure 4-18:

```
Select Case month
   Case 1
      Console.WriteLine("January")

   Case 2
      Console.WriteLine("February")

   Case 3
      Console.WriteLine("March")

   Case Else
      Console.WriteLine("Error: Invalid month")
End Select
```

In this example the *testExpression* is the `month` variable. If the value in the `month` variable is 1, the program will branch to the `Case 1` section and execute the `Console.WriteLine("January")` statement that immediately follows it. If the value in the `month` variable is 2, the program will branch to the `Case 2` section and execute the `Console.WriteLine("February")` statement that immediately follows it. If the value in the `month` variable is 3, the program will branch to the `Case 3` section and execute the `Console.WriteLine("March")` statement that immediately follows it. If the value in the `month` variable is not 1, 2, or 3, the program will branch to the `Case Else` section and execute the `Console.WriteLine("Error: Invalid month")` statement that immediately follows it.

Note: The `Case Else` section is optional, but in most situations you should have one. The `Case Else` section is executed when the *testExpression* does not match any of the `Case` values.

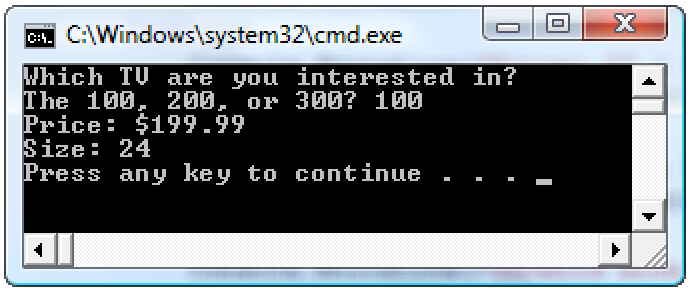
Program 4-7 shows a complete example. This is the Visual Basic version of pseudocode Program 4-8 in your textbook.

Program 4-7

This program is the VB version of **Program 4-8** in your textbook!

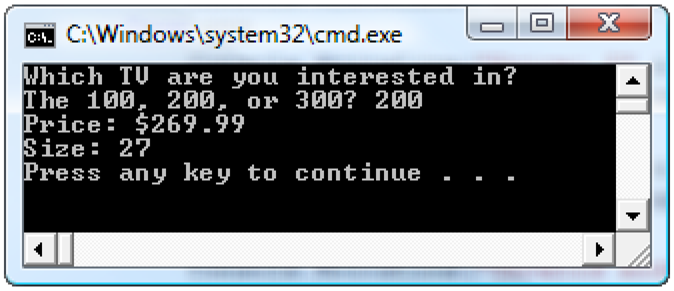
```
Module Module1

    Sub Main()
        ' Constants for the TV prices
        Const MODEL_100_PRICE As Double = 199.99
        Const MODEL_200_PRICE As Double = 269.99
        Const MODEL_300_PRICE As Double = 349.99

        ' Constants for the TV sizes
        Const MODEL_100_SIZE As Double = 24
        Const MODEL_200_SIZE As Double = 27
        Const MODEL_300_SIZE As Double = 32

        ' Variable for the model number.
        Dim modelNumber As Integer

        ' Get the model number.
        Console.WriteLine("Which TV are you interested in?")
        Console.Write("The 100, 200, or 300? ")
        modelNumber = CInt(Console.ReadLine())

        ' Display the price and size.
        Select Case modelNumber
            Case 100
                Console.WriteLine("Price: $" & MODEL_100_PRICE)
                Console.WriteLine("Size: " & MODEL_100_SIZE)
            Case 200
                Console.WriteLine("Price: $" & MODEL_200_PRICE)
                Console.WriteLine("Size: " & MODEL_200_SIZE)
            Case 300
                Console.WriteLine("Price: $" & MODEL_300_PRICE)
                Console.WriteLine("Size: " & MODEL_300_SIZE)
            Case Else
                Console.WriteLine("Invalid model number.")
        End Select
    End Sub

End Module
```

This is the program's output.

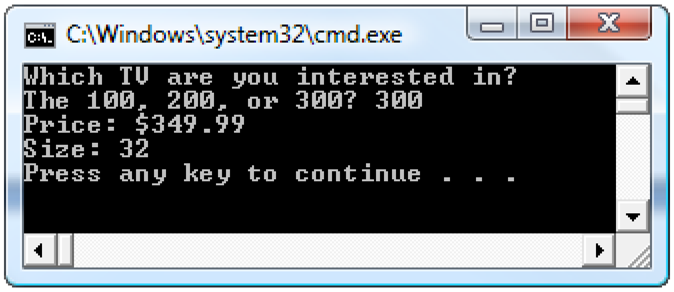


```
Which TV are you interested in?
The 100, 200, or 300? 100
Price: $199.99
Size: 24
Press any key to continue . . .
```

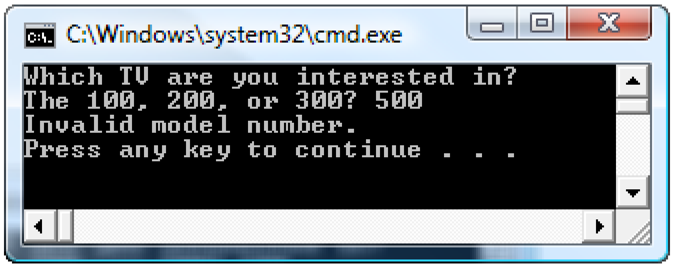
Program output continued on next page…



```
Which TV are you interested in?
The 100, 200, or 300? 200
Price: $269.99
Size: 27
Press any key to continue . . .
```



```
Which TV are you interested in?
The 100, 200, or 300? 300
Price: $349.99
Size: 32
Press any key to continue . . .
```



```
Which TV are you interested in?
The 100, 200, or 300? 500
Invalid model number.
Press any key to continue . . .
```

Logical Operators

Visual Basic provides a more comprehensive set of logical operators than many languages. Table 4-2 shows the Visual Basic logical operators.

For example, the following `If-Then` statement checks the value in `x` to determine if it is in the range of 20 through 40:

```
If x >= 20 And x <= 40 then
   Console.WriteLine(x & " is in the acceptable range.")
End If
```

The Boolean expression in the `If-Then` statement will be true only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be true. The following statement determines whether `x` is outside the range of 20 through 40:

```
If x < 20 Or x > 40 Then
   Console.WriteLine(x & " is outside the acceptable range.")
End If
```

Here is an `If-Then` statement using the `Not` operator:

```
if Not (temperature > 100) Then
   Console.WriteLine("This is below the maximum temperature.")
End If
```

First, the expression `(temperature > 100)` is tested and a value of either true or false is the result. Then the `Not` operator is applied to that value. If the expression `(temperature > 100)` is true, the `Not` operator returns false. If the expression `(temperature > 100)` is false, the `Not` operator returns true. The previous code is equivalent to asking: "Is the temperature not greater than 100?"

Table 4-2 Visual Basic's Logical Operators

Operator	Meaning
`And`	Combines two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true. (*Note: this operator does not perform short-circuit evaluation.*)
`Or`	Combines two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which. (*Note: this operator does not perform short-circuit evaluation.*)
`Not`	The `Not` operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `Not` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.
`Xor`	This operator performs an exclusive OR operation. It combines two Boolean expressions into one compound expression. One of the subexpressions, but not both, must be true for the compound expression to be true. If both subexpressions are true, or both subexpressions are false, the overall expression is false.
`AndAlso`	This is a logical AND operator that performs short-circuit evaluation. (See the discussion of short-circuit evaluation in your textbook for details.)
`OrElse`	This is a logical OR operator that performs short-circuit evaluation. (See the discussion of short-circuit evaluation in your textbook for details.)

Boolean Variables

In Visual Basic you use the `Boolean` data type to create Boolean variables. A `Boolean` variable can hold one of two possible values: `True` or `False`. Here is an example of a `Boolean` variable declaration:

```
Dim highScore As Boolean
```

`Boolean` variables are commonly used as flags that signal when some condition exists in the program. When the flag variable is set to `False`, it indicates the condition does not yet exist. When the flag variable is set to `True`, it means the condition does exist.

For example, suppose a test grading program has a `Boolean` variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```
If average > 95 Then
   highScore = true
End If
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine whether a high score has been achieved:

```
If highScore Then
   Console.WriteLine("That's a high score!")
End If
```

Chapter 5 Repetition Structures

You textbook describes the logic of the following condition-controlled loops:

- The `While` loop (a pretest condition-controlled loop)
- The `Do-While` loop (a posttest condition-controlled loop)
- The `Do-Until` loop (also a posttest condition-controlled loop).

In Visual Basic, both the `While` loop and the `Do-While` loop are written as variations of the `Do-While` loop. In other words, the `Do-While` loop in Visual Basic can be written as either a pretest or a posttest loop. Here is the general format of how you write a pretest `Do-While` loop:

```
Do While BooleanExpression
    statement
    statement
    etc
Loop
```

The line that begins `Do While` is the beginning of the loop and the line that reads `Loop` is the end of the loop. The statements that appear between these lines are known as the body of the loop.

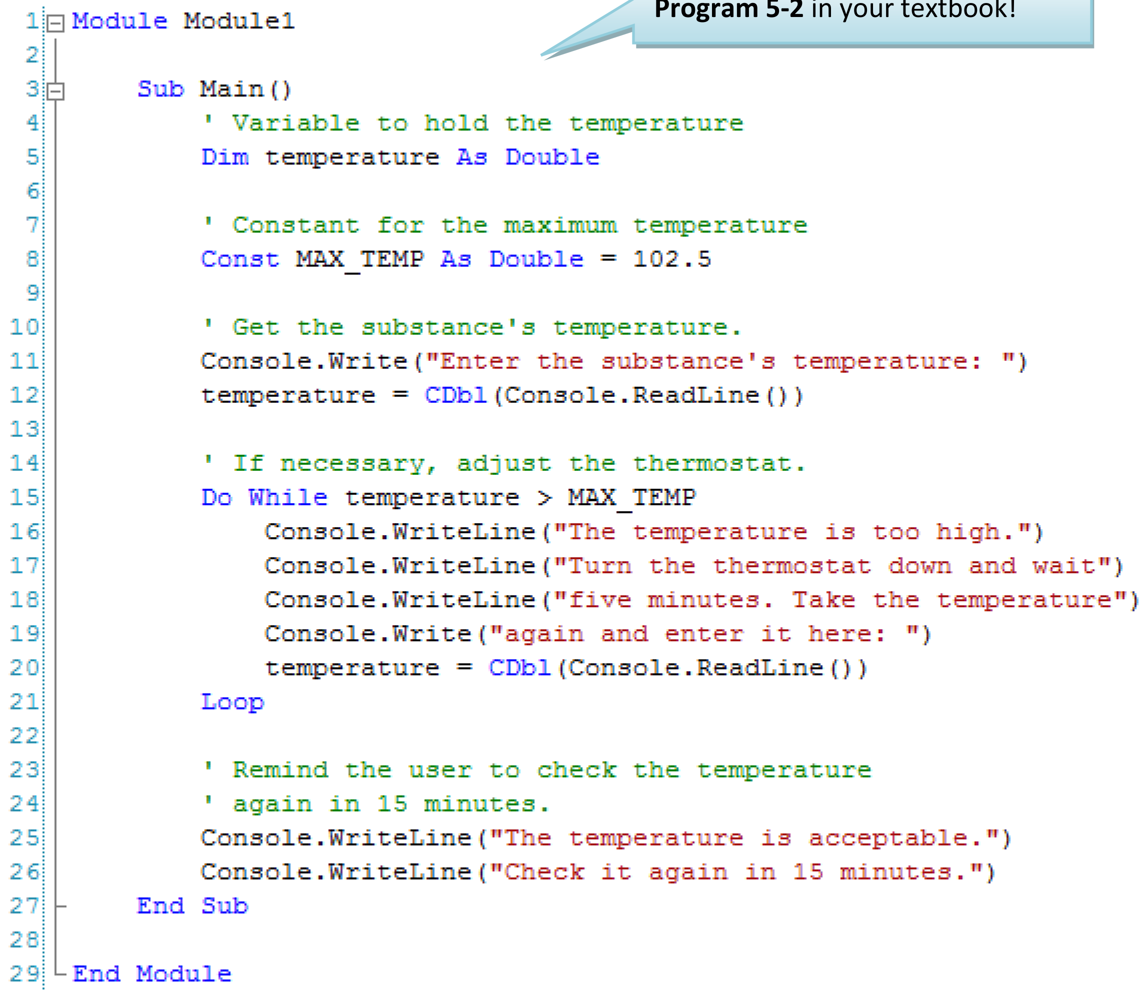
When the `Do-while` loop executes, the *`BooleanExpression`* is tested. If the *`BooleanExpression`* is true, the statements that appear in the body of the loop are executed, and then the loop starts over. If the *`BooleanExpression`* is false, the loop ends and the program resumes execution at the statement immediately following the loop.

We say that the statements in the body of the loop are conditionally executed because they are executed only under the condition that the *`BooleanExpression`* is true.

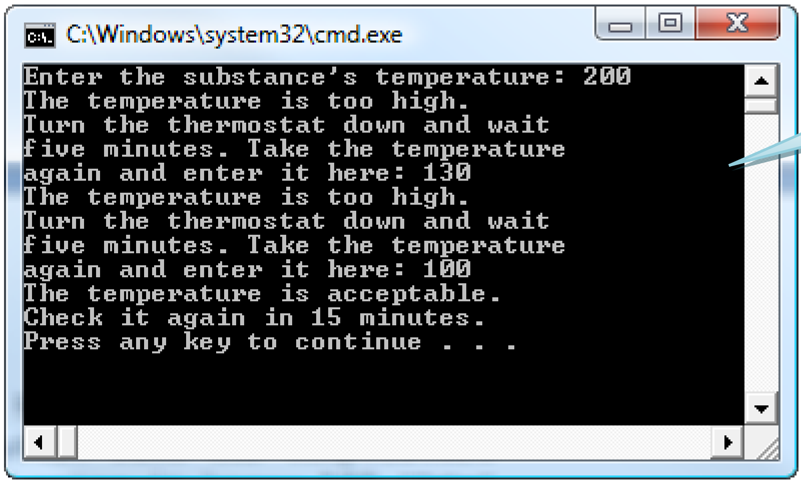
Program 5-1 shows an example of the `Do-While` loop. This is the Visual Basic version of pseudocode Program 5-2 in your textbook.

Program 5-1

This program is the VB version of **Program 5-2** in your textbook!



```
Module Module1

    Sub Main()
        ' Variable to hold the temperature
        Dim temperature As Double

        ' Constant for the maximum temperature
        Const MAX_TEMP As Double = 102.5

        ' Get the substance's temperature.
        Console.Write("Enter the substance's temperature: ")
        temperature = CDbl(Console.ReadLine())

        ' If necessary, adjust the thermostat.
        Do While temperature > MAX_TEMP
            Console.WriteLine("The temperature is too high.")
            Console.WriteLine("Turn the thermostat down and wait")
            Console.WriteLine("five minutes. Take the temperature")
            Console.Write("again and enter it here: ")
            temperature = CDbl(Console.ReadLine())
        Loop

        ' Remind the user to check the temperature
        ' again in 15 minutes.
        Console.WriteLine("The temperature is acceptable.")
        Console.WriteLine("Check it again in 15 minutes.")
    End Sub

End Module
```



```
Enter the substance's temperature: 200
The temperature is too high.
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here: 130
The temperature is too high.
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here: 100
The temperature is acceptable.
Check it again in 15 minutes.
Press any key to continue . . .
```

This is the program's output.

The Posttest `Do-While` Loop

Here is the general format of the posttest `Do-While` loop:

```
Do
    statement
    statement
    etc
Loop While BooleanExpression
```

This loop works like the `Do-While` loop that is described in your textbook.

The `Do-Until` Loop

As described in your textbook, the `Do-Until` loop is a posttest loop that iterates until a Boolean expression is true. Here is the general format of the `Do-Until` loop in Visual Basic:

```
Do
    statement
    statement
    etc
Loop Until BooleanExpression
```

This loop works like the `Do-Until` loop that is described in your textbook.

The `For…Next` Loop

The `For...Next` in Visual Basic works like the `For` loop described in your textbook. It is ideal for situations that require a counter because it initializes, tests, and increments a counter variable. Here is a simplified version of the format of the `For...Next` loop:

```
For CounterVariable = StartValue To EndValue
    statement
    statement
    etc.
Next
```

Here's a summary of the syntax:

- `CounterVariable` is the variable to be used as a counter. It must be a numeric variable.
- `StartValue` is the value the counter variable will be initially set to. This value must be numeric.
- `EndValue` is the value the counter variable is tested against just prior to each iteration of the loop. This value must be numeric.

Here is an example of a simple `For`…`Next` loop that prints "Hello" five times (assuming the variable `count` has already been declared as an `Integer`):

```
For count = 1 To 5
    Console.WriteLine("Hello")
Next
```

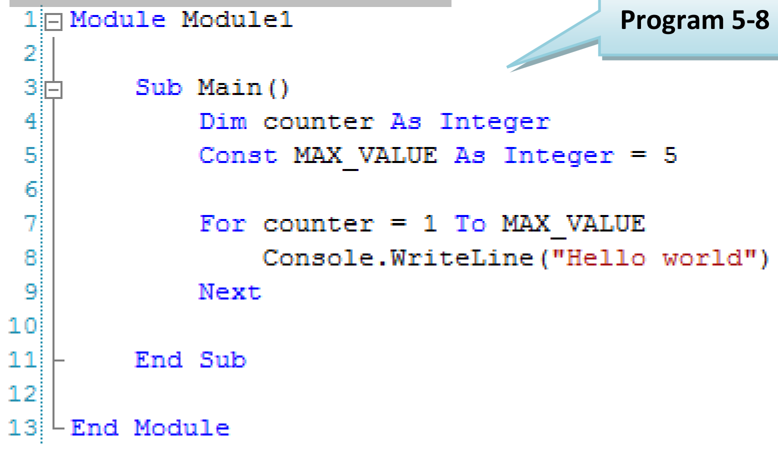
This loop executes the statement `Console.WriteLine("Hello")` 10 times. The following steps take place when the loop executes:

1. `count` is set to 1 (the start value).
2. `count` is compared to 5 (the end value). If `count` is less than or equal to 5, continue to Step 3. Otherwise the loop is exited.
3. The statement `Console.WriteLine("Hello")`is executed.
4. `count` is incremented by 1.
5. Go back to Step 2 and repeat this sequence.

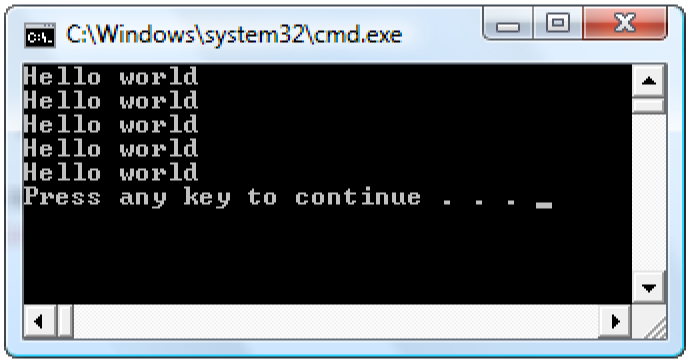
Program 5-2 shows an example. This is the Visual Basic version of pseudocode Program 5-8 in your textbook.

Program 5-2

This program is the VB version of **Program 5-8** in your textbook!



```
Module Module1

    Sub Main()
        Dim counter As Integer
        Const MAX_VALUE As Integer = 5

        For counter = 1 To MAX_VALUE
            Console.WriteLine("Hello world")
        Next

    End Sub

End Module
```



```
Hello world
Hello world
Hello world
Hello world
Hello world
Press any key to continue . . . _
```

This is the program's output.

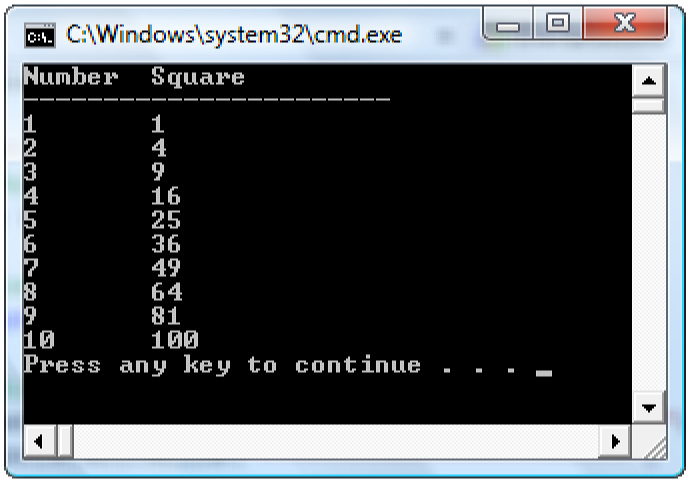
Program 5-3 shows another example. The `For`…`Next` loop in this program uses the value of the counter variable in a calculation in the body of the loop. This is the Visual Basic version of pseudocode Program 5-9 in your textbook.

I should point out that `vbTab`, which appears in lines 11 and 18 works similarly to the word `Tab` that is used in pseudocode in your textbook. As you can see in the program output, `vbTab` causes the output cursor to "tab over." It is useful for aligning output in columns on the screen.

Program 5-3

This program is the VB version of **Program 5-9** in your textbook!

```
 1  Module Module1
 2  
 3      Sub Main()
 4          ' Variables
 5          Dim counter, square As Integer
 6  
 7          ' Constant for the maximum value
 8          Const MAX_VALUE As Integer = 10
 9  
10          ' Display the table headings.
11          Console.WriteLine("Number" & vbTab & "Square")
12          Console.WriteLine("-----------------------")
13  
14          ' Display the numbers 1 through 10 and
15          ' their squares.
16          For counter = 1 To MAX_VALUE
17              square = counter * counter
18              Console.WriteLine(counter & vbTab & square)
19          Next
20  
21      End Sub
22  
23  End Module
```



```
Number  Square
-----------------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
Press any key to continue . . . _
```

This is the program's output.

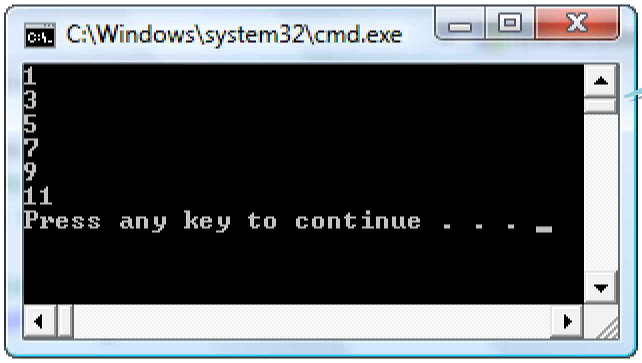
Incrementing by Values Other Than 1

In Visual Basic you can use a `Step` clause in a `For`…`Next` loop to specify the loop's step amount. It works like the pseudocode `Step` clause that is described in your textbook. Program 5-4 demonstrates how a `Step` clause can be used to increment the counter variable by 2. This causes the statement in line 11 to display only the odd numbers in the range of 1 through 11. This program is the Visual Basic version of pseudocode Program 5-10 in your textbook.

Program 5-4

```
1  Module Module1
2
3      Sub Main()
4          ' Declare a counter variable
5          Dim counter As Integer
6
7          ' Constant for the maximum value
8          Const MAX_VALUE As Integer = 11
9
10         For counter = 1 To MAX_VALUE Step 2
11             Console.WriteLine(counter)
12         Next
13
14     End Sub
15
16 End Module
```

This program is the VB version of **Program 5-10** in your textbook!



```
1
3
5
7
9
11
Press any key to continue . . .
```

This is the program's output.

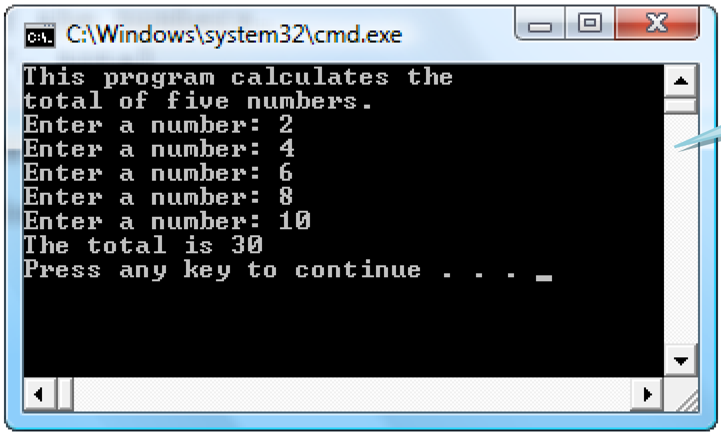
Calculating a Running Total

Your textbook discusses the common programming task of calculating the sum of a series of values, also known as calculating a running total. Program 5-5 demonstrates how this is done in Visual Basic. The `total` variable that is declared in line 10 is the accumulator variable. Notice that it is initialized with the value 0. During each loop iteration the user enters a number, and in line 23 this number is added to the value already stored in `total`. The `total` variable accumulates the sum of all the numbers entered by the user. This program is the Visual Basic version of pseudocode Program 5-18 in your textbook.

Program 5-4

This program is the VB version of **Program 5-18** in your textbook!

```
Module Module1

    Sub Main()
        ' Declare a variable to hold each number
        ' entered by the user.
        Dim number As Integer

        ' Declare an accumulator variable,
        ' initialized with 0.
        Dim total As Integer = 0

        ' Declare a counter variable for the loop.
        Dim counter As Integer

        ' Explain what we are doing.
        Console.WriteLine("This program calculates the")
        Console.WriteLine("total of five numbers.")

        ' Get five numbers and accumulate them.
        For counter = 1 To 5
            Console.Write("Enter a number: ")
            number = CInt(Console.ReadLine())
            total = total + number
        Next

        ' Display the total of the numbers.
        Console.WriteLine("The total is " & total)
    End Sub

End Module
```



```
This program calculates the
total of five numbers.
Enter a number: 2
Enter a number: 4
Enter a number: 6
Enter a number: 8
Enter a number: 10
The total is 30
Press any key to continue . . . _
```

This is the program's output.

Chapter 6 Functions

Generating Random Integers

To generate random numbers in Visual Basic, you have to create a special type of object known as a `Random` object in memory. Random objects have methods and properties that make generating random numbers fairly easy. Here is an example of a statement that creates a Random object:

```
Dim rand As New Random
```

This statement declares a variable named `rand`. The expression `New Random` creates a `Random` object in memory. After this statement executes, the `rand` variable will refer to the `Random` object. As a result, you will be able to use the `rand` variable to call the object's methods for generating random numbers. (There is nothing special about the variable name `rand` used in this example. You can use any legal variable name.)

The `Next` Method

Once you have created a `Random` object, you can call its `Next` method to get a random integer number. The following code shows an example:

```
' Declare an Integer variable.
Dim number As Integer

' Create a Random object.
Dim rand As New Random

' Get a random integer and assign it to the number variable.
number = rand.Next()
```

After this code executes, the `number` variable will contain a random number. If you call the `Next` method with no arguments, as shown in this example, the returned integer is somewhere between 0 and 2,147,483,647. Alternatively, you can pass an argument that specifies an upper limit to the generated number’s range. In the following statement, the value assigned to `number` is somewhere between 0 and 99:

```
number = rand.Next(100)
```

The random integer's range does not have to begin at zero. You can add or subtract a value to shift the numeric range upward or downward. In the following statement, we call the Next method to get a random number in the range of 0 through 9, and then we add 1 to it. So, the number assigned to the `number` variable will be somewhere in the range of 1 through 10:

```
number = rand.Next(10) + 1
```

The following statement shows another example. It assigns a random integer between –50 and +49 to `number`:

```
number = rand.Next(100) - 50
```

The `NextDouble` Method

You can call a `Random` object's `NextDouble` method to get a random floating-point number between 0.0 and 1.0 (not including 1.0). The following code shows an example:

```
' Declare an a Double variable.
Dim number As Double

' Create a Random object.
Dim rand As New Random

' Get a random number and assign it to the number variable.
number = rand.NextDouble()
```

After this code executes, the `number` variable will contain a random floating-point number in the range of 0.0 up to (but not including) 1.0. If you want the random number to fall within a larger range, multiply it by a scaling factor. The following statement assigns a random number between 0.0 and 500.0 to `number`:

```
number = rand.NextDouble() * 500.0
```

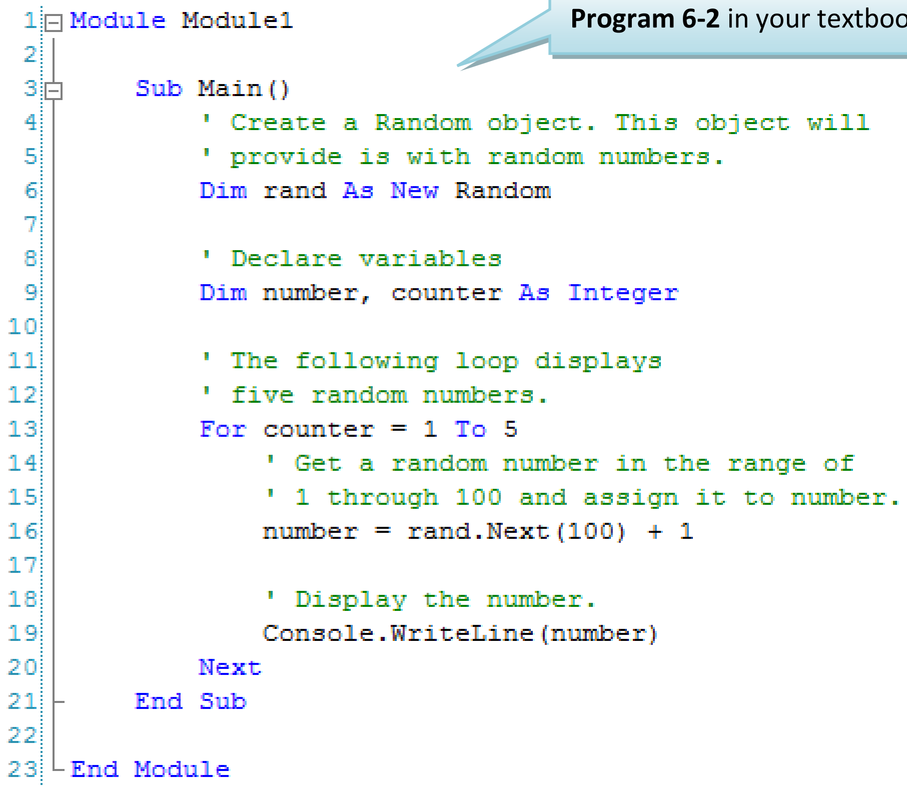
The following statement generates a random number between 100.0 and 600.0:

```
number = (rand.NextDouble() * 500.0) + 100.0
```

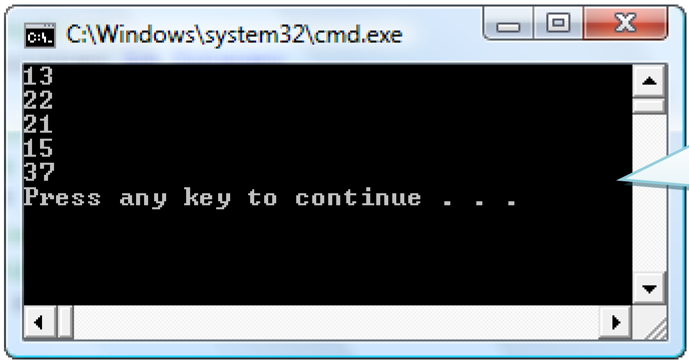
Program 6-1 shows a complete demonstration. This is the Visual Basic version of pseudocode Program 6-2 in your textbook.

Program 6-1

This program is the VB version of **Program 6-2** in your textbook!



```
 1 Module Module1
 2
 3     Sub Main()
 4         ' Create a Random object. This object will
 5         ' provide is with random numbers.
 6         Dim rand As New Random
 7
 8         ' Declare variables
 9         Dim number, counter As Integer
10
11         ' The following loop displays
12         ' five random numbers.
13         For counter = 1 To 5
14             ' Get a random number in the range of
15             ' 1 through 100 and assign it to number.
16             number = rand.Next(100) + 1
17
18             ' Display the number.
19             Console.WriteLine(number)
20         Next
21     End Sub
22
23 End Module
```



```
13
22
21
15
37
Press any key to continue . . .
```

This is the program's output. (Remember, these are random numbers, so they will be different each time the program runs.)

Let's take a closer look at the code:

- The statement in Line 6 creates a `Random` object in memory and gives it the name `rand`.
- Line 9 declares two `Integer` variables: `counter` and `number`. The `counter` variable will be used in a `For`…`Next` loop, and the `number` variable will be used to hold random numbers.

- The `For…Next` loop that begins in line 13 iterates 5 times.
- Inside the loop, the statement in line 16 generates a random number in the range of 0 through 100 and assigns it to the `number` variable.
- The statement in line 19 displays the value of the `number` variable.

Writing Your Own Functions

Here is the general format for writing functions in Visual Basic:

```
Function FunctionName (ParameterList) As DataType
    statement
    statement
    etc.
    Return value
End Function
```

A function must have a `Return` statement. This causes a value to be sent back to the part of the program that called the function.

The first line, known as the function header, begins with the word `Function` and is followed by these items:

- *`FunctionName`* is the name of the function.
- An optional parameter list appears inside a set of parentheses. If the function does not accept arguments, then an empty set of parentheses will appear.
- *`DataType`* is the data type of the value that the function returns. For example, if the function returns an `Integer`, the word `Integer` will appear here. If the function returns a `Double`, the word `Double` will appear here. Likewise, if the function returns a `String`, the word `String` will appear here.

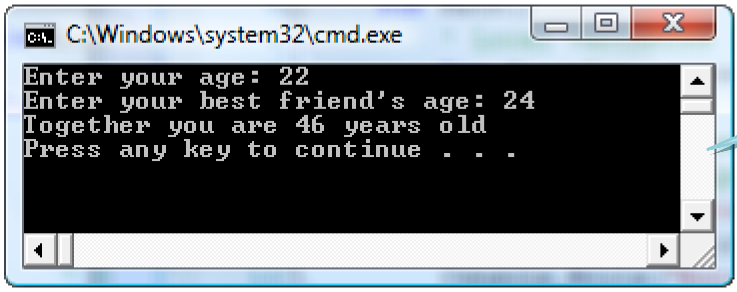
This general format is nearly identical to the pseudocode format used in Chapter 6 of your textbook. The only difference is where the data type is specified.

Program 6-2 shows a complete Java program that demonstrates this method. The program is the Visual Basic version of pseudocode Program 6-6 in your textbook.

Program 6-1

This program is the VB version of **Program 6-6** in your textbook!

```
Module Module1

    Sub Main()
        ' Local variables
        Dim firstAge, secondAge, total As Integer

        ' Get the user's age and the user's
        ' best friend's age.
        Console.Write("Enter your age: ")
        firstAge = CInt(Console.ReadLine())
        Console.Write("Enter your best friend's age: ")
        secondAge = CInt(Console.ReadLine())

        ' Get the sum of both ages.
        total = Sum(firstAge, secondAge)

        ' Display the sum.
        Console.WriteLine("Together you are " & total & " years old")
    End Sub

    Function Sum(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
        Dim result As Integer
        result = num1 + num2
        Return result
    End Function

End Module
```



```
Enter your age: 22
Enter your best friend's age: 24
Together you are 46 years old
Press any key to continue . . .
```

This is the program's output.

Returning Strings

The following code shows an example of how a function can return string. Notice that the function header specifies `String` as the return type. This function accepts two string arguments (a person's first name and last name). It combines those two strings into a string that contains the person's full name. The full name is then returned.

```
Function FullName(ByVal firstName As String, ByVal lastName As String) As String
    Dim name As String

    name = firstName & lastName
    Return name
End Function
```

The following code snippet shows how we might call the function:

```
Dim customerName As String
customerName = FullName("John", "Martin")
```

After this code executes, the value of the `custerName` variable will be "John Martin".

Returning a `Boolean` Value

Functions can also return `Boolean` values. The following function accepts an argument and returns `True` if the argument is within the range of 1 through 100, or `False` otherwise:

```
Function IsValid(ByVal number As Integer) As Boolean
    Dim status As Boolean

    If number >= 1 And number <= 100 Then
        status = True
    Else
        status = False
    End If

    Return status
End Function
```

The following code shows an `If-Then` statement that uses a call to the function:

```
Dim value As Integer = 20

If IsValid(value) Then
    Console.WriteLine("The value is within range.")
Else
    Console.WriteLine("The value is out of range.")
End If
```

When this code executes, the message “The value is within range.” will be displayed.

Math Functions

Visual Basic provides several functions that are useful for performing mathematical operations. Table 6-2 lists many of the math functions. These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. For example, one of the functions is named `Math.Sqrt`. The `Math.Sqrt` function accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```
result = Math.Sqrt(16)
```

Table 6-2 Many of the Visual Basic math functions

Function	Description
`Math.Abs(x)`	Returns the absolute value of *x*.
`Math.Acos(x)`	Returns the arc cosine of *x*, *in radians*.
`Math.Asin(x)`	Returns the arc sine of *x*, *in radians*.
`Math.Atan(x)`	Returns the arc tangent of *x*, *in radians*.
`Math.Ceiling(x)`	Returns the smallest integer that is greater than or equal to *x*.
`Math.Cos(x)`	Returns the cosine of *x* in radians.
`Math.Exp(x)`	Returns e^x
`Math.Floor(x)`	Returns the largest integer that is less than or equal to *x*.
`Math.Log(x)`	Returns the natural logarithm of *x*.
`Math.Log10(x)`	Returns the base-10 logarithm of *x*.
`Math.Sin(x)`	Returns the sine of *x* in radians.
`Math.Sqrt(x)`	Returns the square root of *x*.
`Math.Tan(x)`	Returns the tangent of *x* in radians.</TB></TBL>

The `Math.PI` and `Math.E` Constants

Visual Basic also defines two constants, `Math.PI` and `Math.E`, which are assigned mathematical values for *pi* and *e*. You can use these constants in equations that require their values. For example, the following statement, which calculates the area of a circle, uses `Math.PI`.

```
area = Math.PI * radius^2.0
```

Formatting Numbers with the `ToString` method

All numeric data types in Visual Basic have a method named `ToString` that returns the contents of a variable converted to a string. You call the method using the following general format:

```
variableName.ToString()
```

The following code segment shows an example:

```
Dim number As Integer = 123
Dim str As String
str = number.ToString()
```

The first statement declares an `Integer` variable named `number` (initialized with the value 123) and the second statement declares a `String` variable named `str`. The third statement does the following:

- The expression `number.ToString()` returns the value of the `number` variable as the string "123".
- The string "123" is assigned to the `str` varable.

By passing an appropriate argument to the `ToString` method, you can indicate how you want the number to be formatted when it is returned as a string. The following statements create a string containing the number 1234.5 in Currency format:

```
Dim number As Double
Dim resultString As String
number = 1234.5
resultString = number.ToString("c")
```

When the last statement executes, the value assigned to `resultString` is "$1,234.50". Notice that an extra zero was added at the end because currency values usually have two digits to the right of the decimal point. The argument `"c"` is called a *format string*. Table 6-3 shows various format strings that can be passed to the `ToString` method. The format strings are not case sensitive, so you can code them as uppercase or lowercase letters.

Table 6-3 Numeric format strings

Format	String Description
N or n	Number format
F or f	Fixed-point scientific format
E or e	Exponential scientific format
C or c	Currency format
P or p	Percent format

Number Format

Number format (n or N) displays numeric values with thousands separators and a decimal point. By default, two digits display to the right of the decimal point. Negative values are displayed with a leading minus (–) sign. Example:

-2,345.67

Fixed-Point Format

Fixed-point format (f or F) displays numeric values with no thousands separator and a decimal point. By default, two digits display to the right of the decimal point. Negative values are displayed with a leading minus (–) sign. Example:

-2345.67

Exponential Format

Exponential format (e or E) displays numeric values in scientific notation. The number is normalized with a single digit to the left of the decimal point. The exponent is marked by the letter e, and the exponent has a leading + or – sign. By default, six digits display to the right of the decimal point, and a leading minus sign is used if the number is negative. Example:

-2.345670e+003

Currency Format

Currency format (c or C) displays a leading currency symbol (such as $), digits, thousands separators, and a decimal point. By default, two digits display to the right of the decimal point. Negative values are surrounded by parentheses. Example:

($2,345.67)

Percent Format

Percent format (p or P) causes the number to be multiplied by 100 and displayed with a trailing space and % sign. By default, two digits display to the right of the decimal point. Negative values are displayed with a leading minus (–) sign. The following example uses –.2345:

-23.45 %

Specifying the Precision

Each numeric format string can optionally be followed by an integer that indicates the number of digits to display after the decimal point. For example, the format n3 displays three digits after the decimal point. Table 6-4 shows a variety of numeric formatting examples.

Table 6-4 Numeric formatting examples

Number Value	Format String	`ToString( )` Value
12.3	n3	12.300
12.348	n2	12.35
1234567.1	n	1,234,567.10
123456.0	f2	123456.00
123456.0	e3	1.235e+005
.234	p	23.40%
–1234567.8	c	($1,234,567.80)

Rounding

Rounding can occur when the number of digits you have specified after the decimal point in the format string is smaller than the precision of the numeric value. Suppose, for example, that the value 1.235 were displayed with a format string of n2. Then the displayed value would be 1.24. If the next digit after the last displayed digit is 5 or higher, the last displayed digit is rounded *away from zero*. Table 6-5 shows examples of rounding using a format string of n2.

Table 6-5 Rounding examples using `n2` as the format string

Number	Value Formatted As
1.234	1.23
1.235	1.24
1.238	1.24
−1.234	−1.23
−1.235	−1.24
−1.238	−1.24

String Methods

Getting a String's length

In Visual Basic, strings have a Length property that gives the length of the string. The following code snippet shows an example of how you use it:

```
' Declare and initialize a string variable.
Dim name As String = "Charlemagne"

' Assign the length of the string to the strlen variable.
Dim strlen As Integer = name.Length
```

This code declares a `String` variable named `name`, and initializes it with the string `"Charlemagne"`. Then, it declares an `Integer` variable named `strlen`. The `strlen` variable is initialized with the value of the `name.Length` property, which is 11.

Appending a String to Another String

Appending a string to another string is called concatenation. In Visual Basic you can perform string concatenation in two ways: using the & operator, and using the `String.Concat` function. Here is an example of how the & operator works:

```
Dim lastName As String = "Conway"
Dim salutation As String = "Mr. "
Dim properName As String
properName = salutation & lastName
```

After this code executes the `properName` variable will contain the string `"Mr. Conway"`. Here is an example of how you perform the same operation using the `Concat` method:

```
Dim lastName As String = "Conway"
Dim salutation As String = "Mr. "
Dim properName As String
properName = String.Concat(salutation, lastName)
```

The last statement in this code snippet calls the `String.Concat` function, passing `salutation` and `lastName` as arguments. The function will return a copy of the string in the `salutation` variable, with the contents of the `lastName` variable concatenated to it. After this code executes the `properName` variable will contain the string `"Mr. Conway"`.

The `ToUpper` and `ToLower` Methods

The `ToUpper` method returns a copy of a string with all of its letters converted to uppercase. Here is an example:

```
Dim littleName As String = "herman"
Dim bigName As String = littleName.ToUpper()
```

The `ToLower` method returns a copy of a string with all of its letters converted to lowercase. Here is an example:

```
Dim bigName As String = "HERMAN"
Dim littleName As String = bigName.ToLower()
```

The `Substring` Method

The `Substring` method returns part of another string. (A string that is part of another string is commonly referred to as a "substring.") The method takes two arguments:

- The first argument is the substring's starting position (position numbers start at 0).
- The second argument is the substring's length.

Here is an example of how the method is used:

```
Dim fullName As String = "Cynthia Susan Lee"
Dim middleName As String = fullName.Substring(8, 5)
Console.WriteLine("The full name is " & fullName)
Console.WriteLine("The middle name is " & middleName)
```

The code will produce the following output:

```
The full name is Cynthia Susan Lee
The middle name is Susan
```

The `IndexOf` Method

In Visual Basic you can use the `IndexOf` method to perform a task similar to that of the `contains` function discussed in your textbook. The `IndexOf` methods searches for substrings within a string. Here is the general format:

```
string1.IndexOf(string2)
```

In the general format *`string1`* and *`string2`* are strings. If *`string2`* is found in *`string1`*, its beginning position is returned. Otherwise, –1 is returned. The following code shows an example. It determines whether the word "and" appears in the string "four score and seven years ago".

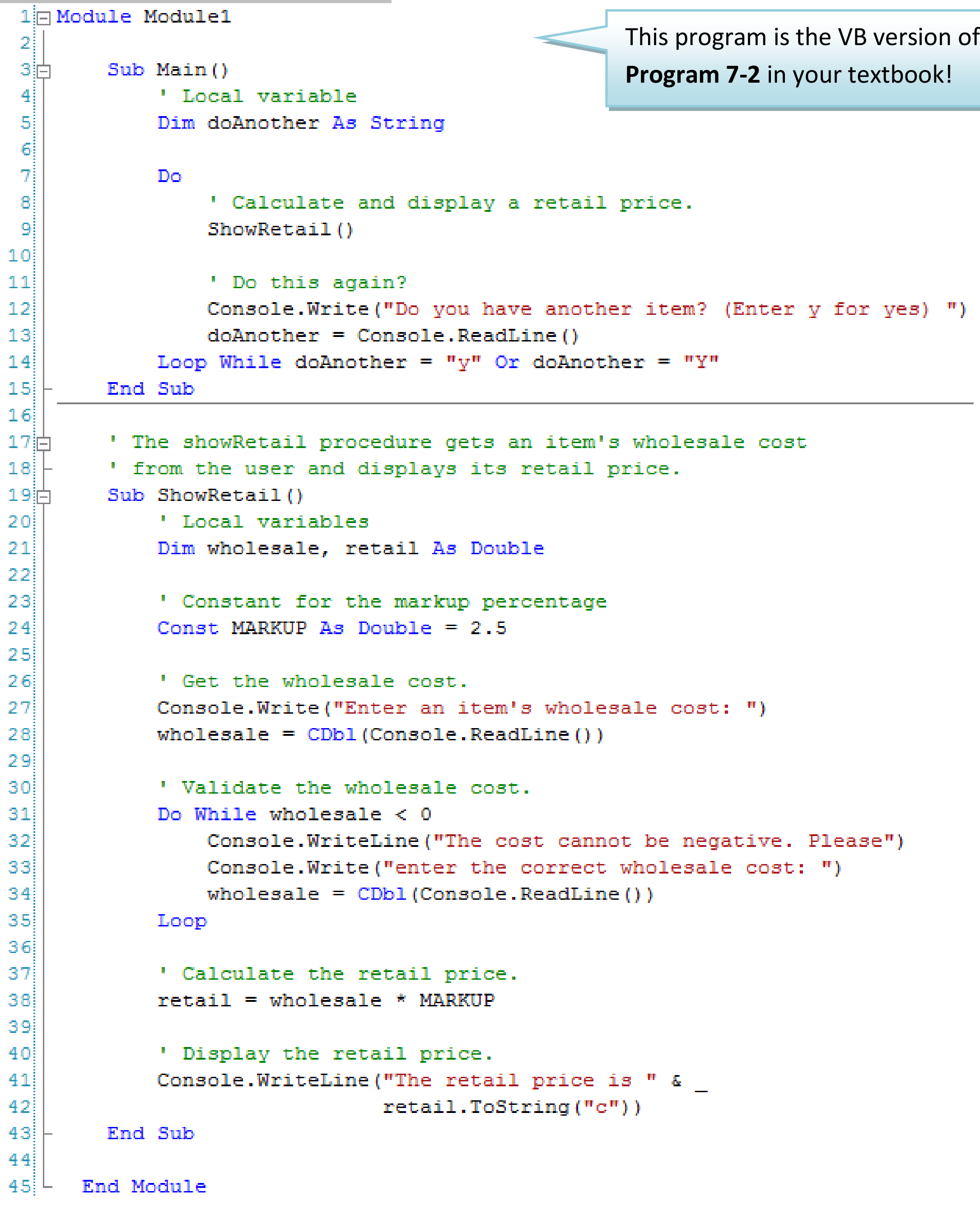
```
Dim string1 As String = "four score and seven years ago"
Dim string2 As String = "seven"
If string1.IndexOf(string2) <> -1 Then
    Console.WriteLine(string2 & " appears in the string.")
Else
     Console.WriteLine(string2 & " does not appear in the string.")
End If
```

Chapter 7 Input Validation

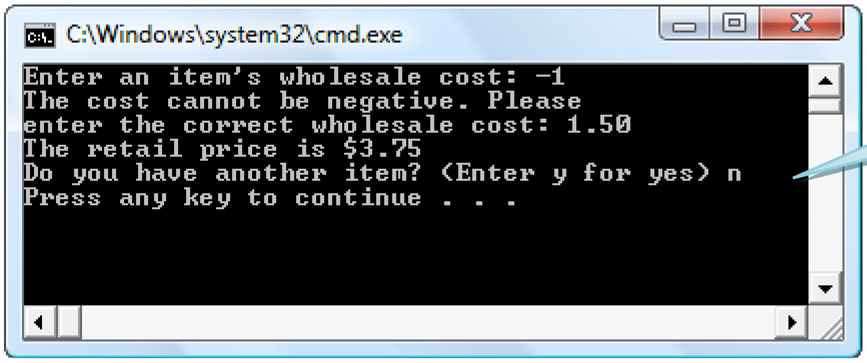
Chapter 7 in your textbook discusses the process of input validation in detail. There are no new language features introduced in the chapter, so here we will simply show you a Visual Basic version of the pseudocode Program 7-2. This program uses an input validation loop in lines 42 through 47 to validate that the value entered by the user is not negative.

Program 7-1

This program is the VB version of **Program 7-2** in your textbook!



```
Module Module1

    Sub Main()
        ' Local variable
        Dim doAnother As String

        Do
            ' Calculate and display a retail price.
            ShowRetail()

            ' Do this again?
            Console.Write("Do you have another item? (Enter y for yes) ")
            doAnother = Console.ReadLine()
        Loop While doAnother = "y" Or doAnother = "Y"
    End Sub

    ' The showRetail procedure gets an item's wholesale cost
    ' from the user and displays its retail price.
    Sub ShowRetail()
        ' Local variables
        Dim wholesale, retail As Double

        ' Constant for the markup percentage
        Const MARKUP As Double = 2.5

        ' Get the wholesale cost.
        Console.Write("Enter an item's wholesale cost: ")
        wholesale = CDbl(Console.ReadLine())

        ' Validate the wholesale cost.
        Do While wholesale < 0
            Console.WriteLine("The cost cannot be negative. Please")
            Console.Write("enter the correct wholesale cost: ")
            wholesale = CDbl(Console.ReadLine())
        Loop

        ' Calculate the retail price.
        retail = wholesale * MARKUP

        ' Display the retail price.
        Console.WriteLine("The retail price is " & _
                          retail.ToString("c"))
    End Sub

  End Module
```



```
Enter an item's wholesale cost: -1
The cost cannot be negative. Please
enter the correct wholesale cost: 1.50
The retail price is $3.75
Do you have another item? (Enter y for yes) n
Press any key to continue . . .
```

This is the program's output.

Chapter 8 Arrays

Here is an example of an array declaration in Visual Basic:

```
Dim numbers(6) As Integer
```

This statement declares `numbers` as an `Integer` array. The number inside the parentheses is the subscript of the last element in the array. The subscript of the first element is 0, so this array has a total of seven elements. Note that the number inside the parentheses is not a size declarator, as in the pseudocode array declarations shown in your textbook.

Note: With Visual Basic arrays, parentheses are used instead of the square brackets that are shown in your textbook's pseudocode.

Here is an example where we have used a named constant for the upper subscript:

```
Const MAX_SUB As Integer = 6
Dim numbers(MAX_SUB) As Integer
```

Here is another example:

```
Const MAX_SUB As Integer = 199
Dim temperatures(MAX_SUB) As Double
```

This code snippet declares `temperatures` as an array of 200 `Double`s. The subscript of the last element is 199. Here is one more:

```
Const MAX_SUB As Integer = 9
Dim names(MAX_SUB) As String
```

This declares `names` as an array of 10 `String`s.

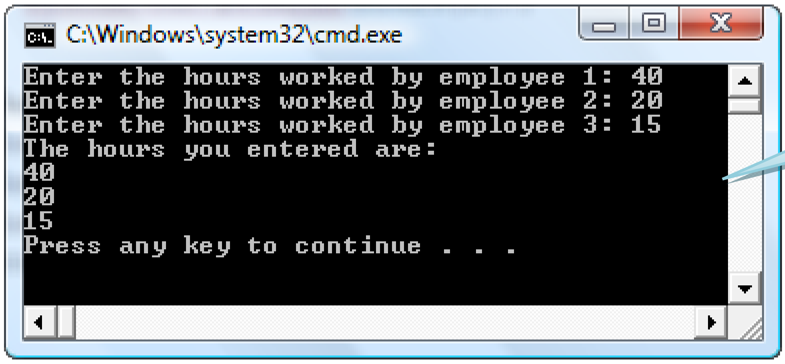
Array Elements and Subscripts

You access each element of an array with a subscript. As discussed in your textbook, the first element's subscript is 0, the second element's subscript is 1, and so forth. Program 8-1 shows an example of an array being used to hold values entered by the user. This is the Visual Basic version of pseudocode Program 8-1 in your textbook.

Program 8-1

This program is the VB version of **Program 8-1** in your textbook!

```
Module Module1

    Sub Main()
        ' Create a constant for the last subscript.
        Const MAX_SUB As Integer = 2

        ' Declare an array to hold the number of hours
        ' worked by each employee.
        Dim hours(MAX_SUB) As Integer

        ' Get the hours worked by employee 1.
        Console.Write("Enter the hours worked by employee 1: ")
        hours(0) = CInt(Console.ReadLine())

        ' Get the hours worked by employee 2.
        Console.Write("Enter the hours worked by employee 2: ")
        hours(1) = CInt(Console.ReadLine())

        ' Get the hours worked by employee 3.
        Console.Write("Enter the hours worked by employee 3: ")
        hours(2) = CInt(Console.ReadLine())

        ' Display the values entered.
        Console.WriteLine("The hours you entered are:")
        Console.WriteLine(hours(0))
        Console.WriteLine(hours(1))
        Console.WriteLine(hours(2))
    End Sub

End Module
```



```
Enter the hours worked by employee 1: 40
Enter the hours worked by employee 2: 20
Enter the hours worked by employee 3: 15
The hours you entered are:
40
20
15
Press any key to continue . . .
```

This is the program's output.

The Length Property

Each array in Visual Basic has a property named Length. The value of the Length property is the number of elements in the array. For example, consider the array created by the following statement:

```
Dim temperatures(24) As Double
```

Because the temperatures array has 25 elements, the following statement would assign 25 to the variable size:

```
size = temperatures.Length
```

The Length property can be useful when processing the entire contents of an array. For example, the following loop steps through an array and displays the contents of each element. The array's Length property is used in the test expression as the upper limit for the loop control variable:

```
Dim index As Integer
For index = 0 To temperatures.Length - 1
    Console.WriteLine(temperatures(index))
Next
```

Be careful not to cause an off-by-one error when using the Length property as the upper limit of a subscript. The Length property contains the number of elements in an array. The largest subscript in an array is Length – 1.

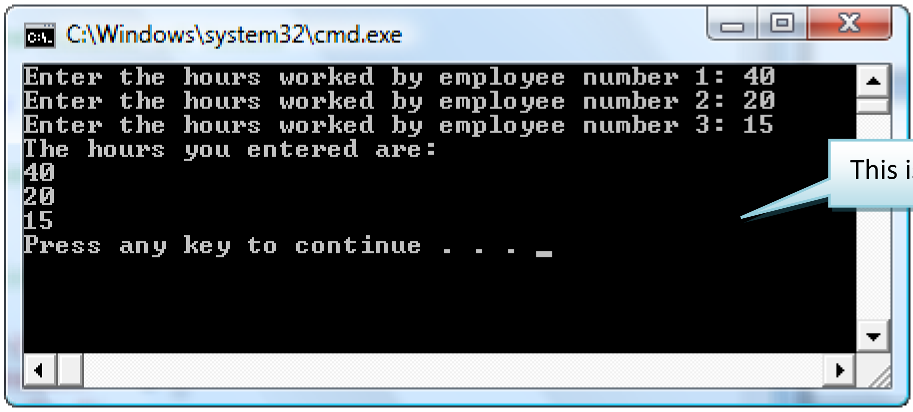
Using a Loop to Process an Array

It is usually much more efficient to use a loop to access an array's elements, rather than writing separate statements to access each element. Program 8-2 demonstrates how to use a loop to step through an array's elements. This is the Visual Basic version of pseudocode Program 8-3 in your textbook.

Program 8-2

This program is the VB version of **Program 8-3** in your textbook!

```
Module Module1

    Sub Main()
        ' Create a constant for the last subscript.
        Const MAX_SUB As Integer = 2

        ' Declare an array to hold the number of hours
        ' worked by each employee.
        Dim hours(MAX_SUB) As Integer

        ' Declare a variable to use in the loops.
        Dim index As Integer

        ' Get the hours for each employee.
        For index = 0 To MAX_SUB
            Console.Write("Enter the hours worked by " & _
                          "employee number " & _
                          index + 1 & ": ")
            hours(index) = CInt(Console.ReadLine())
        Next

        ' Display the values entered.
        Console.WriteLine("The hours you entered are:")
        For index = 0 To MAX_SUB
            Console.WriteLine(hours(index))
        Next
    End Sub

End Module
```



C:\Windows\system32\cmd.exe

```
Enter the hours worked by employee number 1: 40
Enter the hours worked by employee number 2: 20
Enter the hours worked by employee number 3: 15
The hours you entered are:
40
20
15
Press any key to continue . . . _
```

This is the program's output.

Initializing an Array

You can initialize an array with values when you declare it. Here is an example:

```
Dim days() As Integer = {31, 28, 31, 30, _
                         31, 30, 31, 31, _
                         30, 31, 30, 31}
```

This statement declares days as an array of `Integer`s, and stores initial values in the array. The series of values inside the braces and separated with commas is called an initialization list. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days(0)`, the second value, 28, is stored in `days(1)`, and so forth.) Note that you do not specify the last subscript when you use an initialization list. Visual Basic automatically creates the array and stores the values in the initialization list in it.

The Visual Basic compiler determines the size of the array by the number of items in the initialization list. Because there are 12 items in the example statement's initialization list, the array will have 12 elements.

Sequentially Searching an Array

Section 8.2 in your textbook discusses the sequential search algorithm, in which a program steps through each of an array's elements searching for a specific value. Program 8-3 shows an example of the sequential search algorithm. This is the Visual Basic version of pseudocode Program 8-6 in the textbook.

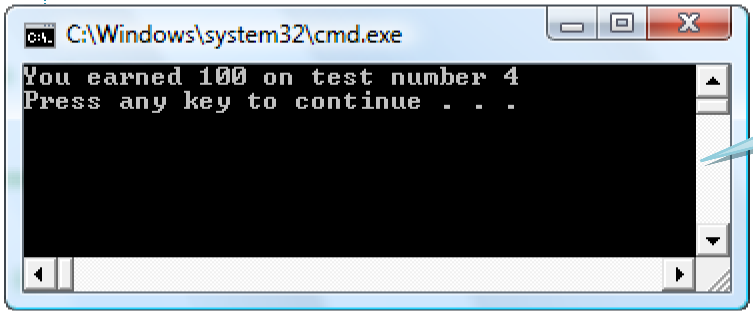
Searching a String Array

Program 8-4 demonstrates how to find a string in a string array. This is the Visual Basic version of pseudocode Program 8-7 in the textbook.

Program 8-3

This program is the VB version of **Program 8-6** in your textbook!

```
Module Module1

    Sub Main()
        Dim scores() As Integer = {87, 75, 98, 100, 82, _
                                   72, 88, 92, 60, 78}

        ' Declare a Boolean variable to act as a flag.
        Dim found As Boolean

        ' Declare a variable to use as a loop counter.
        Dim index As Integer

        ' The flag must initially be set to False.
        found = False

        ' Set the counter variable to 0.
        index = 0

        ' Step through the array searching for a score
        ' equal to 100.
        Do While found = False And index <= scores.Length - 1
            If scores(index) = 100 Then
                found = True
            Else
                index = index + 1
            End If
        Loop

        ' Display the search results.
        If found Then
            Console.WriteLine("You earned 100 on test number " & _
                              index + 1)
        Else
            Console.WriteLine("You did not earn 100 on any test.")
        End If
    End Sub

End Module
```



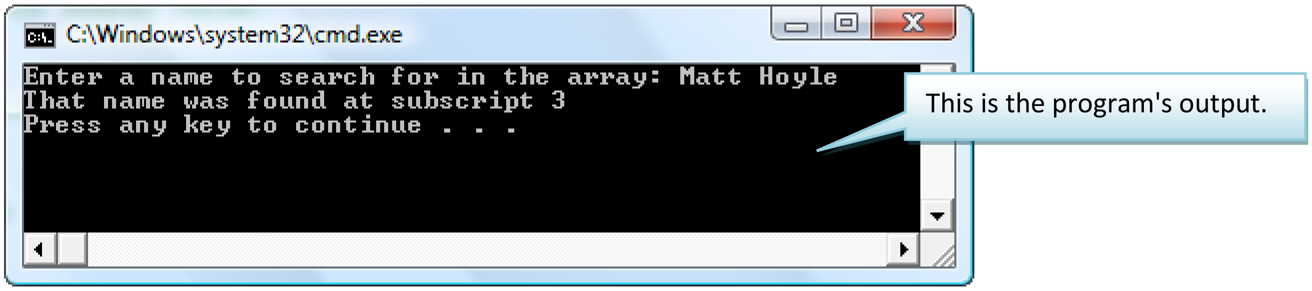
```
You earned 100 on test number 4
Press any key to continue . . .
```

This is the program's output.

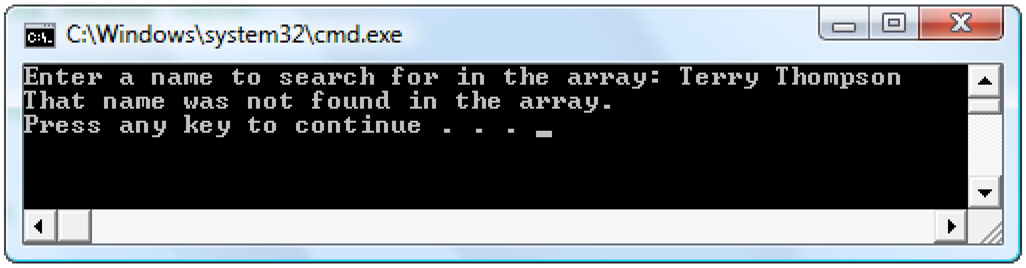
Program 8-4

This program is the VB version of **Program 8-7** in your textbook!

```
Module Module1

    Sub Main()
        Dim names() As String = {"Ava Fischer", "Chris Rich", _
                                 "Gordon Pike", "Matt Hoyle", _
                                 "Rose Harrison", "Giovanni Ricci"}

        ' Declare a variable to hold the search value.
        Dim searchValue As String

        ' Declare a Boolean variable to act as a flag.
        Dim found As Boolean

        ' Declare a variable to use as a loop counter.
        Dim index As Integer

        ' The flag must initially be set to False.
        found = False

        ' Set the counter variable to 0.
        index = 0

        ' Get the string to search for.
        Console.Write("Enter a name to search for in the array: ")
        searchValue = Console.ReadLine()

        ' Step through the array searching for
        ' the specified name.
        Do While found = False And index <= names.Length - 1
            If names(index) = searchValue Then
                found = True
            Else
                index = index + 1
            End If
        Loop

        ' Display the search results.
        If found Then
            Console.WriteLine("That name was found at subscript " & _
                              index)
        Else
            Console.WriteLine("That name was not found in the array.")
        End If
    End Sub

End Module
```



```
Enter a name to search for in the array: Matt Hoyle
That name was found at subscript 3
Press any key to continue . . .
```



```
Enter a name to search for in the array: Terry Thompson
That name was not found in the array.
Press any key to continue . . .
```

Passing an Array as an Argument to a Procedure or Function

When passing an array as an argument to a procedure or function in Visual Basic, it is not necessary to pass a separate argument indicating the array's size. This is because arrays in Visual Basic have the `Length` property that reports the array's size. The following code shows a procedure that has been written to accept an `Integer` array as an argument:

```
Sub ShowArray(ByVal array() As Integer)
    Dim index As Integer

    For index = 0 To array.Length - 1
        Console.WriteLine(array(index) & " ")
    Next
End Sub
```

Notice that the parameter variable, `array`, is declared with an empty set of parentheses. This indicates that the parameter receives an array as an argument. When we call this procedure we must pass an `Integer` array as an argument. Assuming that `numbers` is the name of an `Integer` array, here is an example of a procedure call that passes the `numbers` array as an argument to the `ShowArray` procedure:

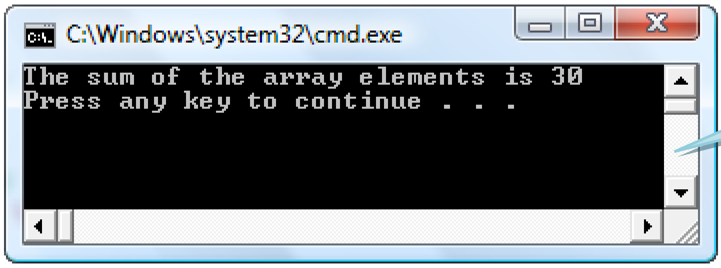
```
ShowArray(numbers)
```

Program 8-5 gives a complete demonstration of passing an array to a procedure. This is the Visual Basic version of pseudocode Program 8-13 in your textbook.

Program 8-5

This program is the VB version of **Program 8-13** in your textbook!

```vb
Module Module1

    Sub Main()
        ' An array initialized with values.
        Dim numbers() As Integer = {2, 4, 6, 8, 10}

        ' A variable to hold the sum of the elements.
        Dim sum As Integer

        ' Get the sum of the elements.
        sum = GetTotal(numbers)

        ' Display the sum of the array elements.
        Console.WriteLine("The sum of the array elements is " & sum)
    End Sub

    ' The GetTotal function accepts an Integer array as an
    ' argument. It returns the total of the array elements
    Function GetTotal(ByVal array() As Integer) As Integer
        ' Loop counter
        Dim index As Integer

        ' Accumulator, initialized to 0
        Dim total As Integer = 0

        ' Calculate the total of the array elements.
        For index = 0 To array.Length - 1
            total = total + array(index)
        Next

        ' Return the total.
        Return total
    End Function

End Module
```



```
The sum of the array elements is 30
Press any key to continue . . .
```

This is the program's output.

Two-Dimensional Arrays

Here is an example declaration of a two-dimensional array with three rows and four columns:

```
Dim scores(2, 3) As Double
```

The numbers 2 and 3 inside the parentheses are the upper subscripts. The number 2 is the subscript of the last row, and the number 3 is the subscript of the last column. So, the row subscripts for this array are 0, 1, and 2. The column subscripts are 0, 1, 2, and 3.

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array, the elements in row 0 are referenced as follows:

```
scores(0)(0)   ←————————   The first element in row 0
scores(0)(1)   ←————————   The second element in row 0
scores(0)(2)   ←————————   The third element in row 0
scores(0)(3)   ←————————   The fourth element in row 0
```

The elements in row 1 are as follows:

```
scores(1)(0)   ←————————   The first element in row 1
scores(1)(1)   ←————————   The second element in row 1
scores(1)(2)   ←————————   The third element in row 1
scores(1)(3)   ←————————   The fourth element in row 1
```

And the elements in row 2 are as follows:

```
scores(2)(0)   ←————————   The first element in row 2
scores(2)(1)   ←————————   The second element in row 2
scores(2)(2)   ←————————   The third element in row 2
scores(2)(3)   ←————————   The fourth element in row 2
```

To access one of the elements in a two-dimensional array, you must use both subscripts. For example, the following statement stores the number 95 in `scores(2, 1)`:

```
scores(2, 1) = 95
```

Programs that process two-dimensional arrays can do so with nested loops. For example, the following code prompts the user to enter a score, once for each element in the array:

```
Const MAX_ROW As Integer = 3
Const MAX_COL As Integer = 4
Dim row, col As Integer
Dim scores(MAX_ROW, MAX_COL) As Double

For row = 0 To MAX_ROW
    For col = 0 To MAX_COL
        Console.WriteLine("Enter a score: ")
        scores(row, col) = CDbl(Console.ReadLine())
    Next
Next
```

And the following code displays all the elements in the `scores` array:

```
For row = 0 To MAX_ROW
    For col = 0 To MAX_COL
        Console.WriteLine(scores(row, col))
    Next
Next
```

Program 8-6 shows a complete example. It declares an array with three rows and four columns, prompts the user for values to store in each element, and then displays the values in each element. This is the Visual Basic example of pseudocode Program 8-16 in your textbook.

Arrays with Three or More Dimensions

Visual Basic allows you to create arrays with up to 32 dimensions. Here is an example of a three-dimensional array declaration:

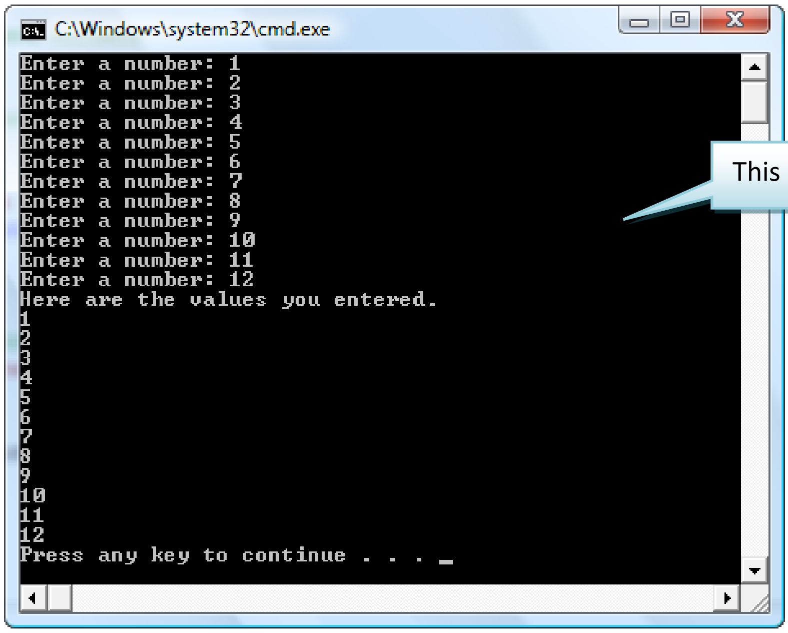
```
Dim seats(2, 4, 7) As Double
```

The last subscripts in the array's dimensions are 2, 4, and 7. This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Program 8-6

This program is the VB version of **Program 8-16** in your textbook!

```
Module Module1

    Sub Main()
        ' Create a 2D array
        Const MAX_ROW As Integer = 2
        Const MAX_COL As Integer = 3
        Dim values(MAX_ROW, MAX_COL) As Integer

        ' Counter variables for rows and columns
        Dim row, col As Integer

        ' Get values to store in the array.
        For row = 0 To MAX_ROW
            For col = 0 To MAX_COL
                Console.Write("Enter a number: ")
                values(row, col) = CInt(Console.ReadLine())
            Next
        Next

        ' Display the values in the array.
        Console.WriteLine("Here are the values you entered.")
        For row = 0 To MAX_ROW
            For col = 0 To MAX_COL
                Console.WriteLine(values(row, col))
            Next
        Next
    End Sub

End Module
```



C:\Windows\system32\cmd.exe

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: 5
Enter a number: 6
Enter a number: 7
Enter a number: 8
Enter a number: 9
Enter a number: 10
Enter a number: 11
Enter a number: 12
Here are the values you entered.
1
2
3
4
5
6
7
8
9
10
11
12
Press any key to continue . . . _
```

This is the program's output.

Chapter 9 Sorting and Searching Arrays

Chapter 9 discusses the following sorting algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort

The Binary Search algorithm is also discussed. The textbook chapter examines these algorithms in detail, and no new language features are introduced. For these reasons we will simply present the Visual Basic code for the algorithms in this chapter. For more in-depth coverage of the logic involved, consult the textbook.

Bubble Sort

Program 9-1 is only a partial program. It shows the Visual Basic version of pseudocode Program 9-1, which is the Bubble Sort algorithm. The code listing also shows the Visual Basic version of the `Swap` procedure.

Selection Sort

Program 9-2 is a complete program. It shows the Visual Basic version of the `selectionSort` pseudocode module that is shown in Program 9-5 in your textbook.

Insertion Sort

Program 9-3 is a complete program. It shows the Visual Basic version of the `insertionSort` pseudocode module that is shown in Program 9-6 in your textbook.

Binary Search

Program 9-4 is a partial program. It shows the Visual Basic version of the `binarySearch` pseudocode module that is shown in Program 9-7 in your textbook.

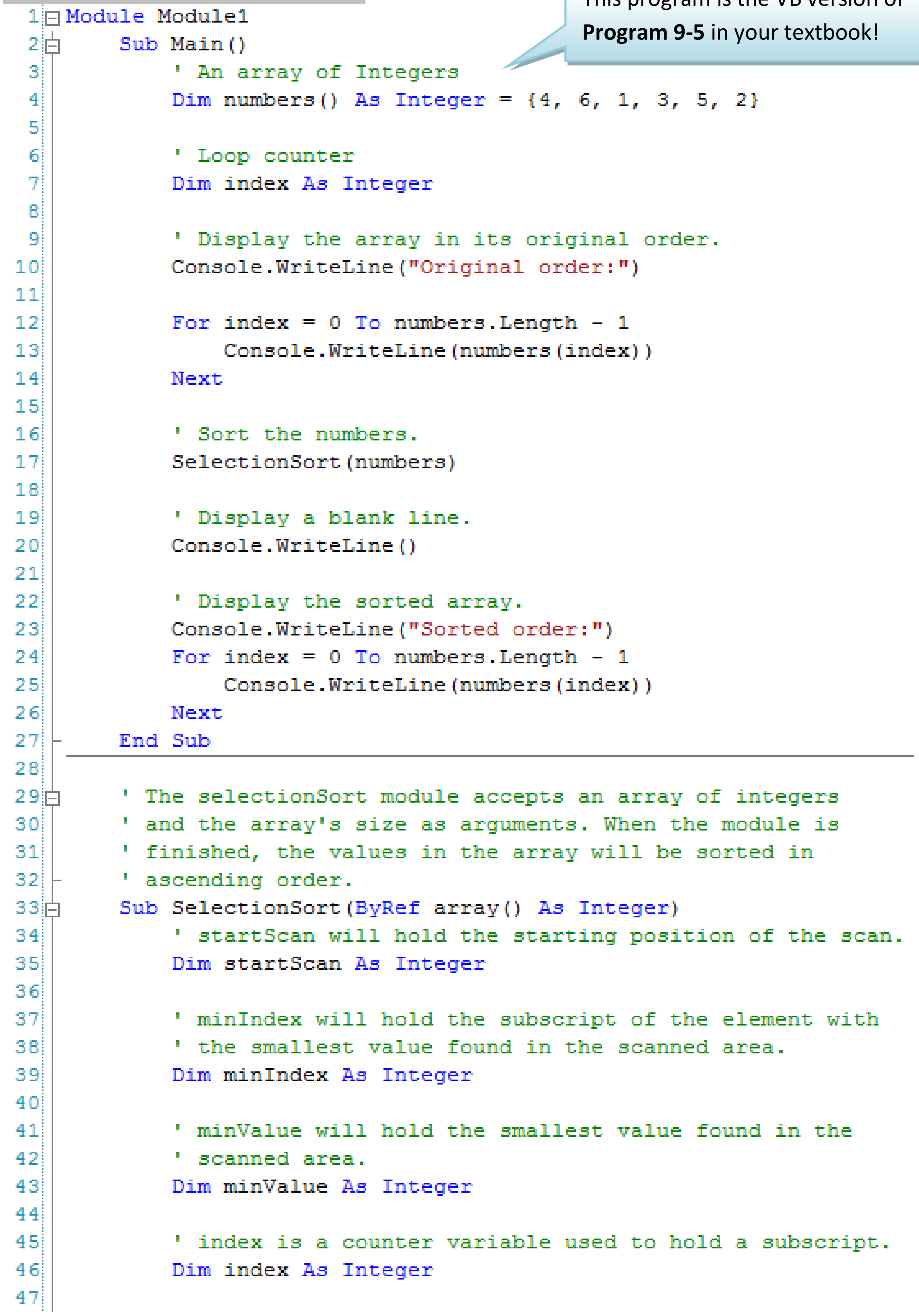
Program 9-1

This program is the VB version of **Program 9-1** in your textbook!

```
    Sub BubbleSort(ByRef array() As Integer)
        ' The maxElement variable will contain the subscript
        ' of the last element in the array to compare.
        Dim maxElement As Integer

        ' The index variable will be used as a counter
        ' in the inner loop.
        Dim index As Integer

        ' The outer loop positions maxElement at the last
        ' element to compare during each pass through the
        ' array. Initially maxElement is the index of the
        ' last element in the array. During each iteration,
        ' it is decreased by one.
        For maxElement = array.Length - 1 To 0 Step -1
            ' The inner loop steps through the array, comparing
            ' each element with its neighbor. All of the
            ' elements from index 0 through maxElement are
            ' involved in the comparison. If two elements are
            ' out of order, they are swapped.
            For index = 0 To maxElement - 1
                ' Compare an element with its neighbor and swap
                ' if necessary.
                If array(index) > array(index + 1) Then
                    Swap(array(index), array(index + 1))
                End If
            Next
        Next
    End Sub

    Sub Swap(ByRef a As Integer, ByRef b As Integer)
        ' Local variable for temporary storage.
        Dim temp As Integer

        ' Swap the values in a and b.
        temp = a
        a = b
        b = temp
    End Sub
```

Program 9-2

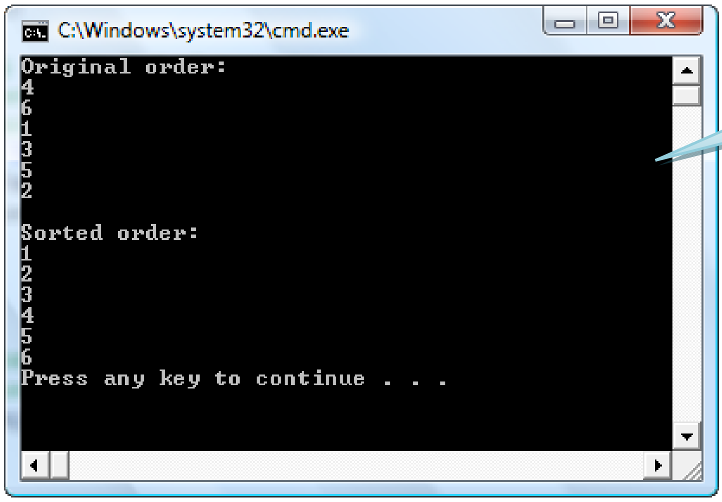
This program is the VB version of **Program 9-5** in your textbook!



```vb
Module Module1
    Sub Main()
        ' An array of Integers
        Dim numbers() As Integer = {4, 6, 1, 3, 5, 2}

        ' Loop counter
        Dim index As Integer

        ' Display the array in its original order.
        Console.WriteLine("Original order:")

        For index = 0 To numbers.Length - 1
            Console.WriteLine(numbers(index))
        Next

        ' Sort the numbers.
        SelectionSort(numbers)

        ' Display a blank line.
        Console.WriteLine()

        ' Display the sorted array.
        Console.WriteLine("Sorted order:")
        For index = 0 To numbers.Length - 1
            Console.WriteLine(numbers(index))
        Next
    End Sub

    ' The selectionSort module accepts an array of integers
    ' and the array's size as arguments. When the module is
    ' finished, the values in the array will be sorted in
    ' ascending order.
    Sub SelectionSort(ByRef array() As Integer)
        ' startScan will hold the starting position of the scan.
        Dim startScan As Integer

        ' minIndex will hold the subscript of the element with
        ' the smallest value found in the scanned area.
        Dim minIndex As Integer

        ' minValue will hold the smallest value found in the
        ' scanned area.
        Dim minValue As Integer

        ' index is a counter variable used to hold a subscript.
        Dim index As Integer

```

Program continues on next page….

```
        ' The outer loop iterates once for each element in the
        ' array, except the last element. The startScan variable
        ' marks the position where the scan should begin.
        For startScan = 0 To array.Length - 2
            ' Assume the first element in the scannable area
            ' is the smallest value.
            minIndex = startScan
            minValue = array(startScan)

            ' Scan the array, starting at the 2nd element in
            ' the scannable area. We are looking for the smallest
            ' value in the scannable area.
            For index = startScan + 1 To array.Length - 1
                If array(index) < minValue Then
                    minValue = array(index)
                    minIndex = index
                End If
            Next

            ' Swap the element with the smallest value
            ' with the first element in the scannable area.
            Swap(array(minIndex), array(startScan))
        Next
    End Sub

    Sub Swap(ByRef a As Integer, ByRef b As Integer)
        ' Local variable for temporary storage.
        Dim temp As Integer

        ' Swap the values in a and b.
        temp = a
        a = b
        b = temp
    End Sub
End Module
```


C:\Windows\system32\cmd.exe

```
Original order:
4
6
1
3
5
2

Sorted order:
1
2
3
4
5
6
Press any key to continue . . .
```

This is the program's output.

Program 9-3

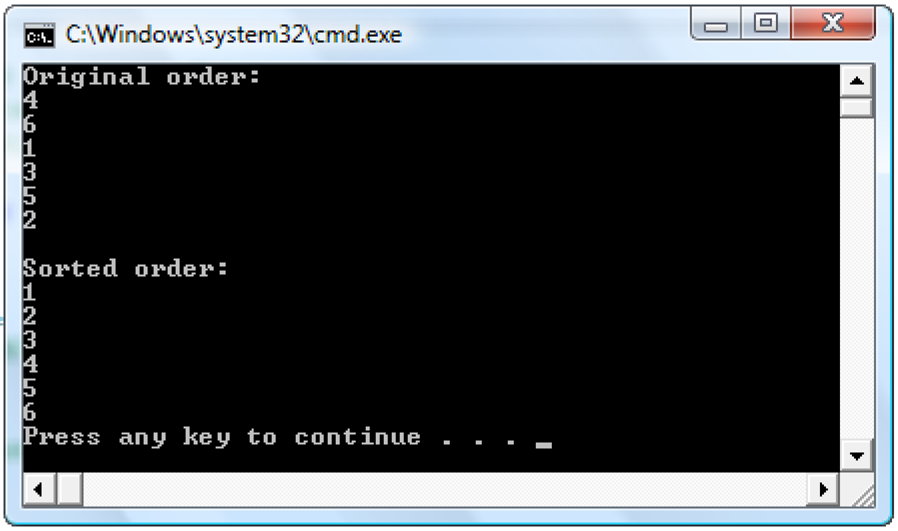
This program is the VB version of **Program 9-6** in your textbook!

```
Module Module1
    Sub Main()
        ' An array of Integers
        Dim numbers() As Integer = {4, 6, 1, 3, 5, 2}

        ' Loop counter
        Dim index As Integer

        ' Display the array in its original order.
        Console.WriteLine("Original order:")

        For index = 0 To numbers.Length - 1
            Console.WriteLine(numbers(index))
        Next

        ' Sort the numbers.
        InsertionSort(numbers)

        ' Display a blank line.
        Console.WriteLine()

        ' Display the sorted array.
        Console.WriteLine("Sorted order:")
        For index = 0 To numbers.Length - 1
            Console.WriteLine(numbers(index))
        Next
    End Sub

    ' The insertionSort module accepts an array of integers
    ' and the array's size as arguments. When the procedure is
    ' finished, the values in the array will be sorted in
    ' ascending order.
    Sub InsertionSort(ByRef array() As Integer)
        ' Loop counter
        Dim index As Integer

        ' Variable used to scan through the array.
        Dim scan As Integer

        ' Variable to hold the first unsorted value.
        Dim unsortedValue As Integer

```

Program continues on next page….

```
    ' The outer loop steps the index variable through
    ' each subscript in the array, starting at 1. This
    ' is because element 0 is considered already sorted.
    For index = 1 To array.Length - 1
      ' The first element outside the sorted subset is
      ' array(index). Store the value of this element
      ' in unsortedValue.
      unsortedValue = array(index)

      ' Start scan at the subscript of the first element
      ' outside the sorted subset.
      scan = index

      ' Move the first element outside the sorted subset
      ' into its proper position within the sorted subset.
      ' NOTE: This loop relies on short-circuit evaluation
      ' to operate properly. In VB we have to use the AndAlso
      ' operator instead of the And operator!
      Do While scan > 0 AndAlso array(scan - 1) > array(scan)
        Swap(array(scan - 1), array(scan))
        scan = scan - 1
      Loop

      ' Insert the unsorted value in its proper position
      ' within the sorted subset.
      array(scan) = unsortedValue
    Next
  End Sub

  Sub Swap(ByRef a As Integer, ByRef b As Integer)
    ' Local variable for temporary storage.
    Dim temp As Integer

    ' Swap the values in a and b.
    temp = a
    a = b
    b = temp
  End Sub
End Module
```

This is the program's output.


C:\Windows\system32\cmd.exe

```
Original order:
4
6
1
3
5
2

Sorted order:
1
2
3
4
5
6
Press any key to continue . . . _
```

Program 9-4

This program is the VB version of **Program 9-7** in your textbook!

```
    Function BinarySearch(ByVal array() As Integer, _
                          ByVal value As Integer) As Integer
        ' Variable to hold the subscript of the first element.
        Dim first As Integer = 0

        ' Variable to hold the subscript of the last element.
        Dim last As Integer = array.Length - 1

        ' Position of the search value
        Dim position As Integer = -1

        ' Flag
        Dim found As Boolean = False

        ' Variable to hold the subscript of the midpoint.
        Dim middle As Integer

        Do While (Not found) And (first <= last)
            ' Calculate the midpoint.
            middle = CInt((first + last) / 2)

            ' See if the value is found at the midpoint...
            If array(middle) = value Then
                found = True
                position = middle

                ' Else, if the value is in the lower half...
            ElseIf array(middle) > value Then
                last = middle - 1

                ' Else, if the value is in the upper half...
            Else
                first = middle + 1
            End If
        Loop

        ' Return the position of the item, or -1
        ' if the item was not found.
        Return position
    End Function
```

Chapter 10 Files

The `Imports` Statement

To work with files in a Visual Basic program you first write the following statement at the top of your program:

```
Imports System.IO
```

This statement must be written *before* the beginning of the `Module` definition.

Opening a File to Write Data to It

In the procedure or function where you wish to open a file and write data to it you declare a `StreamWriter` variable. Here is an example:

```
Dim outputFile As StreamWriter
```

This statement declares a `SreamWriter` variable named `outputFile`. After this statement has executed, we will be able to use the `outputFile` variable to open a file and write data to it. What we do next depends on whether we want to create a new file, or open an existing file and append data to it.

Creating a New File

If we want to create a new file, we call the `File.CreateText` function. Here is an example:

```
Dim outputFile As StreamWriter
outputFile = File.CreateText("StudentData.txt")
```

The first statement declares a `StreamWriter` variable named `outputFile`. The second statement calls the `File.CreateText` function, passing the name of the file that we want to create as an argument. The value that is returned from the function is assigned to the `outputFile` variable. After this code executes, an empty file named *StudentData.txt* will be created on the disk, and the `outputFile` variable will be associated with that file. The program will be able to use the `outputFile` variable to write data to the file. (If the file that you are opening with `File.CreateText` already exists, its contents will be erased.)

Opening an Existing File to Append Data to It

If a file already exists, you may want to add more data to the end of the file. This is called *appending* to the file. First, you declare a `StreamWriter` variable:

```
Dim outputFile As StreamWriter
```

Then you call the `File.AppendText` method, passing it the name of an existing file. For example:

```
outputFile = File.AppendText("StudentData.txt")
```

This statement opens the existing file *StudentData.txt*, preserving its current contents. Any data written to the file will be written to the end of the file's existing contents. (If the file does not exist, it will be created.)

Closing a File

When the program is finished working with a file, it must close the file. Assuming that `outputFile` is the name of a `StreamWriter` variable, here is an example of how to call the `Close` method to close the file:

```
outputFile.Close()
```

Once a file is closed, the connection between it and the `StreamWriter` variable is removed. In order to perform further operations on the file, it must be opened again.

Writing Data to a File

Once you have created a `StreamWriter` variable and opened a file, you can use the `WriteLine` method to write data to the file. The `StreamWriter`'s `WriteLine` method works like the `Console.WriteLine` method. The output goes to a file instead of the screen. The following code snippet demonstrates:

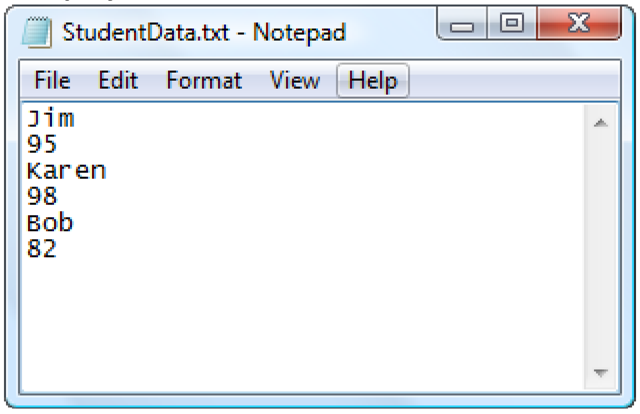
```
' Declare a StreamWriter variable
Dim outputFile As StreamWriter

' Open the file.
outputFile = File.CreateText("StudentData.txt")

' Write data to the file.
outputFile.WriteLine("Jim")
outputFile.WriteLine(95)
outputFile.WriteLine("Karen")
outputFile.WriteLine(98)
outputFile.WriteLine("Bob")
outputFile.WriteLine(82)
```

```
' Close the file.
outputFile.Close()
```

You can visualize the data being written to the file in the following manner:

Jim<*newline*>95<*newline*>Karen<*newline*>98<*newline*>Bob<*newline*>82<*newline*>
The newline characters are represented here as <*newline*>. You do not actually see the newline characters, but when the file is opened in a text editor such as Notepad, its contents appear as shown in Figure 10-1. As you can see from the figure, each newline character causes the data that follows it to be displayed on a new line.

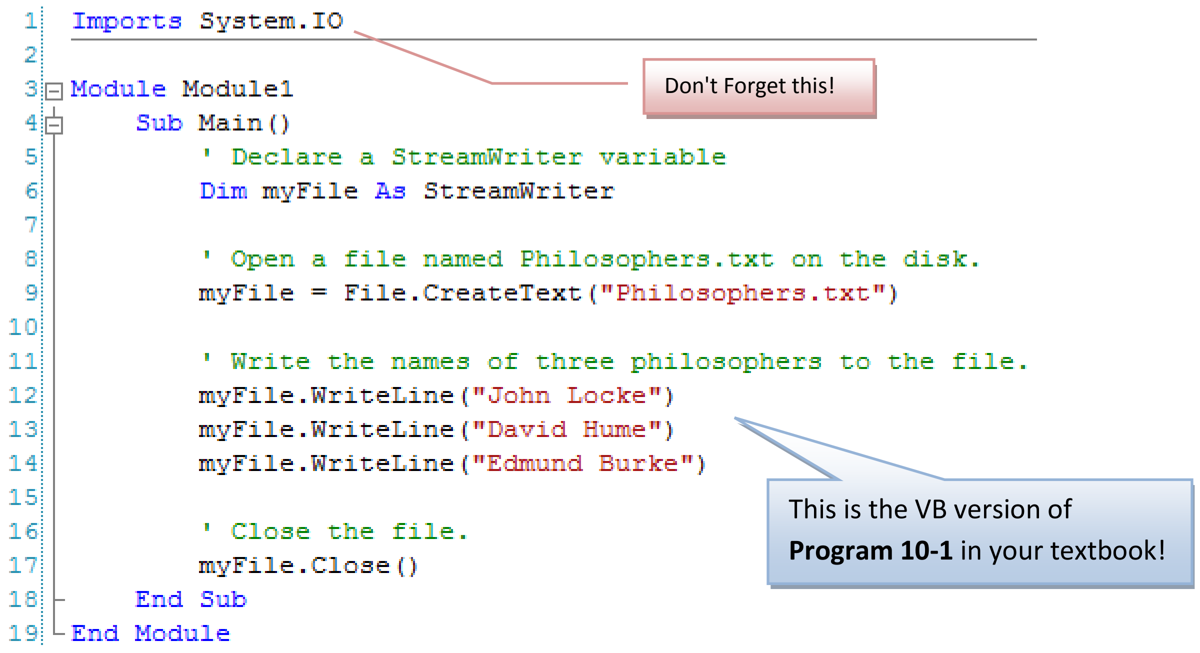
Figure 10-1 File contents displayed in NotePad



```
Jim
95
Karen
98
Bob
82
```

Program 10-1 demonstrates how to open a file for output, write some data to the file, and close the file. This is the Visual Basic version of pseudocode Program 10-1 in your textbook. When this program executes, line 9 creates a file named philosophers.txt on the disk, and lines 12 through 14 write the strings "John Locke", "David Hume", and "Edmund Burke" to the file. Line 17 closes the file.

Program 10-1



```
Imports System.IO

Module Module1
    Sub Main()
        ' Declare a StreamWriter variable
        Dim myFile As StreamWriter

        ' Open a file named Philosophers.txt on the disk.
        myFile = File.CreateText("Philosophers.txt")

        ' Write the names of three philosophers to the file.
        myFile.WriteLine("John Locke")
        myFile.WriteLine("David Hume")
        myFile.WriteLine("Edmund Burke")

        ' Close the file.
        myFile.Close()
    End Sub
End Module
```

Don't Forget this!

This is the VB version of **Program 10-1** in your textbook!

Opening a File to Read Data from It

In the procedure or function where you wish to open an existing file to read data from it, you first declare a `StreamReader` variable. Here is an example:

```
Dim inputFile As StreamReader
```

Next you call the `File.OpenText` function to open the file. Here is an example:

```
Dim inputFile As StreamReader
inputFile = File.OpenText("StudentData.txt")
```

The first statement declares a `StreamReader` variable named `inputFile`. The second statement calls the `File.OpenText` function, passing the name of the file that we want to open as an argument. The value that is returned from the function is assigned to the `inputFile` variable. After this code executes, the specified file will be opened and the `inputFile` variable will be associated with the file. The program will be able to use the `inputFile` variable to read data from the file. (If the file that you are opening with `File.CreateText` does not exist, an error will occur.)

Closing a File

When the program is finished working with a file, it must close the file. Assuming that `inputFile` is the name of a `StreamReader` variable, here is an example of how to call the `Close` method to close the file:

```
inputFile.Close()
```

Once a file is closed, the connection between it and the `StreamReader` variable is removed. In order to perform further operations on the file, it must be opened again.

Reading Data from a File

Once you have created a `StreamReader` variable and opened a file, you can use the `ReadLine` method to read a line of data from the file. (The line of data that is read from the file is returned as a string.) The `StreamReader`'s `ReadLine` method works like the `Console.ReadLine` method, except that the input comes from a file instead of the keyboard. The following code snippet demonstrates:

```
' Declare variables to hold data from the file.
Dim name As String
Dim score As Integer

' Declare a StreamReader variable.
Dim inputFile As StreamReader

' Open the file.
inputFile = File.OpenText("StudentData.txt")

' Read a name from the file.
name = inputFile.ReadLine()

' Read a test score from the file.
score = CInt(inputFile.ReadLine())

' Close the file.
outputFile.Close()

' Display the data
Console.WriteLine("Name: " & name)
Console.WriteLine("Score: " & score)
```

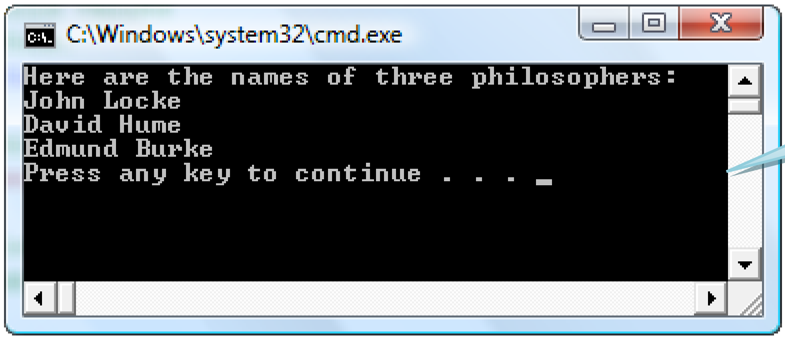
Program 10-2 shows an example that reads strings from a file. This program opens the `philosophers.txt` file that was created by Program 10-1. This is the Visual Basic version of pseudocode Program 10-2 in your textbook.

Program 10-2

This program is the VB version of **Program 10-2** in your textbook!

```
Imports System.IO

Module Module1
    Sub Main()
        ' Declare a StreamReader variable
        Dim myFile As StreamReader

        ' Declare three variables to hold values
        ' that will be read from the file.
        Dim name1, name2, name3 As String

        ' Open a file named Philosophers.txt on the disk.
        myFile = File.OpenText("Philosophers.txt")

        ' Read the names of three philosophers
        ' from the file into the variables.
        name1 = myFile.ReadLine()
        name2 = myFile.ReadLine()
        name3 = myFile.ReadLine()

        ' Close the file.
        myFile.Close()

        ' Display the names that were read.
        Console.WriteLine("Here are the names of three philosophers:")
        Console.WriteLine(name1)
        Console.WriteLine(name2)
        Console.WriteLine(name3)
    End Sub
End Module
```

Don't Forget this!



```
Here are the names of three philosophers:
John Locke
David Hume
Edmund Burke
Press any key to continue . . .
```

This is the program's output.

Here are some specific points about the program:

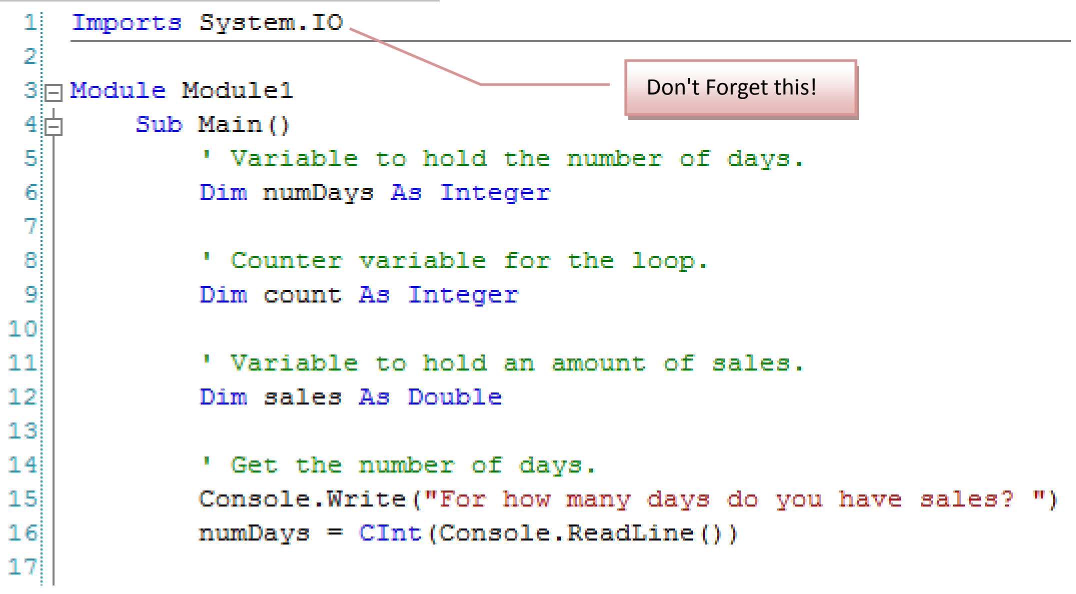
- Line 6 declares a `StreamReader` variable named `myFile`.
- Line 10 declares the `String` variables `name1`, `name2`, and `name3`. These will hold the lines of input that will be read from the file.
- Line 13 opens a file on the disk named `philosophers.txt`, and associates it with the `myFile` variable.
- Line 17 reads a line of text from the file and assigns it, as a string, to the `name1` variable.
- Line 18 reads the next line of text from the file and assigns it, as a string, to the `name2` variable.
- Line 19 reads the next line of text from the file and assigns it, as a string, to the `name3` variable.
- Line 22 closes the file.
- Lines 26, 27, and 28 display the strings that were read from the file.

Using Loops to Process Files

Program 10-3 demonstrates how a loop can be used to collect items of data to be stored in a file. This is the Visual Basic version of pseudocode Program 10-3 in your textbook.

Program 10-3

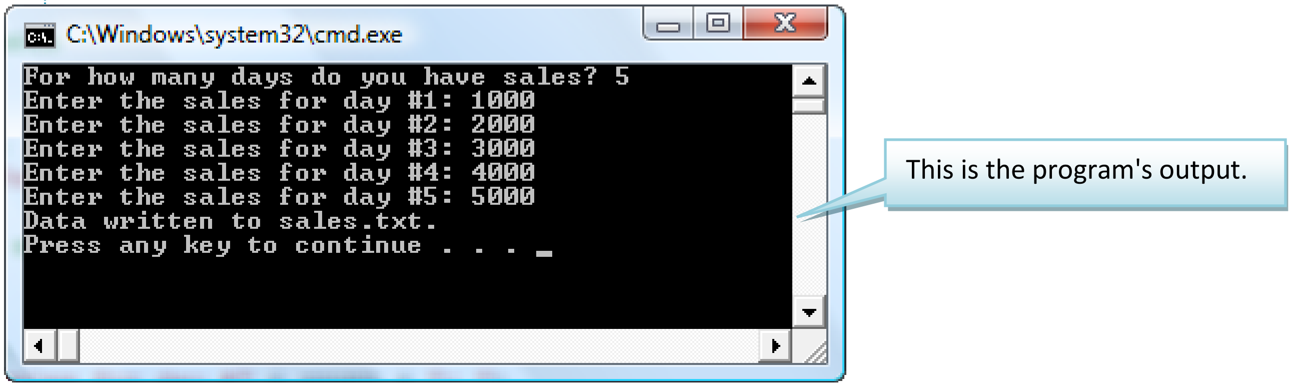
This program is the VB version of **Program 10-3** in your textbook!



```
Imports System.IO

Module Module1
    Sub Main()
        ' Variable to hold the number of days.
        Dim numDays As Integer

        ' Counter variable for the loop.
        Dim count As Integer

        ' Variable to hold an amount of sales.
        Dim sales As Double

        ' Get the number of days.
        Console.Write("For how many days do you have sales? ")
        numDays = CInt(Console.ReadLine())

```

Program continues on next page….

```
        ' Declare a StreamReader variable
        Dim salesFile As StreamWriter

        ' Open a file named Sales.txt.
        salesFile = File.CreateText("Sales.txt")

        ' Get the amount of sales for each day and write
        ' it to the file.
        For count = 1 To numDays
            ' Get the sales for a day.
            Console.Write("Enter the sales for day #" & count & ": ")
            sales = CDbl(Console.ReadLine())

            ' Write the amount to the file.
            salesFile.WriteLine(sales)
        Next

        ' Close the file.
        salesFile.Close()
        Console.WriteLine("Data written to sales.txt.")
    End Sub
End Module
```



```
For how many days do you have sales? 5
Enter the sales for day #1: 1000
Enter the sales for day #2: 2000
Enter the sales for day #3: 3000
Enter the sales for day #4: 4000
Enter the sales for day #5: 5000
Data written to sales.txt.
Press any key to continue . . . _
```

Detecting the End of a File

Sometimes you need to read a file's contents, and you do not know the number of items that are stored in the file. When this is the case, you can use a `StreamReader` method named `Peek` to determine whether the file contains another item before you attempt to read an item from it. This method looks ahead in the file, without moving the current read position, and returns the next character that will be read. If the current read position is at the end of the file (where there are no more characters to read), the method returns –1.

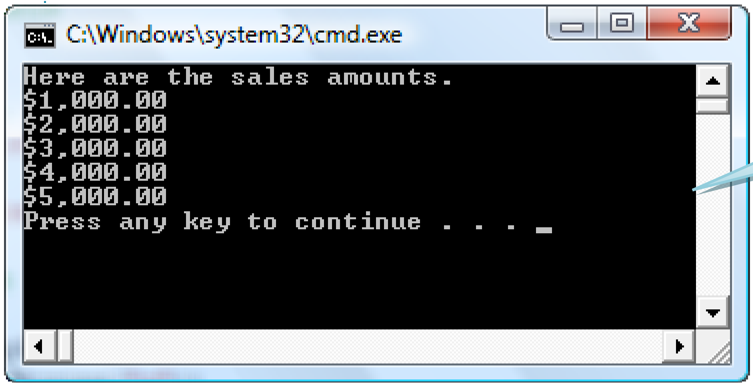
Program 10-4 demonstrates how to use the `Peek` method. This is the Visual Basic version of pseudocode Program 10-4 in your textbook. The program opens the *sales.txt* file that was created by Program 10-3. It reads and displays each item of data in the file.

Program 10-4

This program is the VB version of **Program 10-4** in your textbook!

```
Imports System.IO

Module Module1
    Sub Main()
        ' Variable to hold an amount of sales.
        Dim sales As Double

        ' Declare a StreamReader variable
        Dim salesFile As StreamReader

        ' Open a file named Sales.txt.
        salesFile = File.OpenText("Sales.txt")

        Console.WriteLine("Here are the sales amounts.")

        ' Read all of the items in the file and display them.
        Do Until salesFile.Peek = -1
            sales = CDbl(salesFile.ReadLine())
            Console.WriteLine(sales.ToString("c"))
        Loop
        ' Close the file.
        salesFile.Close()
    End Sub
End Module
```

Don't Forget this!



```
Here are the sales amounts.
$1,000.00
$2,000.00
$3,000.00
$4,000.00
$5,000.00
Press any key to continue . . . _
```

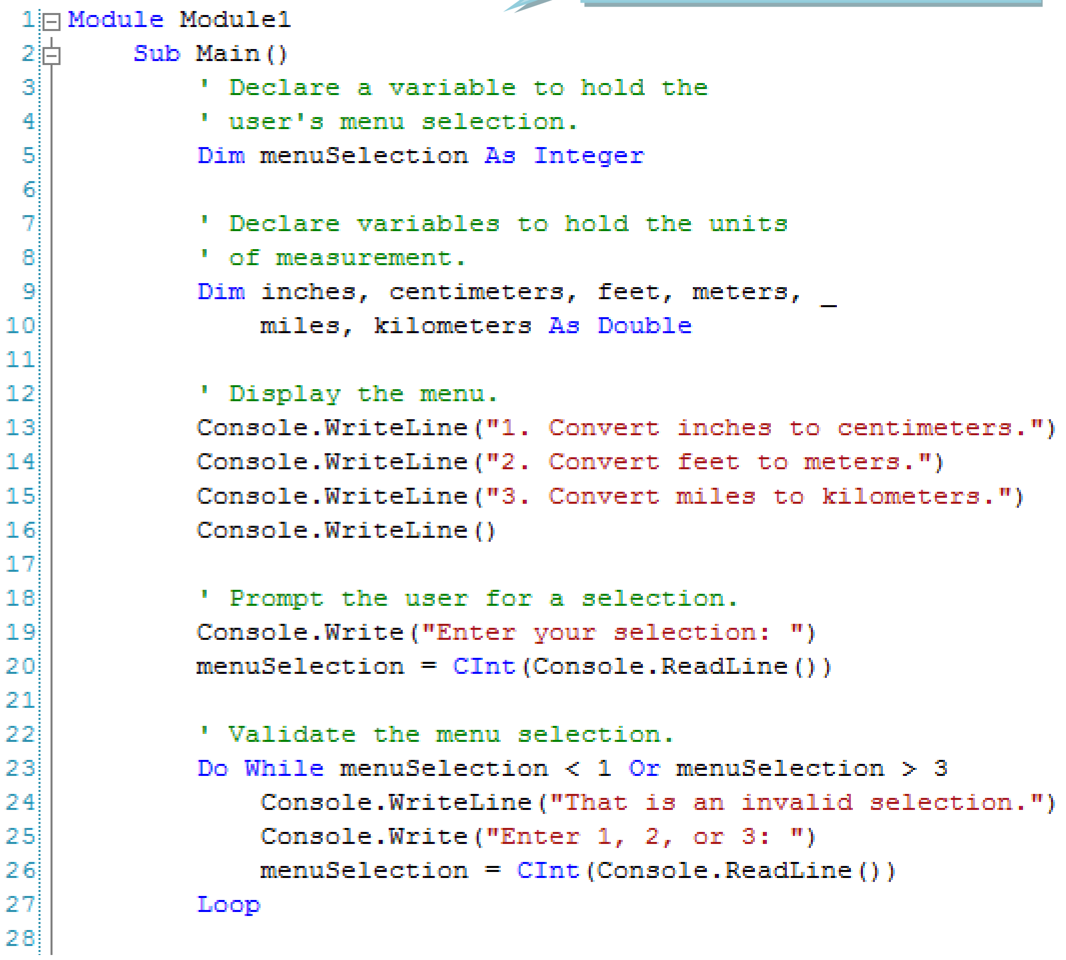
This is the program's output.

Chapter 11 Menu-Driven Programs

Chapter 11 in your textbook discusses menu-driven programs. A menu-driven program presents a list of operations that the user may select from (the menu), and then performs the operation that the user selected. There are no new language features introduced in the chapter, so here we will simply show you a Visual Basic program that is menu-driven. Program 11-1 is the Visual Basic version of the pseudocode Program 11-3.

Program 11-1

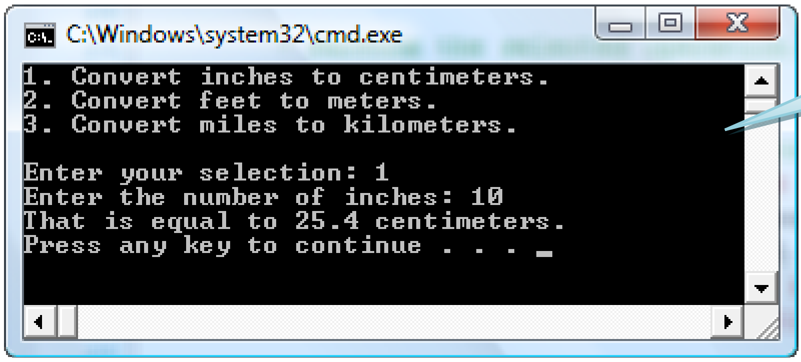
This program is the VB version of **Program 11-3** in your textbook!



```
Module Module1
    Sub Main()
        ' Declare a variable to hold the
        ' user's menu selection.
        Dim menuSelection As Integer

        ' Declare variables to hold the units
        ' of measurement.
        Dim inches, centimeters, feet, meters, _
            miles, kilometers As Double

        ' Display the menu.
        Console.WriteLine("1. Convert inches to centimeters.")
        Console.WriteLine("2. Convert feet to meters.")
        Console.WriteLine("3. Convert miles to kilometers.")
        Console.WriteLine()

        ' Prompt the user for a selection.
        Console.Write("Enter your selection: ")
        menuSelection = CInt(Console.ReadLine())

        ' Validate the menu selection.
        Do While menuSelection < 1 Or menuSelection > 3
            Console.WriteLine("That is an invalid selection.")
            Console.Write("Enter 1, 2, or 3: ")
            menuSelection = CInt(Console.ReadLine())
        Loop

```

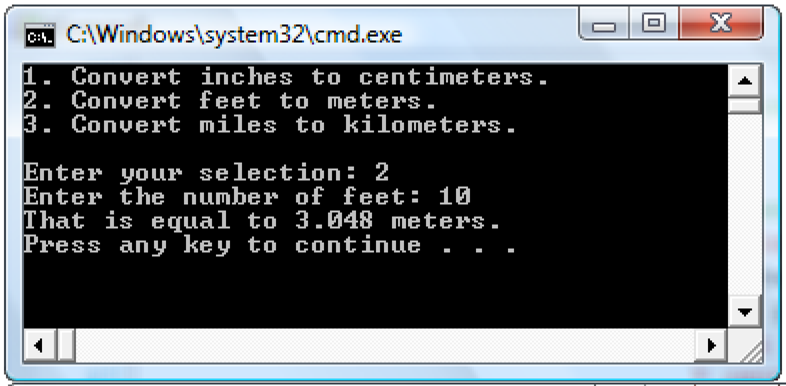
Program continues on next page….

```
        ' Perform the selected operation.
        Select Case menuSelection
            Case 1
                ' Convert inches to centimeters.
                Console.Write("Enter the number of inches: ")
                inches = CDbl(Console.ReadLine())
                centimeters = inches * 2.54
                Console.WriteLine("That is equal to " & centimeters & _
                                  " centimeters.")

            Case 2
                ' Convert feet to meters.
                Console.Write("Enter the number of feet: ")
                feet = CDbl(Console.ReadLine())
                meters = feet * 0.3048
                Console.WriteLine("That is equal to " & meters & _
                                  " meters.")

            Case 3
                ' Convert miles to kilometers.
                Console.Write("Enter the number of miles: ")
                miles = CDbl(Console.ReadLine())
                kilometers = miles * 1.609
                Console.WriteLine("That is equal to " & kilometers & _
                                  " kilometers.")

        End Select
    End Sub
End Module
```

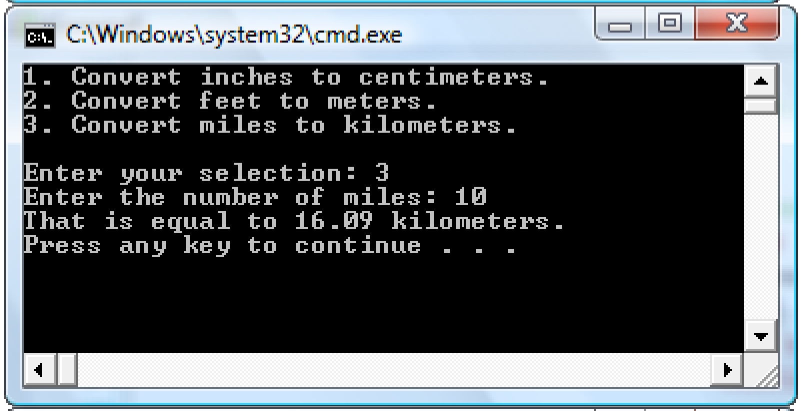


```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 1
Enter the number of inches: 10
That is equal to 25.4 centimeters.
Press any key to continue . . .
```

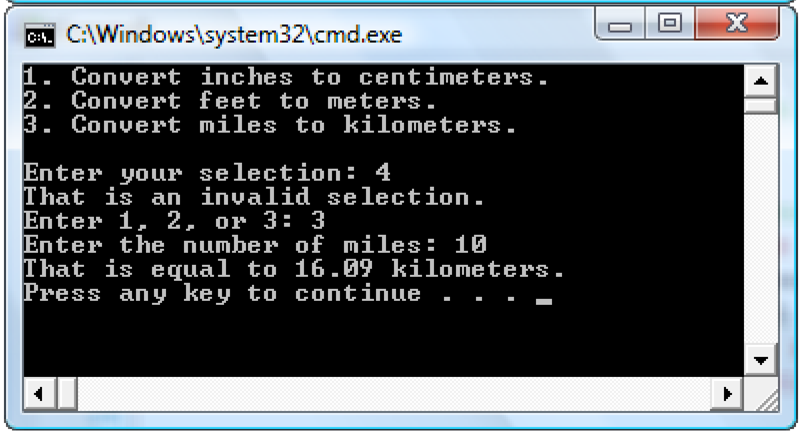
This is the program's output.


C:\Windows\system32\cmd.exe

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 2
Enter the number of feet: 10
That is equal to 3.048 meters.
Press any key to continue . . .
```


C:\Windows\system32\cmd.exe

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 3
Enter the number of miles: 10
That is equal to 16.09 kilometers.
Press any key to continue . . .
```


C:\Windows\system32\cmd.exe

```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 4
That is an invalid selection.
Enter 1, 2, or 3: 3
Enter the number of miles: 10
That is equal to 16.09 kilometers.
Press any key to continue . . . _
```

Chapter 12 Text Processing

Character-By-Character Text Processing

Chapter 12 in your textbook discusses programming techniques for working with the individual characters in a string. Visual Basic allows you to retrieve the individual characters in a string using subscript notation, as described in the book. (Parentheses are used instead of square brackets, however.) For example, the following code creates the string "Hello", and then uses subscript notation to print the first character in the string:

```
Dim greeting As String = "Hello"
Console.WriteLine(greeting(0))
```

Although you can use subscript notation to retrieve the individual characters in a string, you cannot use it to change the value of a character within a string in Visual Basic. As a result, you cannot use an expression in the form `string(index)` on the left side of an assignment operator. For example, the following code will cause an error:

```
' Assign "Bill" to friendName.
Dim friendName As String = "Bill"

' Can we change the first character to "J"?
friendName(0) = "J"     ' No, this will cause an error!
```

The last statement in this code will cause an error because it attempts to change the value of the first character in the string variable `friendName`.

In this chapter we will show you how to do the following in Visual Basic:

- Use subscript notation to access the individual characters in a string.
- Use string testing methods.

However, because we can't use subscript notation to change the characters in a string, we will not be able to show a simple Visual Basic version of pseudocode Program 12-3.

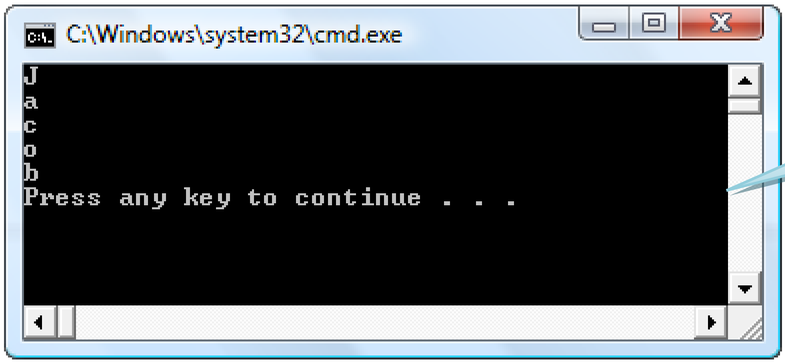
Character-By-Character Text Processing

Program 12-1 shows the Visual Basic version of pseudocode Program 12-1 in the textbook.

Program 12-1

This program is the VB version of **Program 12-1** in your textbook!

```
Module Module1
    Sub Main()
        ' Declare and initialize a string.
        Dim name As String = "Jacob"

        ' Use subscript notation to display the
        ' individual characters in the string.
        Console.WriteLine(name(0))
        Console.WriteLine(name(1))
        Console.WriteLine(name(2))
        Console.WriteLine(name(3))
        Console.WriteLine(name(4))
    End Sub
End Module
```



```
J
a
c
o
b
Press any key to continue . . .
```

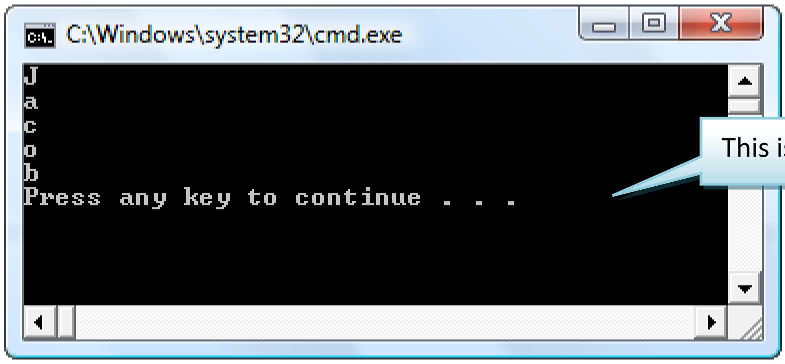
This is the program's output.

Program 12-2 is the Visual Basic version of pseudocode Program 12-1 in your textbook. This program uses a `For-Next` loop to step through all of the characters in a string.

Program 12-2

This program is the VB version of **Program 12-2** in your textbook!

```
Module Module1
    Sub Main()
        ' Declare and initialize a string.
        Dim name As String = "Jacob"

        ' Declare a variable to step through the string.
        Dim index As Integer

        ' Display the characters in the string.
        For index = 0 To name.Length - 1
            Console.WriteLine(name(index))
        Next
    End Sub
End Module
```



```
J
a
c
o
b
Press any key to continue . . .
```

This is the program's output.

Character Testing Methods

Visual Basic provides functions that are similar to the character testing library functions shown in Table 12-2 in your textbook. The Visual Basic functions that are similar to those functions are shown here, in Table 12-1.

Table 12-1 Character Testing Methods

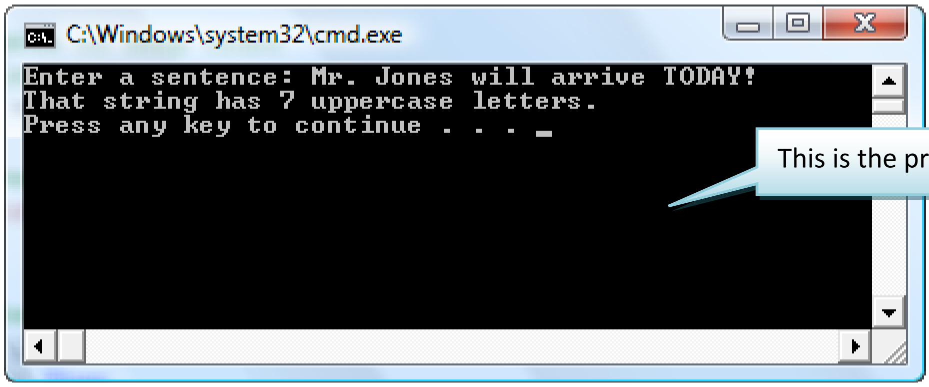
Function	Description
`Char.IsDigit(`*`character`*`)`	Returns `True` if *`character`* is a numeric digit, or `False` otherwise.
`Char.IsLetter(`*`character`*`)`	Returns `True` if *`character`* is an alphabetic letter or `False` otherwise.
`Char.IsLower(`*`character`*`)`	Returns `True` if *`character`* is a lowercase letter or `False` otherwise.
`Char.IsUpper(`*`character`*`)`	Returns `True` if *`character`* is an uppercase letter or `False` otherwise.
`Char.IsWhiteSpace(`*`character`*`)`	Returns `True` if *`character`* is a whitespace character or `False` otherwise. (A whitespace character is a space, a tab, or a newline.)

Program 12-3 demonstrates how the `Char.IsUpper` function is used. This program is the Visual Basic version of Program 12-4 in your textbook.

Program 12-3

This program is the VB version of **Program 12-4** in your textbook!

```
Module Module1
    Sub Main()
        ' Declare a string to hold input.
        Dim str As String

        ' Declare a variable to step through the string.
        Dim index As Integer

        ' Declare an accumulator variable to keep count
        ' of the number of uppercase letters.
        Dim upperCaseCount As Integer = 0

        ' Prompt the user to enter a sentence.
        Console.Write("Enter a sentence: ")
        str = Console.ReadLine()

        ' Display the characters in the string.
        For index = 0 To str.Length - 1
            If Char.IsUpper(str(index)) Then
                upperCaseCount = upperCaseCount + 1
            End If
        Next

        ' Display the number of uppercase characters.
        Console.WriteLine("That string has " & upperCaseCount & _
                          " uppercase letters.")
    End Sub
End Module
```



```
Enter a sentence: Mr. Jones will arrive TODAY!
That string has 7 uppercase letters.
Press any key to continue . . .
```

This is the program's output.

Inserting and Deleting Characters

Visual Basic strings have methods for inserting and removing (deleting) characters. These methods are similar to the library modules that are shown in Table 12-3 in your textbook. The `String` methods that are similar to those functions are shown here, in Table 12-2.

Table 12-2 String Insert and Remove Methods

Function	Description
`StringVar``.Insert(`*`position, string2`*`)`	*`StringVar`* is the name of a `String` variable, *`position`* is an `Integer`, and *`string2`* is a string. The method returns a copy of the string in *`StringVar`* with *`string2`* inserted, beginning at *`position`*.
`StringVar``.Remove(`*`start, count`*`)`	*`StringName`* is the name of a `String` variable, *`start`* is an `Integer`, and *`count`* is an `Integer`. The method returns a copy of the string in *`StringVar`* with `count` characters removed, beginning at the position specified by*`start`*.

Here is an example of how we might use the `Insert` method:

```
Dim str As String = "New City"
Dim strCopy As String
strCopy = str.Insert(4, "York ")
Console.WriteLine(strCopy)
```

The first statement declares the `String` variable `str`, initialized with "New City". The second statement declares a `String` variable named `strCopy`. In the third statement, the `str.Insert` method returns a copy of `str` with the string `"York "` inserted, beginning at position 4. The copy that is returned is assigned to the `strCopy` variable. If these statements were a complete program and we ran it, we would see *New York City* displayed on the screen.

Here is an example of how we might use the `Remove` method:

```
Dim str As String = "I ate 1000 blueberries!"
str = str.Remove(8, 2)
Console.WriteLine(str)
```

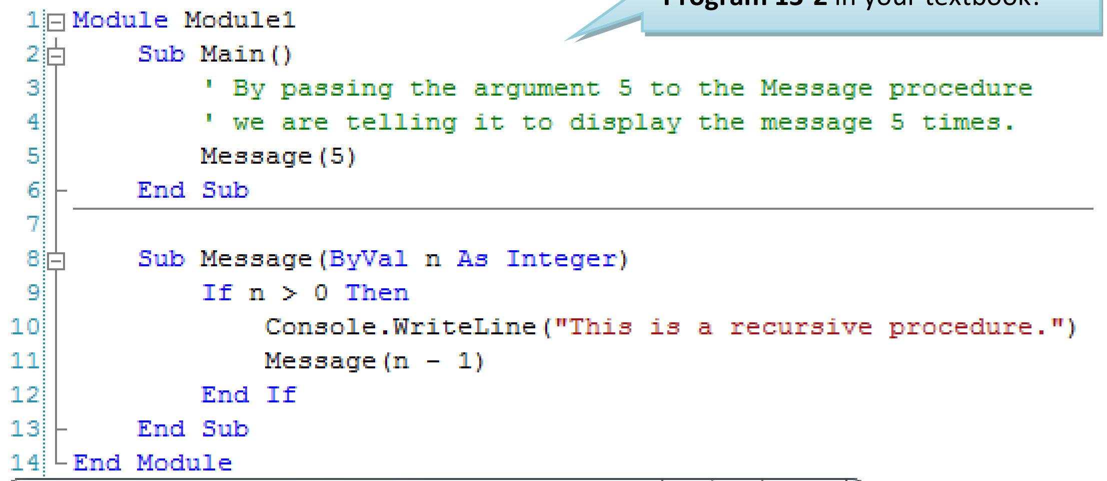
The first statement declares the `String` variable `str`, initialized with "I ate 1000 blueberries!". The second statement declares a `String` variable named `strCopy`. In the third statement, the `str.Remove` method returns a copy of `str` with 2 characters removed, beginning at position 8. If these statements were a complete program and we ran it, we would see *I ate 10 blueberries!* displayed on the screen.

Chapter 13 Recursion

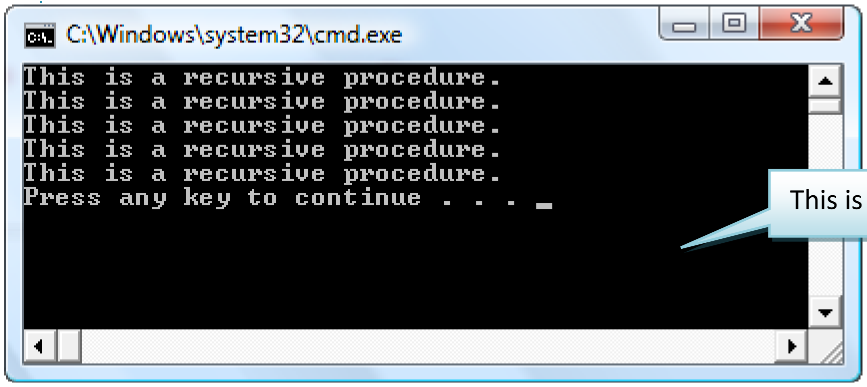
A Visual Basic procedure or function can call itself recursively, allowing you to design algorithms that recursively solve a problem. Chapter 13 in your textbook describes recursion in detail, discusses problem solving with recursion, and provides several pseudocode examples. Other than the technique of a method recursively calling itself, no new language features are introduced. In this chapter we will present Visual Basic versions of two of the pseudocode programs that are shown in the textbook. Both of these programs work exactly as the algorithms are described in the textbook. Program 13-1 is the Visual Basic version of pseudocode Program 13-2.

Program 13-1

This program is the VB version of **Program 13-2** in your textbook!



```
Module Module1
    Sub Main()
        ' By passing the argument 5 to the Message procedure
        ' we are telling it to display the message 5 times.
        Message(5)
    End Sub

    Sub Message(ByVal n As Integer)
        If n > 0 Then
            Console.WriteLine("This is a recursive procedure.")
            Message(n - 1)
        End If
    End Sub
End Module
```



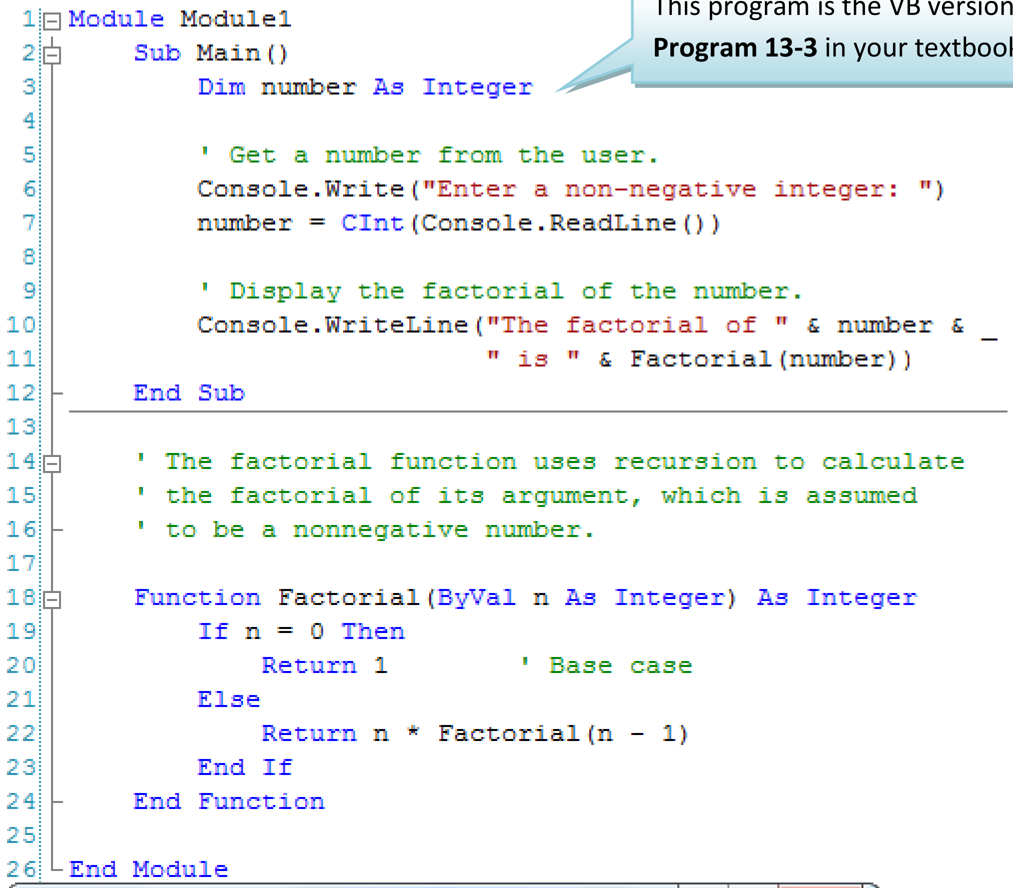
```
This is a recursive procedure.
This is a recursive procedure.
This is a recursive procedure.
This is a recursive procedure.
This is a recursive procedure.
Press any key to continue . . .
```

This is the program's output.

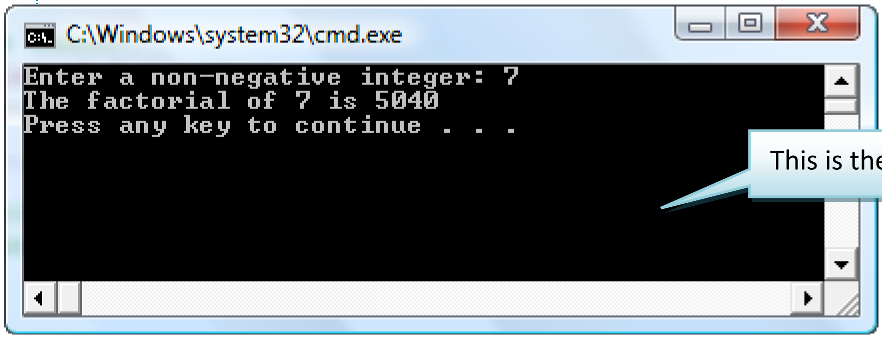
Next, Program 13-2 is the Visual Basic version of pseudocode Program 13-3. This program recursively calculates the factorial of a number.

Program 13-2

This program is the VB version of **Program 13-3** in your textbook!



```
Module Module1
    Sub Main()
        Dim number As Integer

        ' Get a number from the user.
        Console.Write("Enter a non-negative integer: ")
        number = CInt(Console.ReadLine())

        ' Display the factorial of the number.
        Console.WriteLine("The factorial of " & number & _
                          " is " & Factorial(number))
    End Sub

    ' The factorial function uses recursion to calculate
    ' the factorial of its argument, which is assumed
    ' to be a nonnegative number.

    Function Factorial(ByVal n As Integer) As Integer
        If n = 0 Then
            Return 1           ' Base case
        Else
            Return n * Factorial(n - 1)
        End If
    End Function

End Module
```



C:\Windows\system32\cmd.exe

```
Enter a non-negative integer: 7
The factorial of 7 is 5040
Press any key to continue . . .
```

This is the program's output.

Chapter 14 Object-Oriented Programming

Visual Basic is a powerful object-oriented language. An object is an entity that exists in the computer's memory while the program is running. An object contains data and has the ability to perform operations on its data. An object's data is commonly referred to as the object's fields, and the operations that the object performs are the object's methods.

In addition to the many objects that are provided by Visual Basic, you can create objects of your own design. The first step is to write a class. A class is like a blueprint. It is a declaration that specifies the methods for a particular type of object. When the program needs an object of that type, it creates an instance of the class. (An object is an instance of a class.)

Here is the general format of a class declaration in Visual Basic:

```
Public Class ClassName
{
    Field declarations and method definitions go here…
}
```

In Visual Basic, a class declaration is stored in its own file, and the file is part of the project in which you want to use it. To create a class in an existing Visual Basic project, click *Project* on the Visual Basic menu bar, and then click *Add Class*… This is shown in Figure 14-1. You will then see the Add New Item window, as shown in Figure 12-2.

Figure 14-1 Adding a new class to a project

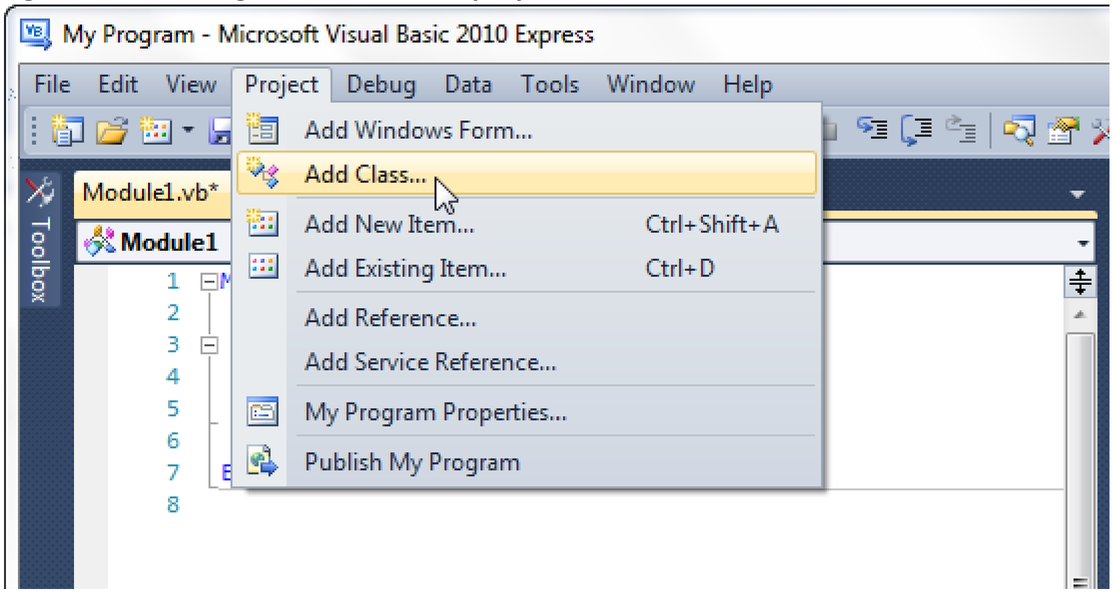
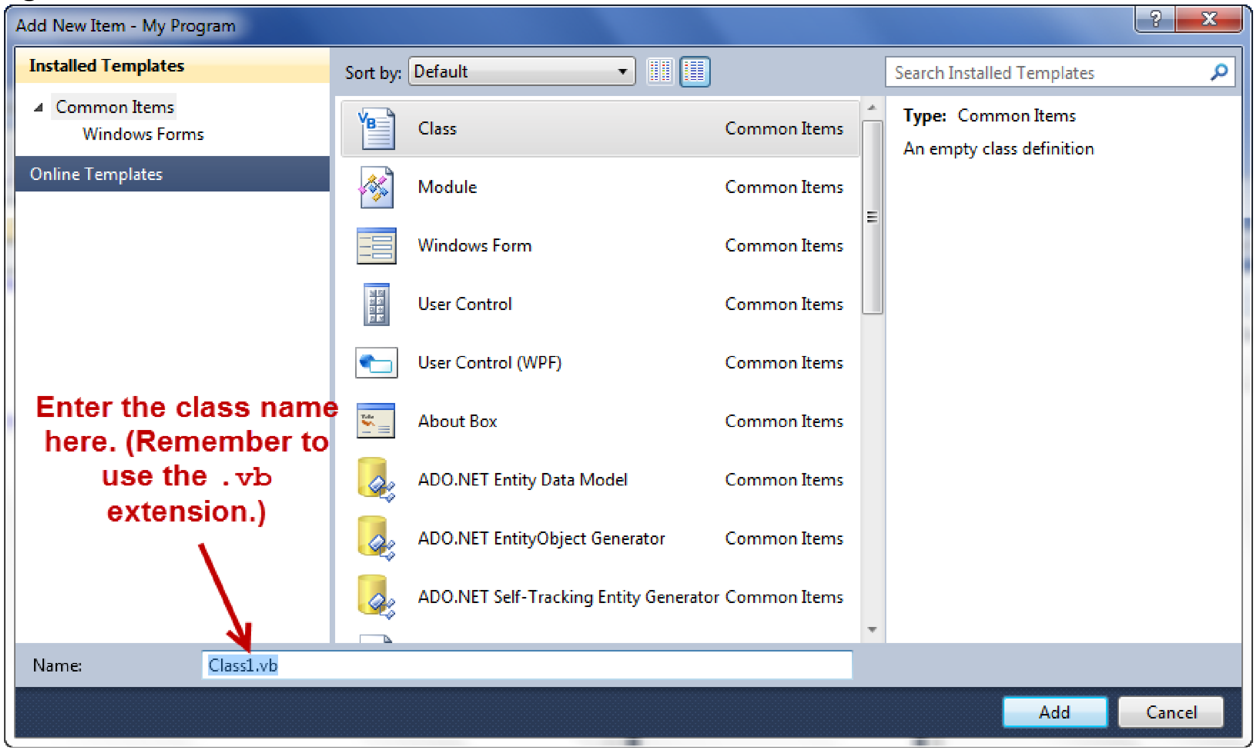


Figure 14-2 The *Add New Item* window


Add New Item - My Program
Installed Templates
Common Items
Windows Forms
Online Templates
Sort by: Default
Search Installed Templates
Class Common Items
Module Common Items
Windows Form Common Items
User Control Common Items
User Control (WPF) Common Items
About Box Common Items
ADO.NET Entity Data Model Common Items
ADO.NET EntityObject Generator Common Items
ADO.NET Self-Tracking Entity Generator Common Items
Type: Common Items
An empty class definition
Enter the class name here. (Remember to use the .vb extension.)
Name: Class1.vb
Add
Cancel

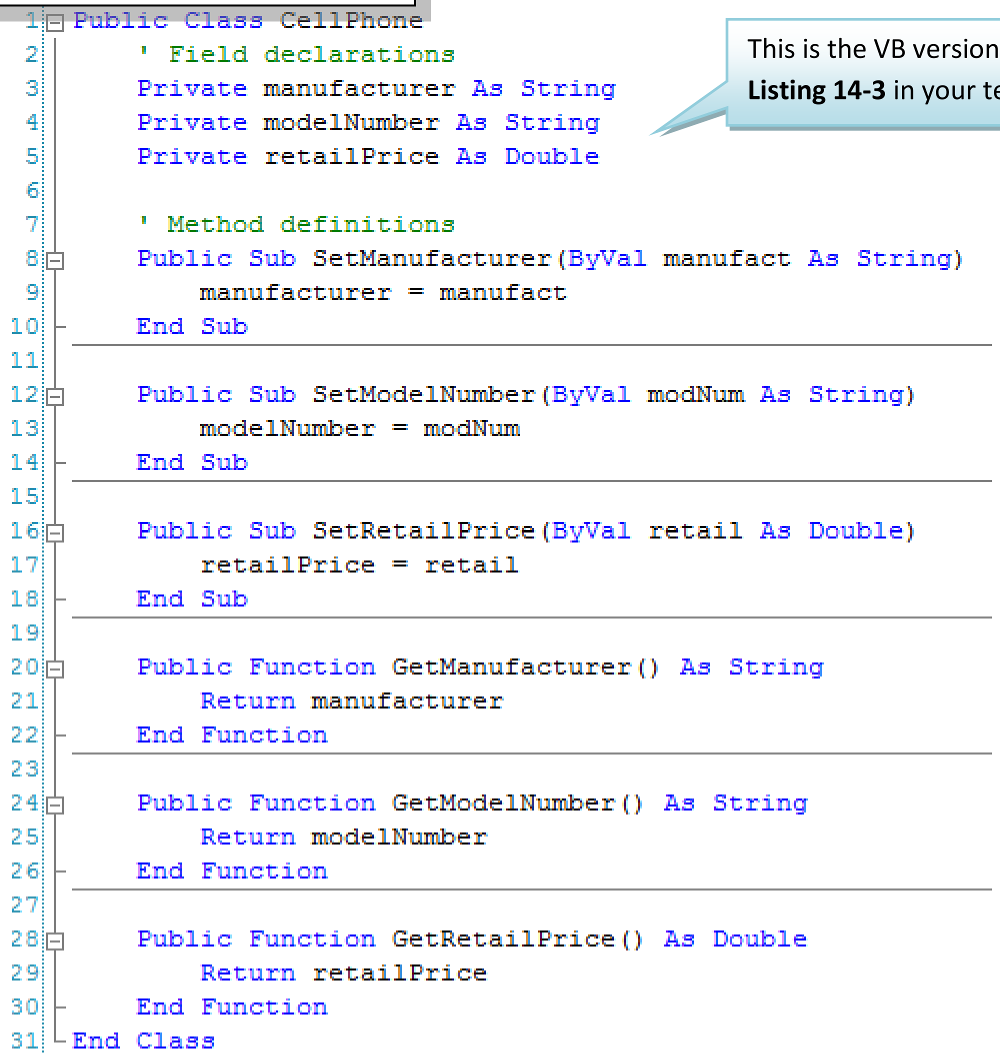
Notice the *Name* box at the bottom of the Add New Item window. This is where you type the name of the file that will contain the class declaration. The filename should be the same as the name of the class, followed by the *.vb* extension. For example, if you want to create a class named `Customer`, you would use the filename *Customer.vb*. When you click the *Add* button, a file will appear in the code editor, with the first and last line of the class declaration already written for you.

Chapter 14 in your textbook steps through the design of a `CellPhone` class. Class Listing 14-1 on the next page is the Visual Basic version of the `CellPhone` class shown in Class Listing 14-3 in your textbook. Notice that each of the field declarations (lines 3 through 5) begin with the key word `Private`. This is an access specifier that makes the fields private to the class. No code outside the class can directly access private class fields. Also notice that each of the method headers begin with the `Public` access specifier. This makes the methods public, which means that code outside the class can call the methods.

Program 14-1 shows how to create an instance of the `CellPhone` class. This is the Visual Basic version of pseudocode Program 14-1.

Class Listing 14-1

This is the VB version of **Class Listing 14-3** in your textbook!

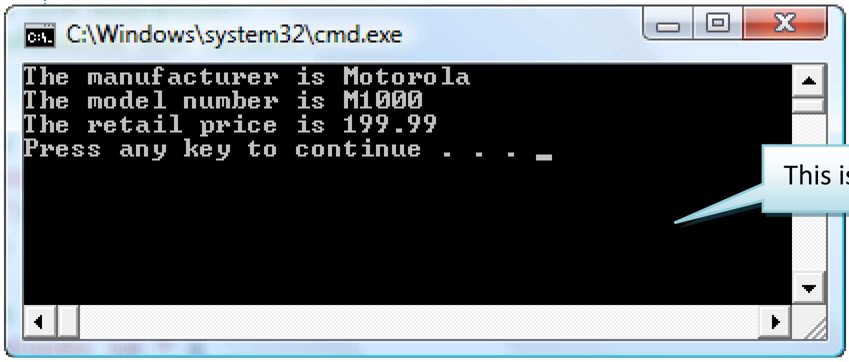


```
Public Class CellPhone
    ' Field declarations
    Private manufacturer As String
    Private modelNumber As String
    Private retailPrice As Double

    ' Method definitions
    Public Sub SetManufacturer(ByVal manufact As String)
        manufacturer = manufact
    End Sub

    Public Sub SetModelNumber(ByVal modNum As String)
        modelNumber = modNum
    End Sub

    Public Sub SetRetailPrice(ByVal retail As Double)
        retailPrice = retail
    End Sub

    Public Function GetManufacturer() As String
        Return manufacturer
    End Function

    Public Function GetModelNumber() As String
        Return modelNumber
    End Function

    Public Function GetRetailPrice() As Double
        Return retailPrice
    End Function
End Class
```

Program 14-1

This is the VB version of **Program 14-1** in your textbook!

```
 1 Module Module1
 2     Sub Main()
 3         ' Declare a variable that can reference
 4         ' a CellPhone object.
 5         Dim myPhone As CellPhone
 6
 7         ' The following statement creates an object
 8         ' using the CellPhone class as its blueprint.
 9         ' The myPhone variable will reference the object.
10         myPhone = New CellPhone()
11
12         ' Store values in the object's fields.
13         myPhone.SetManufacturer("Motorola")
14         myPhone.SetModelNumber("M1000")
15         myPhone.SetRetailPrice(199.99)
16
17         ' Display the values stored in the fields.
18         Console.WriteLine("The manufacturer is " & _
19                           myPhone.GetManufacturer())
20         Console.WriteLine("The model number is " & _
21                           myPhone.GetModelNumber())
22         Console.WriteLine("The retail price is " & _
23                           myPhone.GetRetailPrice())
24     End Sub
25 End Module
```



C:\Windows\system32\cmd.exe

```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
Press any key to continue . . .
```

This is the program's output.

The statement in line 5 declares a `CellPhone` variable name `myPhone`. As described in your textbook, this is a special type of variable that can be used to reference a `CellPhone` object. This statement does not, however, create a `CellPhone` object in memory. That is done in line 10. The expression on the right side of the = operator creates a new `CellPhone` object in memory, and the = operator assigns the object's memory address to the `myPhone` variable. As a result, we say that the `myPhone` variable references a `CellPhone` object.

We can then use the `myPhone` variable to perform operations with the object that it references. This is demonstrated in line 13, where we use the `myPhone` variable to call the `setManufacturer` method. The `"Motorola"` string that we pass as an argument is assigned to the object's `manufacturer` field. Likewise, line 14 uses the `myPhone` variable to call the `setModelNumber` method. That causes the string `"M1000"` to be assigned to the object's `modelNumber` field. And, line 15 uses the `myPhone` variable to call the `setRetailPrice` method. This causes the value 199.99 to be stored in the object's `retailPrice` field. The statements that appear in lines 18 through 23 get the values that are stored in the object's fields and displays them on the screen.

Constructors

A class constructor in Visual Basic is a method named `New`. (The `New` method is written in the class as a `Public Sub` procedure.) When an instance of a class is created in memory, the `New` method (the constructor) is automatically executed. Class Listing 14-2 is a version of the `CellPhone` class that has a constructor. This is the Visual Basic version of Class Listing 14-4 in your textbook. The constructor appears in lines 8 through 13.

Class Listing 14-2

This is the VB version of **Class Listing 14-4** in your textbook!

```
1  Public Class CellPhone
2      ' Field declarations
3      Private manufacturer As String
4      Private modelNumber As String
5      Private retailPrice As Double
6  
7      ' Constructor
8      Public Sub New(ByVal manufact As String, ByVal modNum As String, _
9                     ByVal retail As Double)
10         manufacturer = manufact
11         modelNumber = modNum
12         retailPrice = retail
13     End Sub
14     ' Method definitions
15     Public Sub SetManufacturer(ByVal manufact As String)
16         manufacturer = manufact
17     End Sub
18 
19     Public Sub SetModelNumber(ByVal modNum As String)
20         modelNumber = modNum
21     End Sub
22 
23     Public Sub SetRetailPrice(ByVal retail As Double)
24         retailPrice = retail
25     End Sub
26 
27     Public Function GetManufacturer() As String
28         Return manufacturer
29     End Function
30 
```

Class continues on next page....

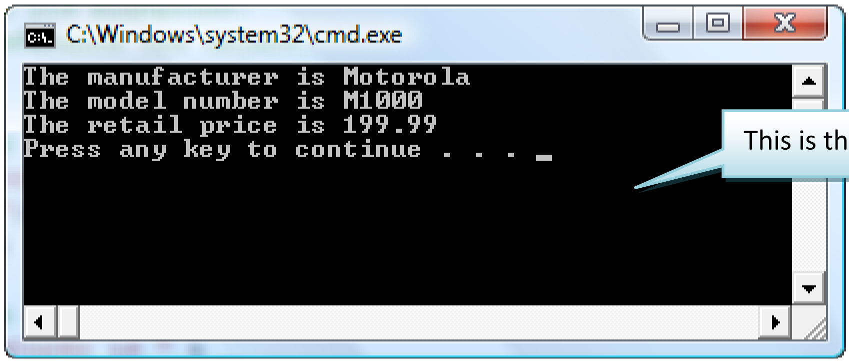
```
    Public Function GetModelNumber() As String
        Return modelNumber
    End Function

    Public Function GetRetailPrice() As Double
        Return retailPrice
    End Function
End Class
```

Program 14-2 demonstrates how to create an instance of the class, passing arguments to the constructor. This is the Visual Basic version of pseudocode Program 14-2. In line 10, an instance of the `CellPhone` class is created and the arguments `"Motorola"`, `"M1000"`, and `199.99` are passed to the constructor.

Program 14-2

This is the VB version of **Program 14-2** in your textbook!

```
 1 Module Module1
 2     Sub Main()
 3         ' Declare a variable that can reference
 4         ' a CellPhone object.
 5         Dim myPhone As CellPhone
 6 
 7         ' The following statement creates an object
 8         ' and initializes its fields with the values
 9         ' passed to the constructor.
10         myPhone = New CellPhone("Motorola", "M1000", 199.99)
11 
12         ' Display the values stored in the fields.
13         Console.WriteLine("The manufacturer is " & _
14                           myPhone.GetManufacturer())
15         Console.WriteLine("The model number is " & _
16                           myPhone.GetModelNumber())
17         Console.WriteLine("The retail price is " & _
18                           myPhone.GetRetailPrice())
19     End Sub
20 End Module
```



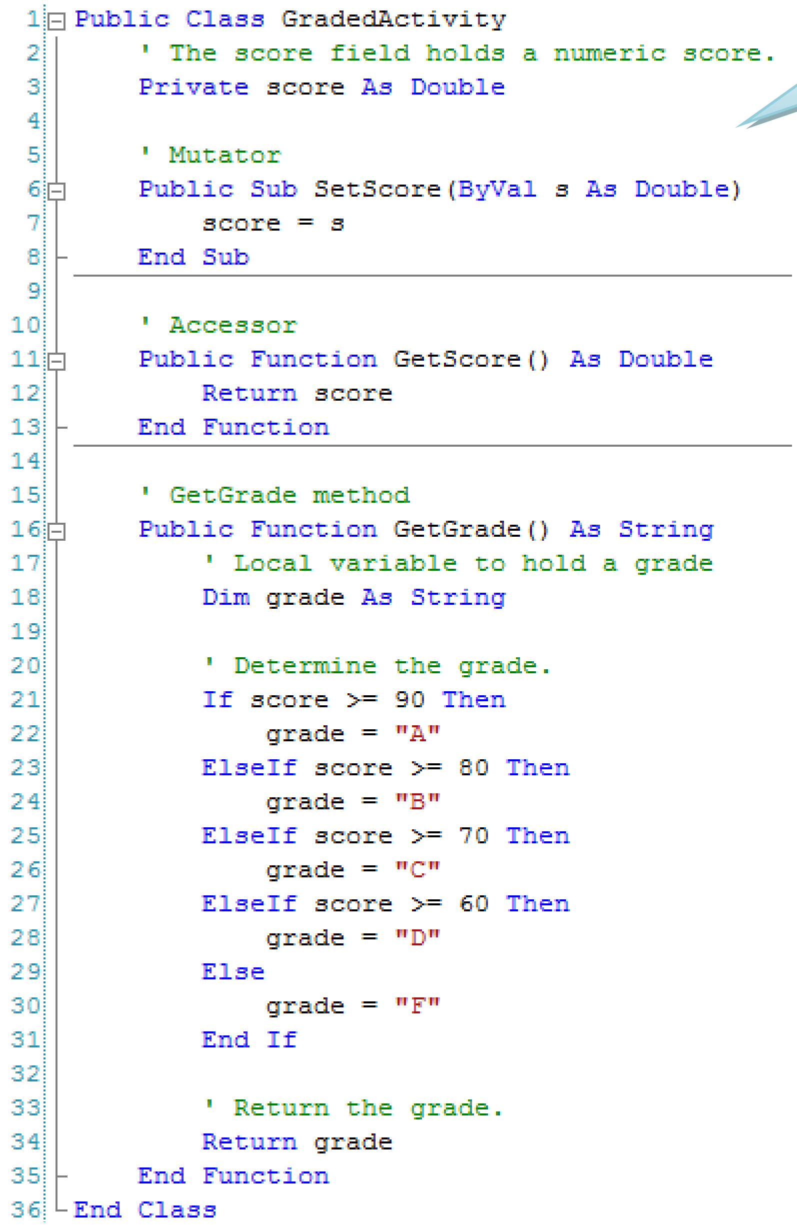
```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
Press any key to continue . . .
```

This is the program's output.

Inheritance

The inheritance example discussed in your textbook starts with the `GradedActivity` class (see Class Listing 14-8), which is used as a superclass. The Visual Basic version of the class is shown here in Class Listing 14-3.

Class Listing 14-3



```
Public Class GradedActivity
    ' The score field holds a numeric score.
    Private score As Double

    ' Mutator
    Public Sub SetScore(ByVal s As Double)
        score = s
    End Sub

    ' Accessor
    Public Function GetScore() As Double
        Return score
    End Function

    ' GetGrade method
    Public Function GetGrade() As String
        ' Local variable to hold a grade
        Dim grade As String

        ' Determine the grade.
        If score >= 90 Then
            grade = "A"
        ElseIf score >= 80 Then
            grade = "B"
        ElseIf score >= 70 Then
            grade = "C"
        ElseIf score >= 60 Then
            grade = "D"
        Else
            grade = "F"
        End If

        ' Return the grade.
        Return grade
    End Function
End Class
```

This is the VB version of **Class Listing 14-8** in your textbook!

A subclass of the `GradedActivity` class named `FinalExam` is shown in your textbook in Class Listing 14-9. The Visual Basic version of the `FinalExam` class is shown here in Class Listing 14-4:

Class Listing 14-4

This is the VB version of **Class Listing 14-9** in your textbook!

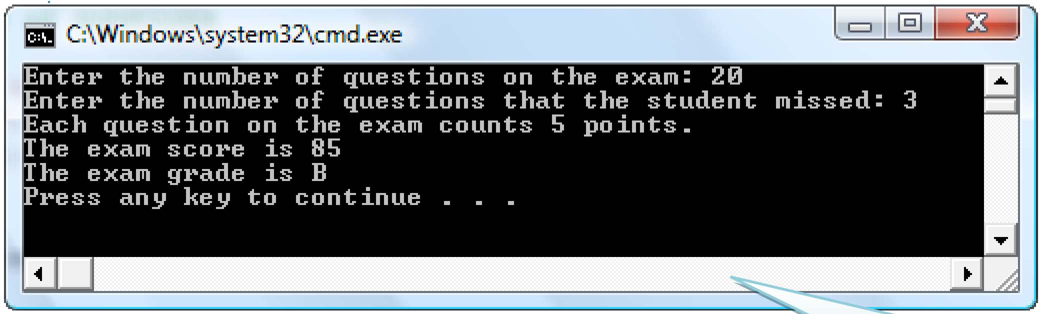
```
Public Class FinalExam
    Inherits GradedActivity

    ' Fields
    Private numQuestions As Integer
    Private pointsEach As Double
    Private numMissed As Integer

    ' The constructor sets the number of
    ' questions on the exam and the number
    ' of questions missed.
    Public Sub New(ByVal questions As Integer, ByVal missed As Integer)
        ' Local variable to hold the numeric score.
        Dim numericScore As Double

        ' the numQuestions and numMissed fields.
        numQuestions = questions
        numMissed = missed

        ' Calculate the points for each question
        ' and the numeric score for this exam.
        pointsEach = 100.0 / questions
        numericScore = 100.0 - (missed * pointsEach)

        ' Call the inherited setScore method to
        ' set the numeric score.
        SetScore(numericScore)
    End Sub

    ' Accessors
    Public Function GetPointsEach() As Double
        Return pointsEach
    End Function

    Public Function GetNumMissed() As Integer
        Return numMissed
    End Function
End Class
```

Notice that in line 2 the `Inherits GradedActivity` clause specifies that the `FinalExam` class inherits from a superclass, `GradedActivity`. Program 14-3 demonstrates the class. This is the Visual Basic version of pseudocode Program 14-5 in your textbook.

Program 14-3

```
Module Module1
    Sub Main()
        ' Variables to hold user input.
        Dim questions, missed As Integer

        ' Class variable to reference a FinalExam object.
        Dim exam As FinalExam

        ' Prompt the user for the number of questions
        ' on the exam.
        Console.Write("Enter the number of questions on the exam: ")
        questions = CInt(Console.ReadLine())

        ' Prompt the user for the number of questions
        ' missed by the student.
        Console.Write("Enter the number of questions that the " & _
                      "student missed: ")
        missed = CInt(Console.ReadLine())

        ' Create a FinalExam object.
        exam = New FinalExam(questions, missed)

        ' Display the test results.
        Console.WriteLine("Each question on the exam counts " & _
                          exam.GetPointsEach() & " points.")
        Console.WriteLine("The exam score is " & exam.GetScore())
        Console.WriteLine("The exam grade is " & exam.GetGrade())
    End Sub
End Module
```

This is the VB version of **Program 14-5** in your textbook!



```
Enter the number of questions on the exam: 20
Enter the number of questions that the student missed: 3
Each question on the exam counts 5 points.
The exam score is 85
The exam grade is B
Press any key to continue . . .
```

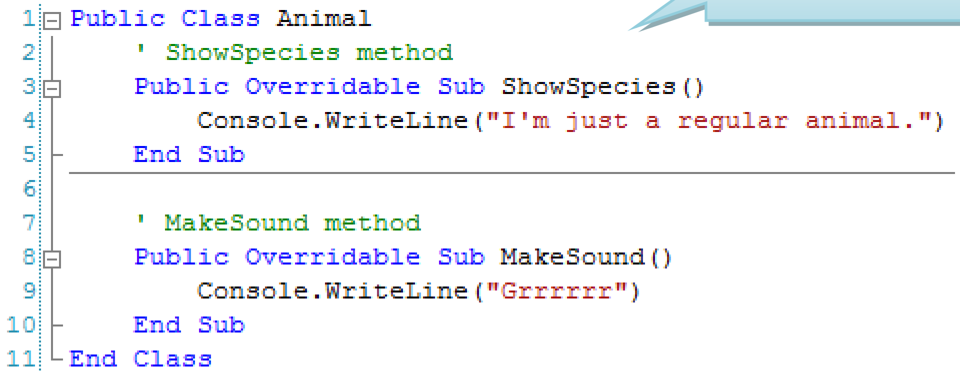
This is the program's output.

Polymorphism

Your textbook presents a polymorphism demonstration that uses the `Animal` class shown in Class Listing 14-10 as a superclass. The Visual Basic version of that class is shown here in Class Listing 14-5:

Class Listing 14-5

This is the VB version of **Class Listing 14-10** in your textbook!



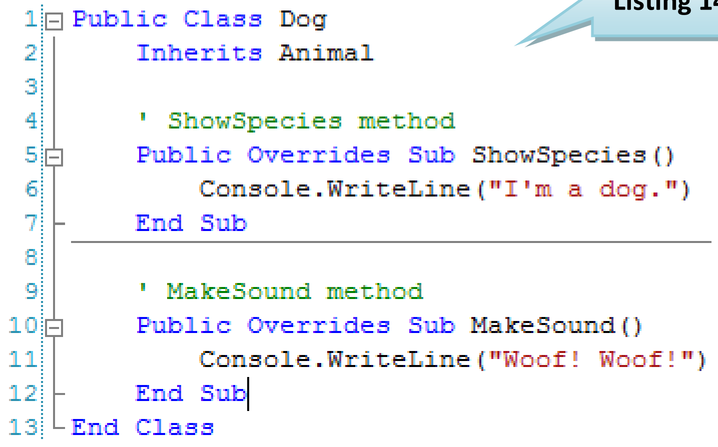
```
 1  Public Class Animal
 2      ' ShowSpecies method
 3      Public Overridable Sub ShowSpecies()
 4          Console.WriteLine("I'm just a regular animal.")
 5      End Sub
 6
 7      ' MakeSound method
 8      Public Overridable Sub MakeSound()
 9          Console.WriteLine("Grrrrrr")
10      End Sub
11  End Class
```

Notice that the key word `Overridable` appears in the method headers for the `ShowSpecies` and `MakeSound` methods (lines 3 and 8). In Visual Basic, you cannot override a method in a subclass unless it is marked in the `Overridable` superclass.

The `Dog` class, which extends the `Animal` class, is shown in Class Listing 14-11 in your textbook. The Visual Basic version of the `Dog` class is shown here in Class Listing 14-6:

Class Listing 14-6

This is the VB version of **Class Listing 14-11** in your textbook!

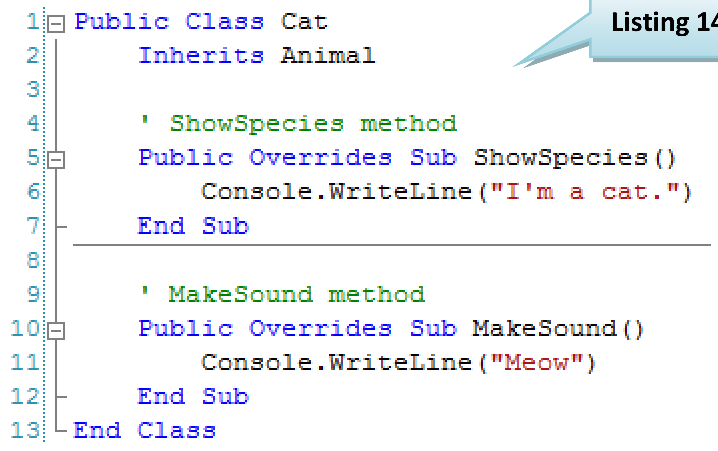


```
 1  Public Class Dog
 2      Inherits Animal
 3
 4      ' ShowSpecies method
 5      Public Overrides Sub ShowSpecies()
 6          Console.WriteLine("I'm a dog.")
 7      End Sub
 8
 9      ' MakeSound method
10      Public Overrides Sub MakeSound()
11          Console.WriteLine("Woof! Woof!")
12      End Sub
13  End Class
```

Notice that the key word `Overrides` appears in the method headers for the `ShowSpecies` and `MakeSound` methods (lines 5 and 10). The `Overrides` key word specifies that the method overrides a method in the superclass.

The `Cat` class, which also extends the `Animal` class, is shown in Class Listing 14-12 in your textbook. The Visual Basic version of the `Cat` class is shown here in Class Listing 14-7:

Class Listing 14-7

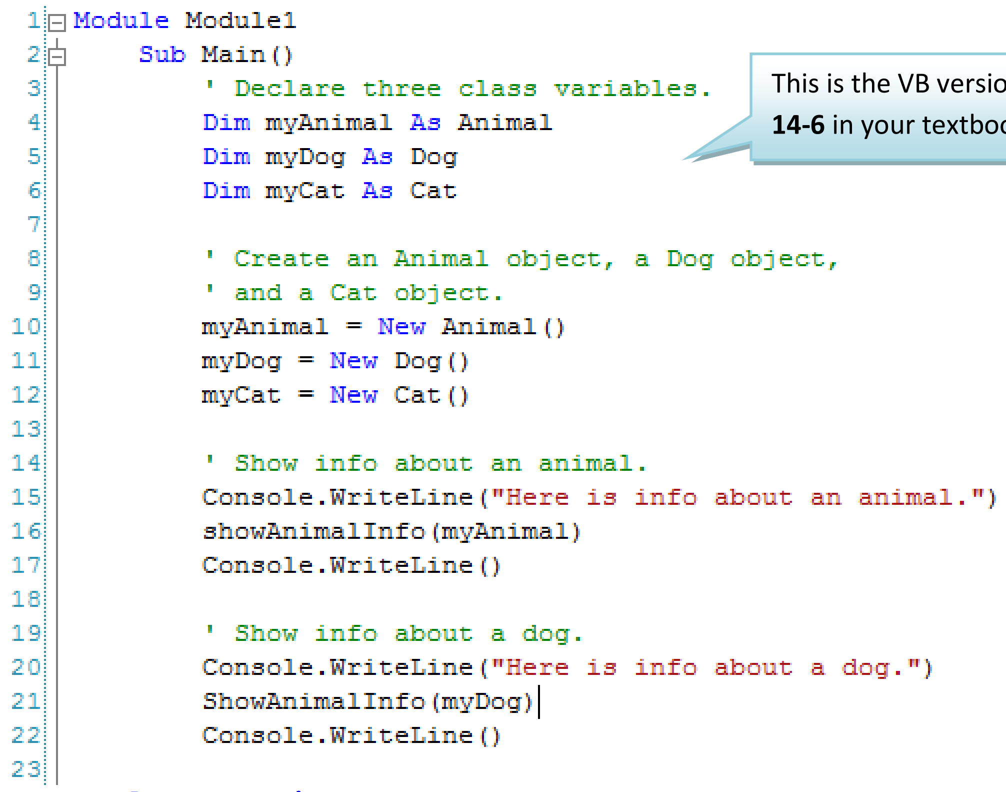


```
Public Class Cat
    Inherits Animal

    ' ShowSpecies method
    Public Overrides Sub ShowSpecies()
        Console.WriteLine("I'm a cat.")
    End Sub

    ' MakeSound method
    Public Overrides Sub MakeSound()
        Console.WriteLine("Meow")
    End Sub
End Class
```

This is the VB version of **Class Listing 14-12** in your textbook!

Program 14-4, shown here, demonstrates the polymorphic behavior of these classes, as discussed in your textbook. This is the Visual Basic version of pseudocode Program 14-6.

Program 14-4



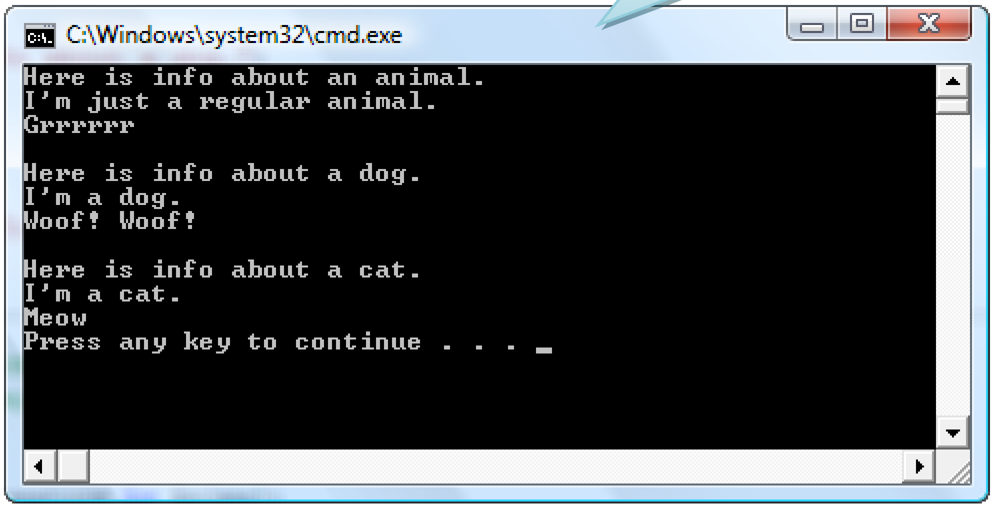
```
Module Module1
    Sub Main()
        ' Declare three class variables.
        Dim myAnimal As Animal
        Dim myDog As Dog
        Dim myCat As Cat

        ' Create an Animal object, a Dog object,
        ' and a Cat object.
        myAnimal = New Animal()
        myDog = New Dog()
        myCat = New Cat()

        ' Show info about an animal.
        Console.WriteLine("Here is info about an animal.")
        showAnimalInfo(myAnimal)
        Console.WriteLine()

        ' Show info about a dog.
        Console.WriteLine("Here is info about a dog.")
        ShowAnimalInfo(myDog)
        Console.WriteLine()

```

This is the VB version of **Program 14-6** in your textbook!

Program continues on next page….

```
            ' Show info about a cat.
            Console.WriteLine("Here is info about a cat.")
            ShowAnimalInfo(myCat)
        End Sub

        ' The showAnimalInfo procedure accepts an Animal
        ' object as an argument and displays information
        ' about it.

        Public Sub ShowAnimalInfo(ByVal creature As Animal)
            creature.ShowSpecies()
            creature.MakeSound()
        End Sub
    End Module
```

This is the program's output.



```
Here is info about an animal.
I'm just a regular animal.
Grrrrrr

Here is info about a dog.
I'm a dog.
Woof! Woof!

Here is info about a cat.
I'm a cat.
Meow
Press any key to continue . . .
```

Chapter 15 GUI Applications and Event-Driven Programming

The programs shown in the previous chapters of this booklet were created as Visual Basic console applications. Console applications display screen output in a console window that can display only text. In this chapter we will discuss how you can use Visual Basic to create GUI applications.

In Visual Basic, a GUI application is known as a *Windows Forms Application*. This is because the windows in an application's user interface are referred to as *forms*. To teach you the process of creating a Windows Forms application in Visual Basic, we will use a simple, step by step tutorial to create the test averaging application that is discussed in Chapter 15's *In the Spotlight* sections, appearing on pages 557 through 559 and 562 through 564. Make sure you've read that material before continuing.

Tutorial: Creating the Test Average GUI Application

Step 1: Start Visual Basic.

Step 2: Click the *File* menu, and then click *New Project*. You should see the New Project window shown in Figure 15-1. As shown in the figure, make sure *Windows Forms Application* is selected under Templates. For the project name, Type *Test Average*. Click the *OK* button.

Note: If you are using Visual Studio 2010, you will see an area titled *Project Types* at the left side of the New Project window. Make sure *Visual Basic* is selected.

Step 3: The Visual Basic environment should appear similar to Figure 15-2. Look at the figure carefully and locate the Toolbox, the Solution Explorer window, the Properties window, and the Designer window. If you do not see all of these windows displayed on your screen, go to Step 4. If you see all of these windows, then go to Step 5.

Figure 15-1 New Project window

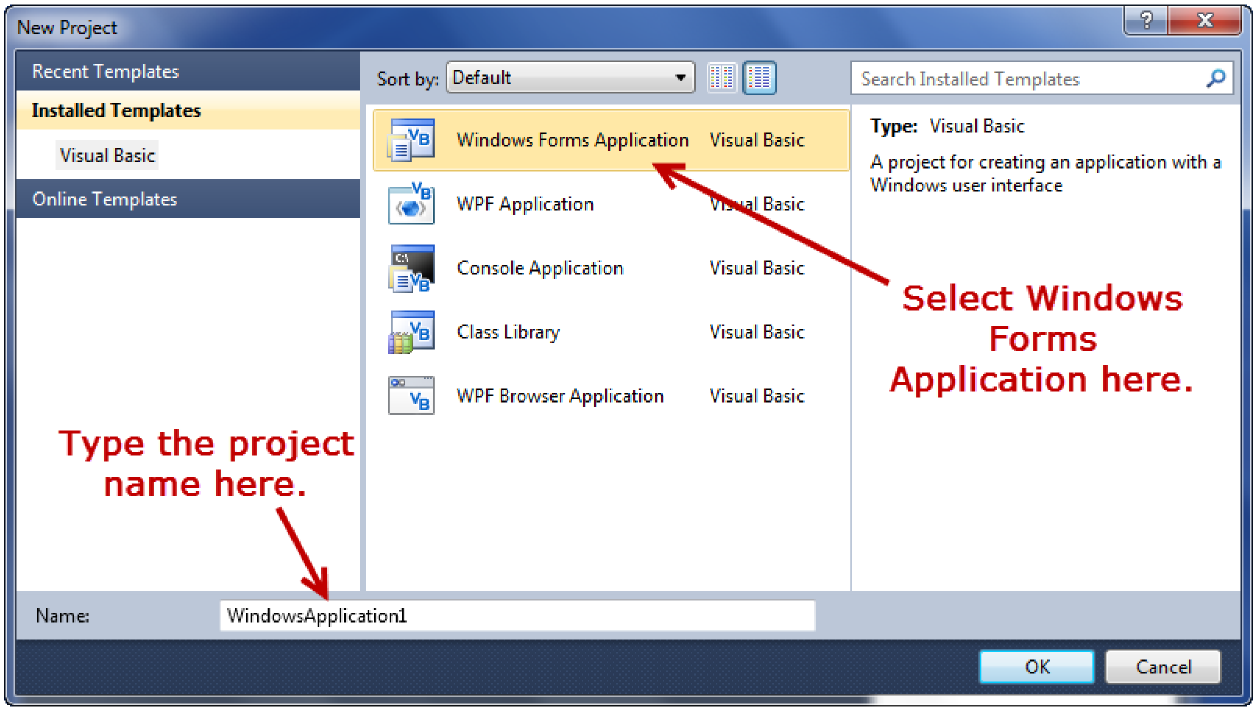
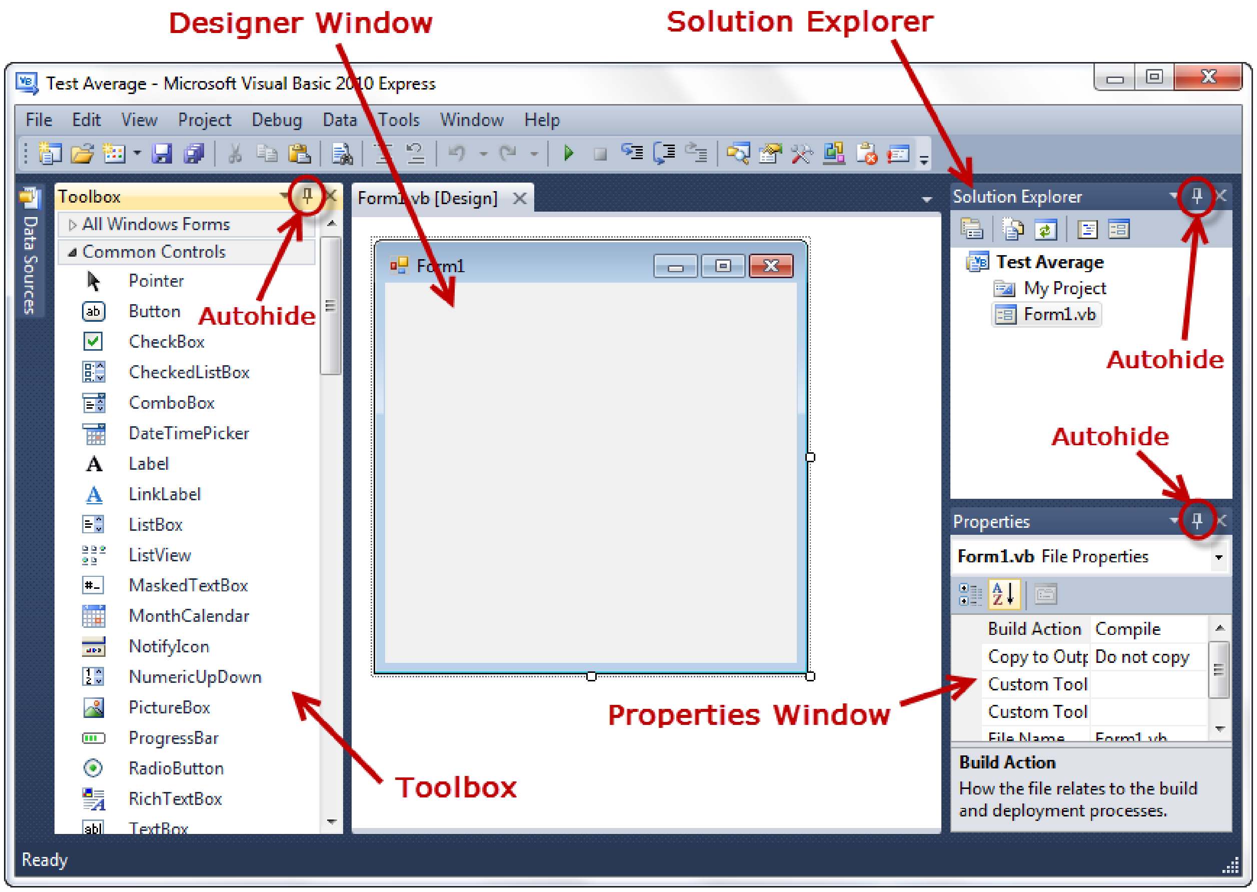

New Project
Recent Templates
Installed Templates
Visual Basic
Online Templates
Sort by: Default
Search Installed Templates
Windows Forms Application Visual Basic
WPF Application Visual Basic
Console Application Visual Basic
Class Library Visual Basic
WPF Browser Application Visual Basic
Type: Visual Basic
A project for creating an application with a Windows user interface
Select Windows Forms Application here.
Type the project name here.
Name: WindowsApplication1
OK
Cancel

Figure 15-2 The Visual Basic Environment



Step 4: In this step you will make sure your Visual Basic environment appears as that shown in Figure 15-2.

If you do not see the Toolbox window displayed as shown in Figure 15-2, perform the following:

- Click *View* on the menu bar, and then click *Toolbox*. This should cause the Toolbox window to appear.
- In the Toolbox window, locate the Autohide button, which appears as a pushpin icon (see Figure 15-2). Make sure the pushpin is pointing down, as shown in the figure. If it is not pointing down, click it. This will make the Toolbox window remain on the screen.

If you do not see the Solution Explorer window displayed as shown in Figure 15-2, perform the following:

- Click *View* on the menu bar, and then click *Solution Explorer*. This should cause the Solution Explorer window to appear.
- In the Solution Explorer window, locate the Autohide button, which appears as a pushpin icon (see Figure 15-2). Make sure the pushpin is pointing down, as shown in the figure. If it is not pointing down, click it. This will make the Toolbox window remain on the screen.

If you do not see the Properties window displayed as shown in Figure 15-2, perform the following:

- Click *View* on the menu bar, and then click *Properties Window*. This should cause the Properties window to appear.
- In the Properties window, locate the Autohide button, which appears as a pushpin icon (see Figure 15-2). Make sure the pushpin is pointing down, as shown in the figure. If it is not pointing down, click it. This will make the Toolbox window remain on the screen.

If you do not see the Designer window displayed with the blank form, as shown in Figure 15-2, perform the following:

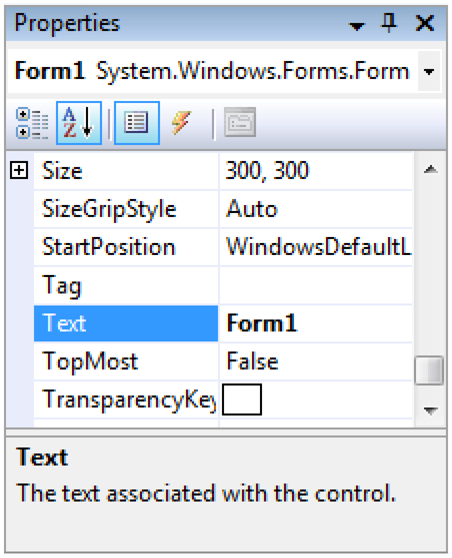
- Inside the Solution Explorer double-click *Form1.vb*. This should cause the Designer Explorer window to appear.

Step 5: In Chapter 15 you learned that a GUI component has properties that determines its appearance. In Visual Basic you can use the Properties window to change a component's property.

For example, a form (same thing as a window) has a *Text* property that determines the text displayed in the form's title bar. If you look at the Designer window right now, you should see that the form's title bar displays the text "Form1". If we want to display something else in the form's title bar, we need to change the form's *Text* property. Let's try it. Perform the following:

- Inside the Designer window, click the form to select it.
- Inside the Properties window, locate the *Text* property, as shown in Figure 15-3. (You might have to scroll down in the Properties window.)
- Notice that the property's value is currently set to *Form1*. To change this, simply double-click the property name, and then type *Test Average* as the property's new value.
- Press Enter and you should see the form's title bar appear as shown in Figure 15-4.

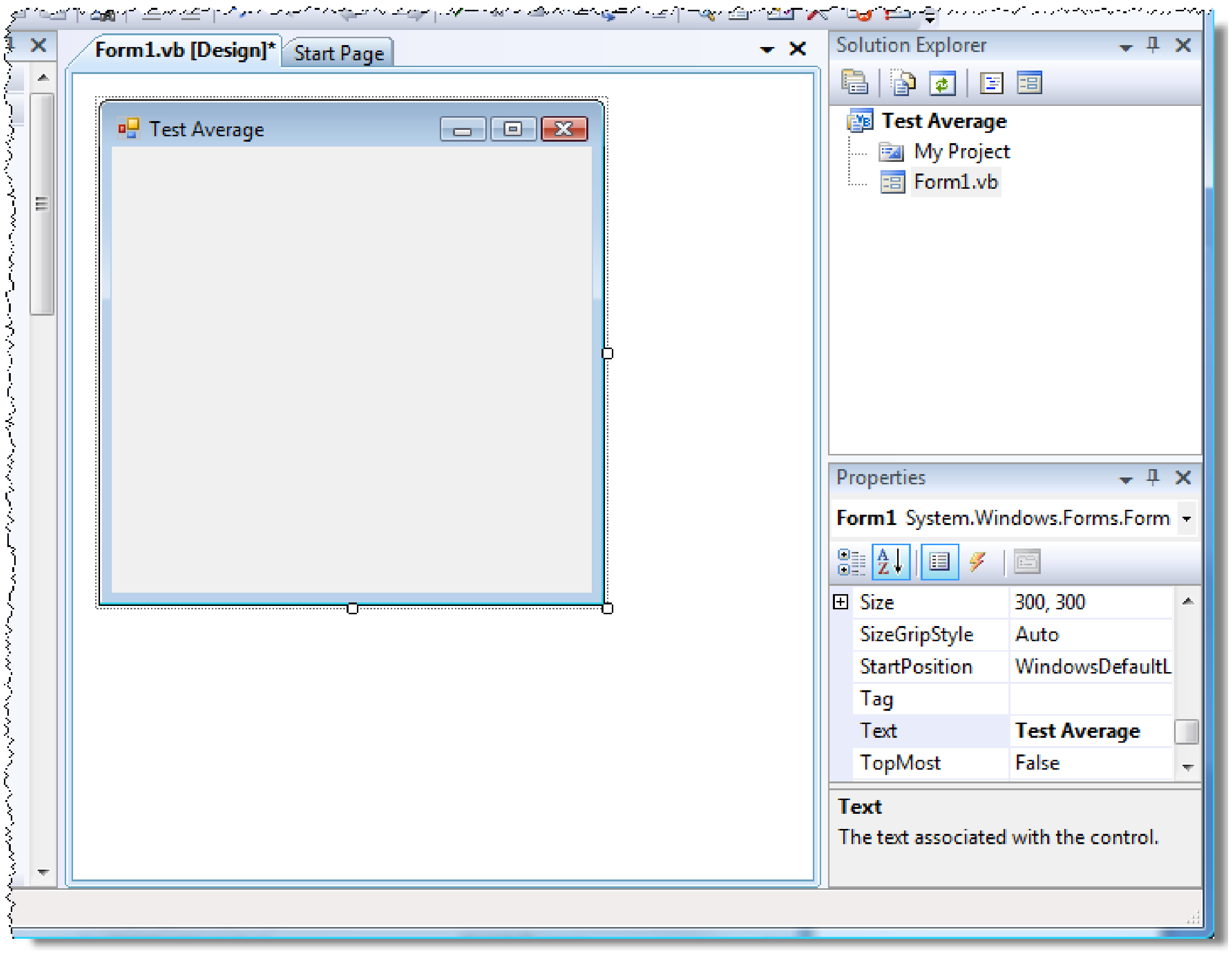
Figure 15-3 The Text property



Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultL
Tag	
Text	Form1
TopMost	False
TransparencyKey	

Text
The text associated with the control.

Figure 15-4 New text displayed in the form's title bar



Form1 System.Windows.Forms.Form

Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultL
Tag	
Text	Test Average
TopMost	False

Text
The text associated with the control.

Step 6: GUI components are commonly referred to as *controls* in Visual Basic. The Toolbox window shows all of the controls that you can place on a form. The controls are grouped into categories. Locate the *Common Controls* group, as shown in Figure 15-5. (You might have to scroll in the Toolbox.)

Notice that one of the items shown in this group is the Label. Double-click the Label control in the Toolbox window. This creates a new Label control on the form, as shown in Figure 16-6.

Figure 15-5 The Common Controls group

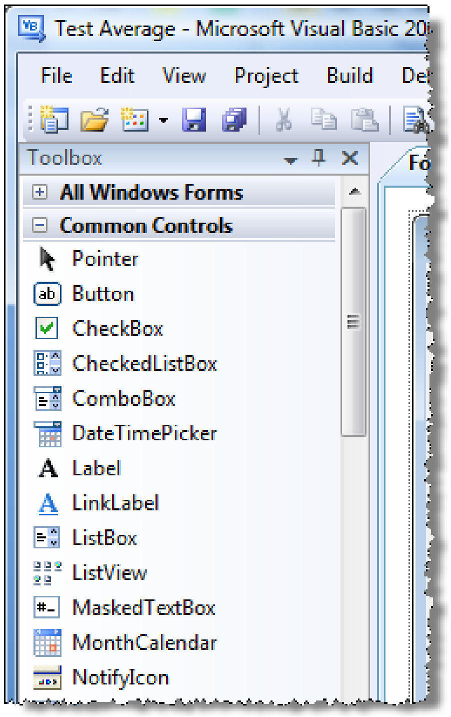
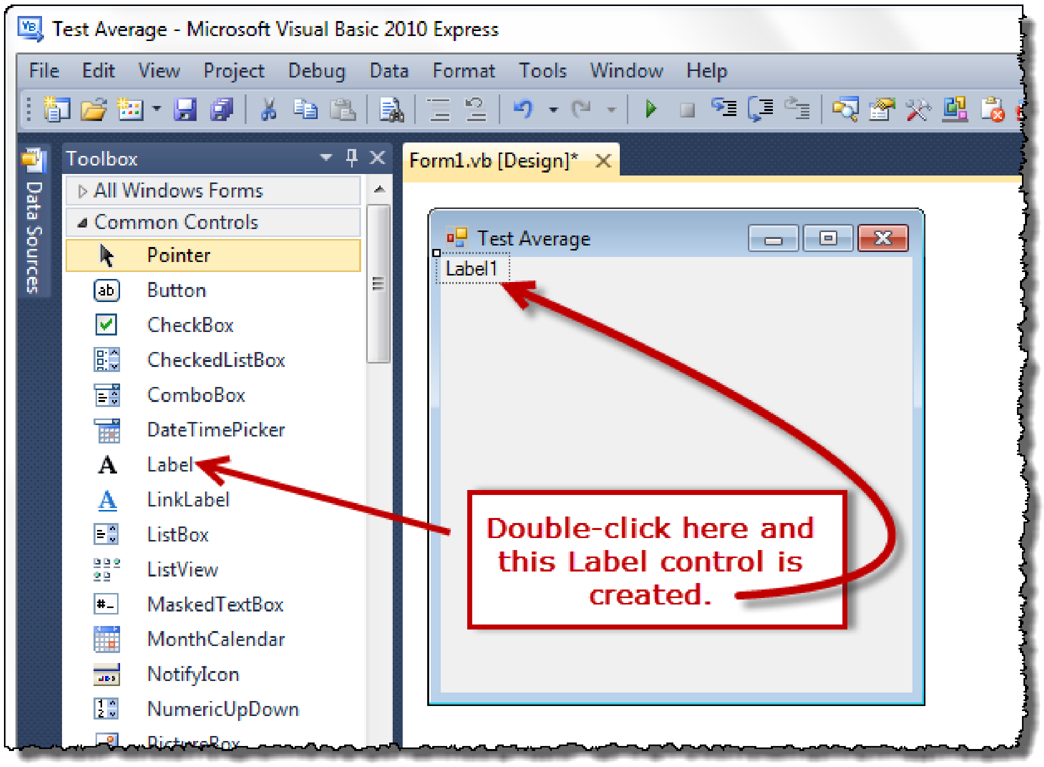


Figure 15-6 Creating a Label control



Step 7: Notice that a thin, dotted border appears around the Label control. This indicates that the Label control is selected. If you click anywhere in the Designer window outside the Label that you just placed, you will deselect the Label and the border will disappear. To select the Label again, just click it with the mouse and the border will reappear around it.

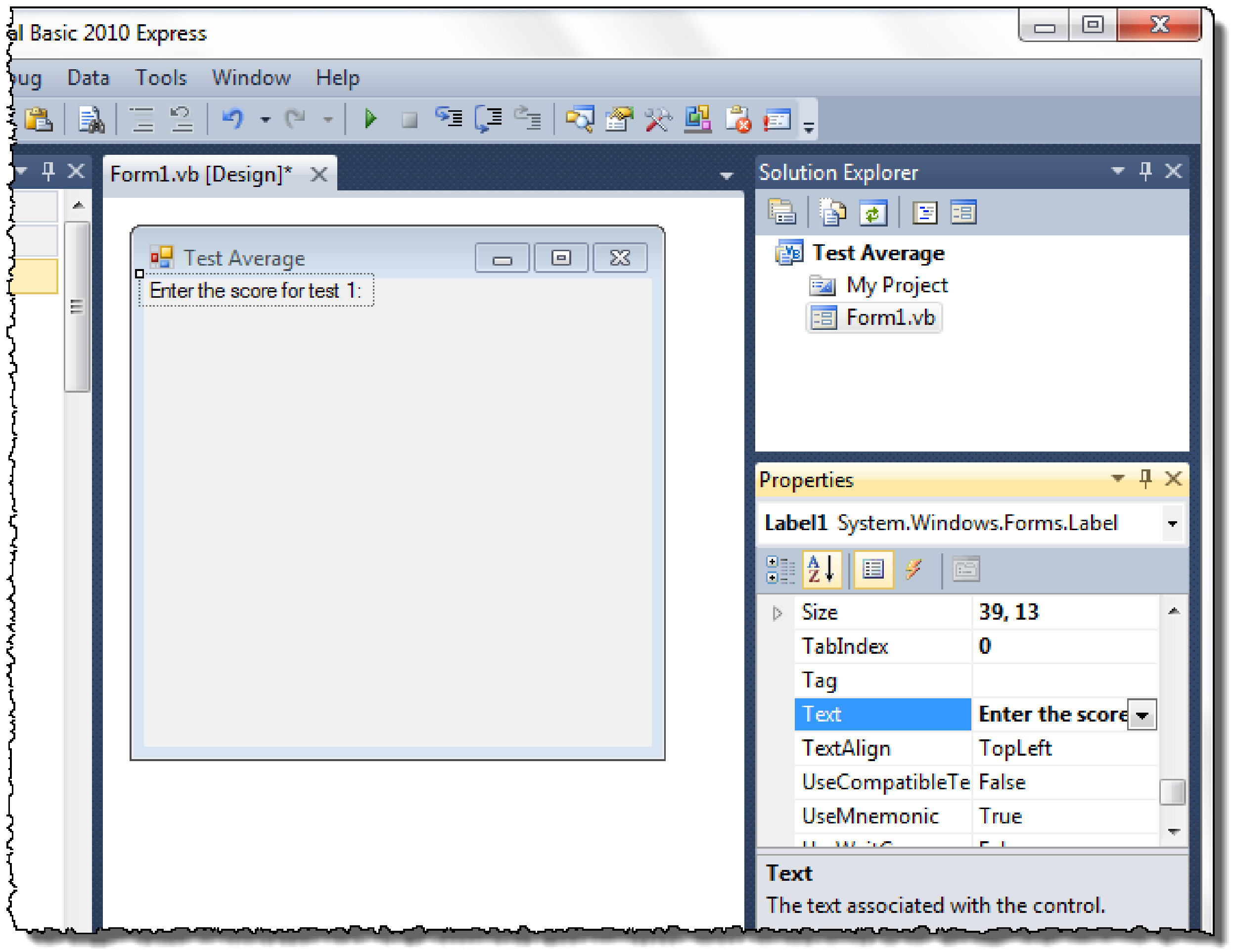
Now you will change the text displayed by the Label control that you just placed on the form. Make sure the Label control is selected, and then do the following:

- Inside the Properties window, locate the *Text* property.
- Notice that the property's value is currently set to *Label1*. To change this, simply double-click the property name, and then type *Enter the score for test 1:* as the property's new value.
- Press Enter and you should see the Label's appearance change as shown in Figure 15-7.

Now you will change the Label control's Name property. Recall from your textbook that a control's name identifies the control in your program's code. If you refer to Figure 15-11 on page 559 in your textbook, you will see that the Label control you just created should have the name *test1Label*. Make sure the Label control is selected, and then do the following:

- Inside the Properties window locate the *Name* property, as shown in Figure 15-8.
- Notice that the property's value is currently set to *Label1*. To change this, simply double-click the property, and then type *test1Label* as the property's new value.
- Press Enter.

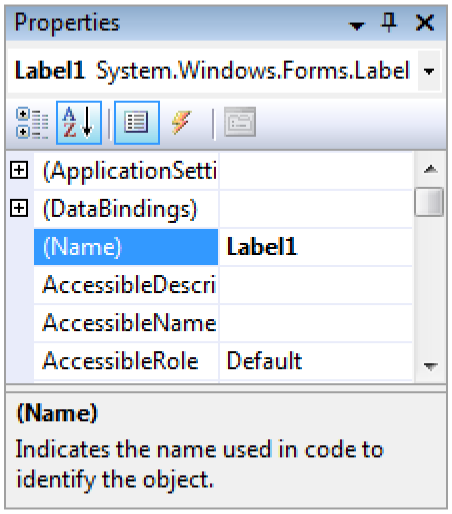
Figure 15-7 Label control's Text property changed



Size	39, 13
TabIndex	0
Tag	
Text	Enter the score
TextAlign	TopLeft
UseCompatibleTe	False
UseMnemonic	True

Text
The text associated with the control.

Figure 15-8 The Name property

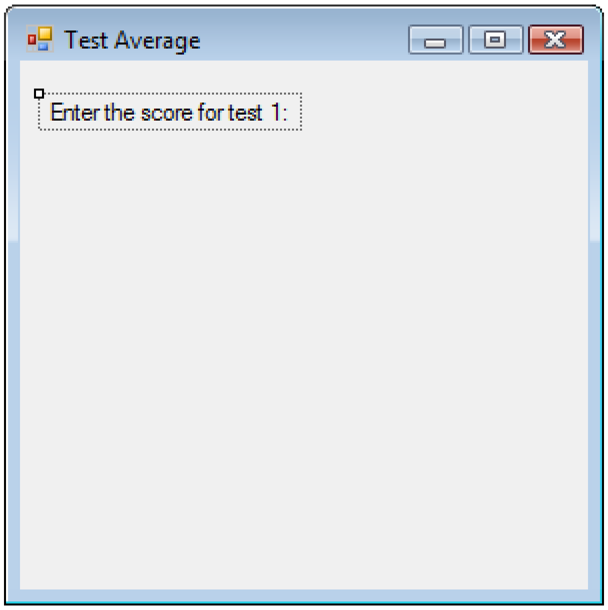


(ApplicationSetti	
(DataBindings)	
(Name)	Label1
AccessibleDescri	
AccessibleName	
AccessibleRole	Default

(Name)
Indicates the name used in code to identify the object.

Step 8: When a control is selected on a form, you can click and drag the control to reposition it. Make sure the Label control that you just placed on the form is selected. Use the mouse to drag the control just a bit to the right, and just a bit below its current location. The form should appear similar to Figure 15-9.

Figure 15-9 Label control repositioned



Step 9: Create another Label control on the form. Change its Text property to *Enter the score for test 2:* and change its Name property to *test2Label*.

Then, create another Label control on the form. Change its Text property to *Enter the score for test 3:* and change its Name property to *test3Label*.

Position the Label controls so the form appears similar to Figure 15-10.

Step 10: Now you will create a TextBox control on the form. Locate the TextBox control in the Toolbox window (in the Common Controls group) and double-click it. This creates a TextBox control on the form. Use the mouse to move the control to the approximate location shown in Figure 15-11.

With the TextBox control that you just placed still selected, use the Properties window to changes the control's Name property to *test1TextBox*.

Figure 15-10 Label controls created and positioned

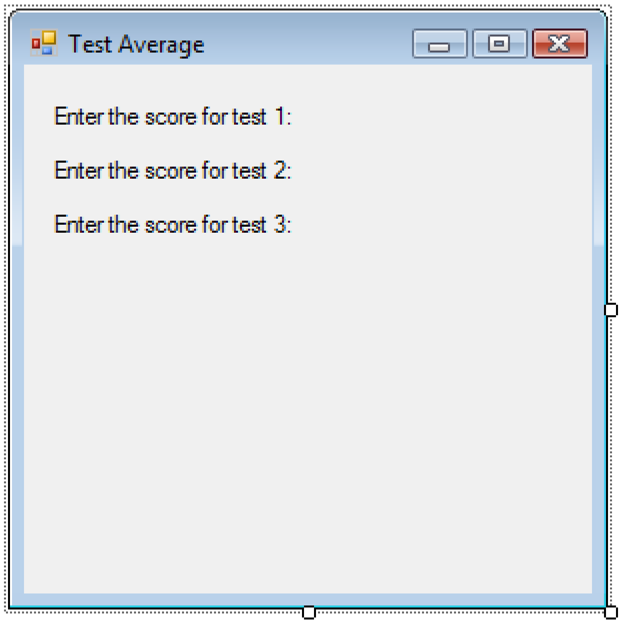
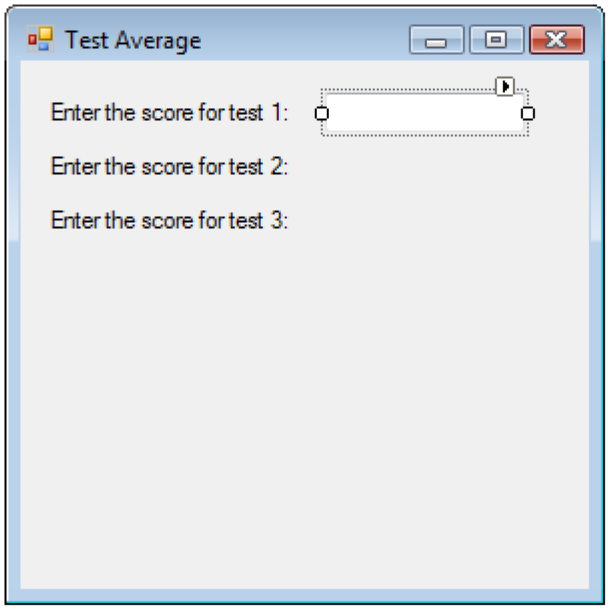

Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:

Figure 15-11 TextBox control created and positioned

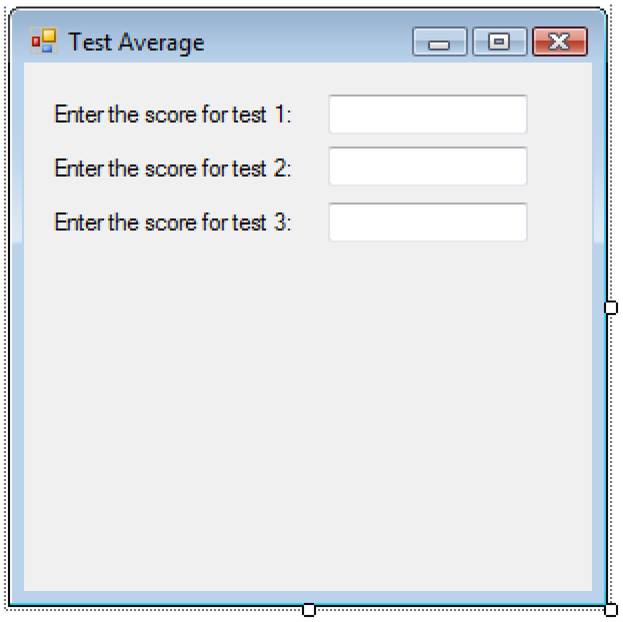

Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:

Step 11: Create another TextBox control on the form, and change its Name property to *test2TextBox*.

Then, create another TextBox control on the form, and change its Name property to *test3TextBox*.

Position the TextBox controls so the form appears similar to Figure 15-12.

Figure 15-12 TextBox controls created and positioned


Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:

Step 12: Create another Label control on the form. Change its Text property to *Average* and change its Name property to *resultLabel*. Position the control so the form appears similar to Figure 15-13.

Step 13: Create another Label control on the form and make the following property settings:

- Locate the control's BorderStyle property in the Properties window. Double-click the property and select *FixedSingle*.
- Locate the control's AutoSize property in the Properties window. Double-click the property and select *False*.
- Clear the contents of the control's Text property. (Simply double-click the Text property's current contents in the Properties window, and press the Delete key.)

Position the control so the form appears similar to Figure 15-14.

Figure 15-13 resultLabel created and positioned

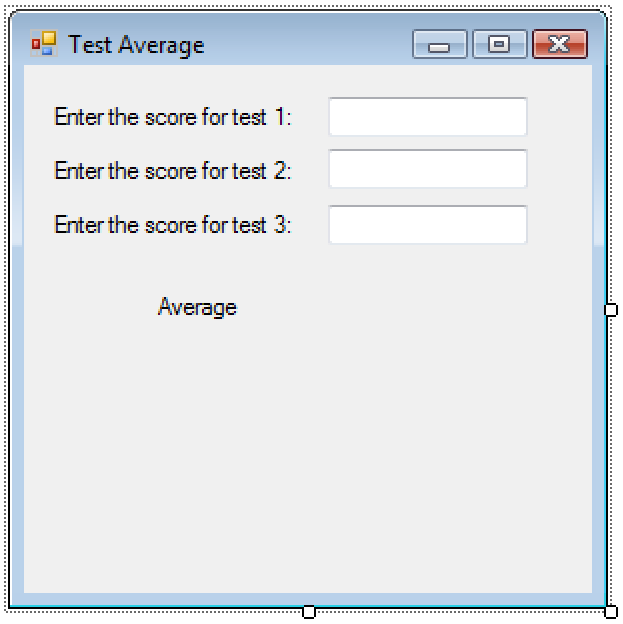
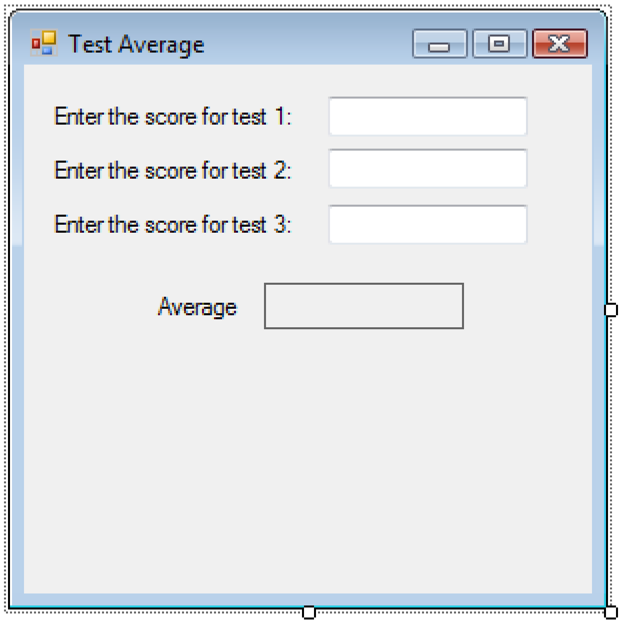

Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average

Figure 15-14 averageLabel created and positioned

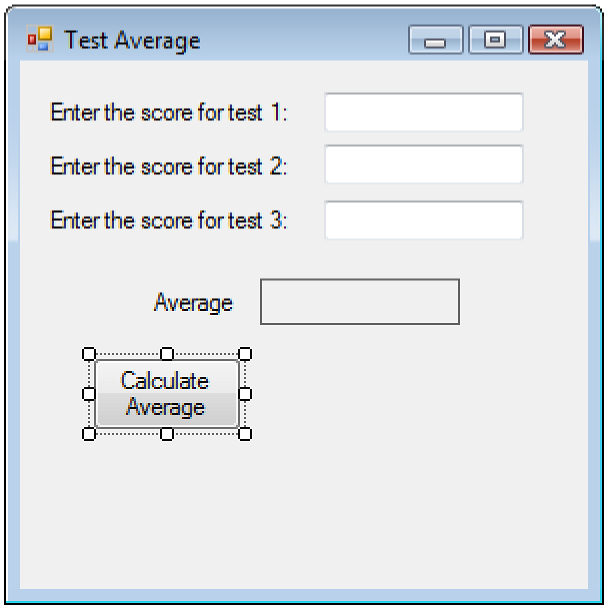

Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average

Step 14: Double-click the Button control in the Toolbox. (You will find it in the Common Controls group.) This creates a Button control on the form. With the Button control that you just created selected, make the following property settings:

- Change the Button's Name property to *calcButton*.
- Change the Button's Text property to *Calculate Average*.

With the Button control that you just created selected, you can click and drag the small squares that appear along its border to resize the control. Enlarge the button so you can see all of the text displayed on its face. Then, move the button so it is in the approximate location shown in Figure 15-15.

Figure 15-15 Calculate Average button created, resized, and positioned

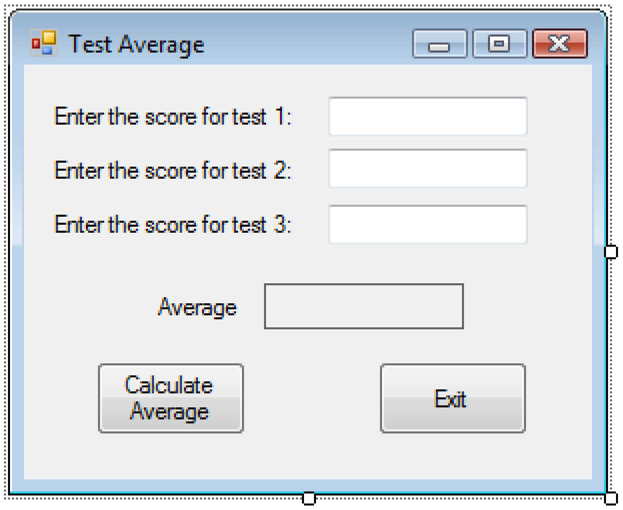

Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Calculate Average

Step 15: Create another Button control on the form and make the following property settings:

- Change the Button's Name property to *exitButton*.
- Change the Button's Text property to *Exit*.

Resize the button and move it so it is in the approximate location shown in Figure 15-16. You can also resize the form, as shown in the figure.

Figure 15-16 Exit button created, resized, and positioned


Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Calculate Average
Exit

Step 16: Now we will write an event handler for the *calcButton* control. The event handler will execute any time the button is clicked. Program 15-1 in your textbook (on page 563) shows the algorithm in pseudocode.

Double-click the *Calculate Average* button in the Designer window. This opens the code editor and creates the event handler code template shown in Figure 15-17. Complete the event handler by writing the code shown Figure 15-18.

Step 17: Now we will write an event handler for the *exitButton* control. The event handler, which will execute any time the button is clicked, will end the program.

First, switch back to the Designer window by clicking the tab that reads *Form1.vb [Design]* at the top of the code editor. The tab is shown in Figure 15-19. Then, double-click the Exit button in the Designer window. This creates a code template for the `exitButton_Click` event handler. Figure 15-20 shows how you are to complete the event handler. Inside the code template, simply write this statement:

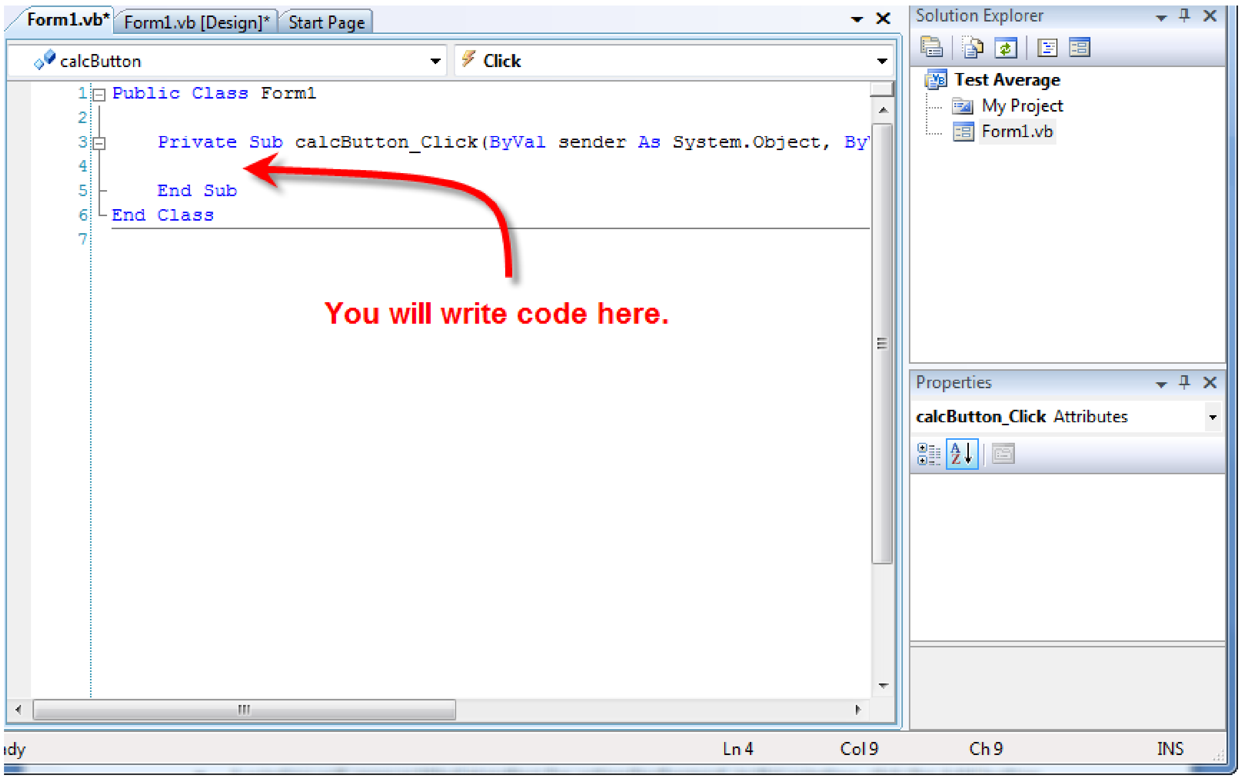
```
Me.Close()
```

Step 18: Press Ctrl+Shift+S on the keyboard to save the project.

Step 17: Press F5 on the keyboard to run the program. If you've typed the code correctly, the application's form should appear as shown in Figure 15-21. Enter some valid numeric test scores into the TextBox controls and then press the Calculate

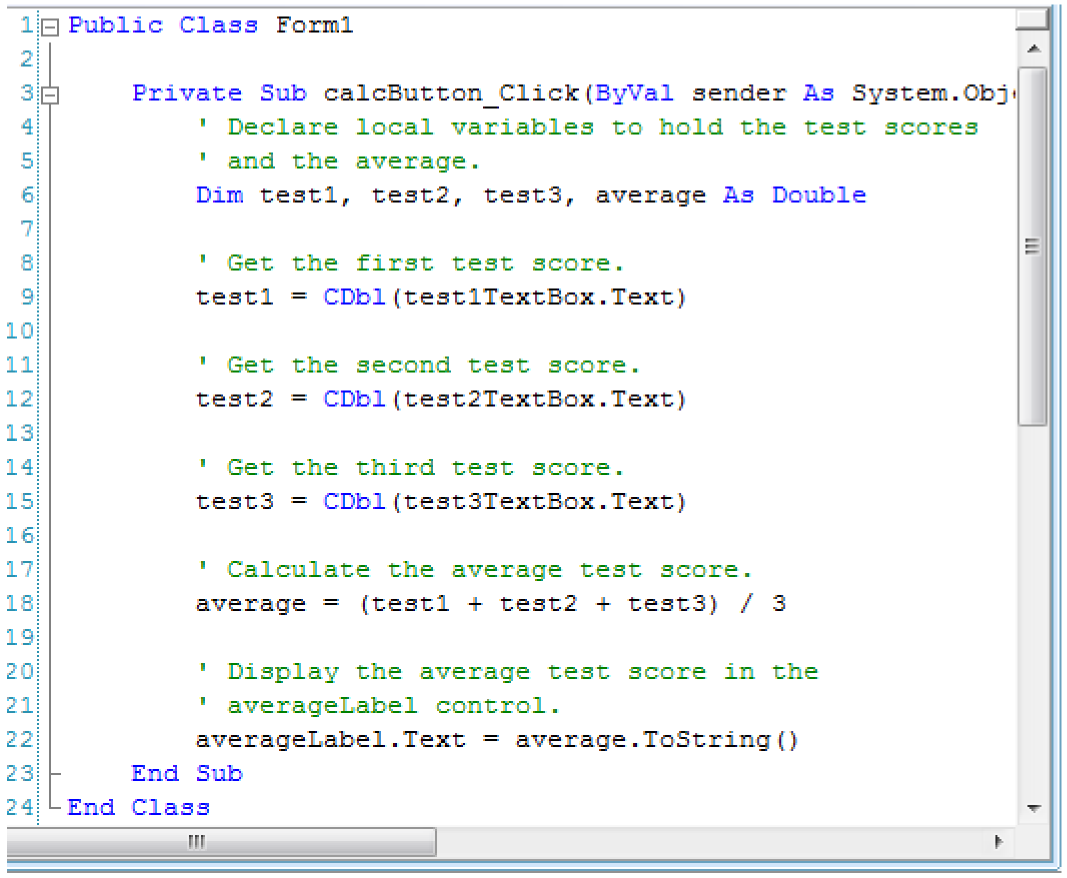
Average button to see the average displayed. When you are finished, click the Exit button to end the program.

Figure 15-17 Code template for the `calcButton_Click` event handler



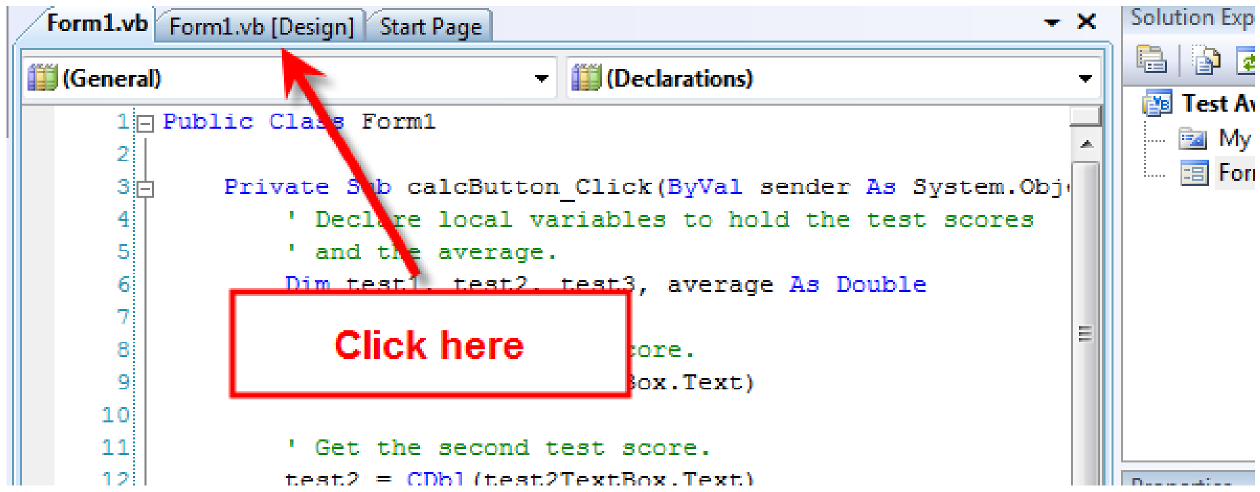
```
Public Class Form1

    Private Sub calcButton_Click(ByVal sender As System.Object, By

    End Sub
End Class
```

Figure 15-18 Completed code for the `calcButton_Click` event handler



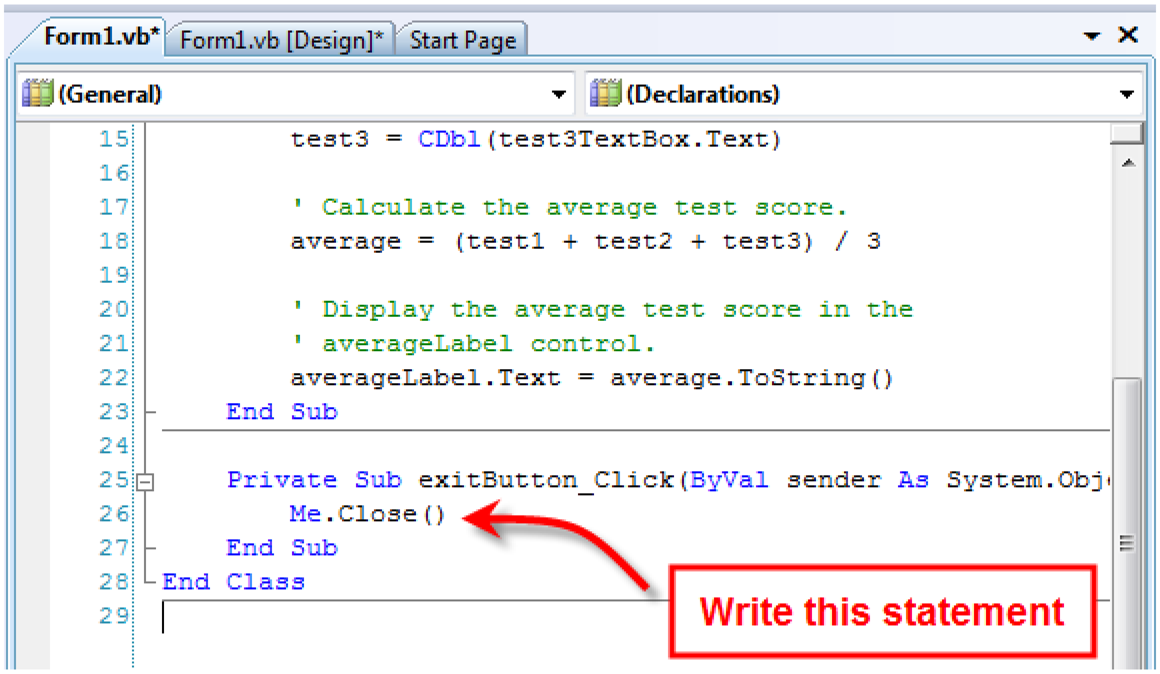
```
Public Class Form1

    Private Sub calcButton_Click(ByVal sender As System.Obj
        ' Declare local variables to hold the test scores
        ' and the average.
        Dim test1, test2, test3, average As Double

        ' Get the first test score.
        test1 = CDbl(test1TextBox.Text)

        ' Get the second test score.
        test2 = CDbl(test2TextBox.Text)

        ' Get the third test score.
        test3 = CDbl(test3TextBox.Text)

        ' Calculate the average test score.
        average = (test1 + test2 + test3) / 3

        ' Display the average test score in the
        ' averageLabel control.
        averageLabel.Text = average.ToString()
    End Sub
End Class
```

Figure 15-19 Using the Form1.vb [Design] tab to switch to the Designer window



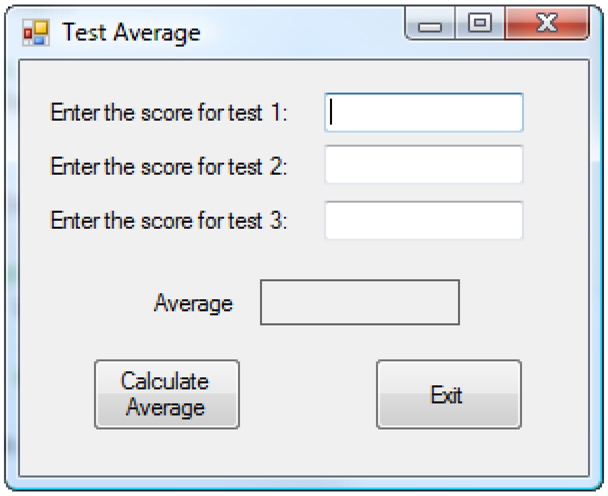
```
Public Class Form1

    Private Sub calcButton_Click(ByVal sender As System.Obj
        ' Declare local variables to hold the test scores
        ' and the average.
        Dim test1, test2, test3, average As Double

        ' Get the first test score.
        test1 = CDbl(test1TextBox.Text)

        ' Get the second test score.
        test2 = CDbl(test2TextBox.Text)
```

Figure 15-20 Completed code for the `calcButton_Click` event handler



```
            test3 = CDbl(test3TextBox.Text)

            ' Calculate the average test score.
            average = (test1 + test2 + test3) / 3

            ' Display the average test score in the
            ' averageLabel control.
            averageLabel.Text = average.ToString()
        End Sub

        Private Sub exitButton_Click(ByVal sender As System.Obj
            Me.Close()
        End Sub
    End Class
```

Figure 15-21 The Test Average program running



Test Average
Enter the score for test 1:
Enter the score for test 2:
Enter the score for test 3:
Average
Calculate Average
Exit