Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

CHAPTER 4

Data, Operations, and Built-In Functions

So far, you have learned more than a dozen Visual Basic (VB) controls, and have explored their important features. These controls present themselves as great tools for building an elegant graphic user interface. Data obtained from the interface, however, usually need to be further processed before they become truly useful. In many cases, complex computations must also be performed; the results are then displayed. In this chapter, you will study various aspects of data and the operations/computations that the computer can perform on them. After you complete this chapter, you should be able to:

- Differentiate among different types of data,
- Declare data types properly for constants and variables,
- Explain the importance of having the IDE check variables for their declaration,
- Articulate the implications of the scope and lifetime of variables under different declarations,
- Explain the nature of the assignment statement,
- Use various arithmetic expressions and functions,
- Use various string functions, and
- Explain how data and VB controls are used together in a project.

4.1 Classification and Declaration of Data

Consider the following code:

```
MsgBox("Your age is " & 24)
```

As simple as this statement, two different types of data are involved. The text enclosed in a pair of quotation marks is a *string*, whereas the number, 24, is *numeric*. A string consists of zero or more characters; numeric data are numbers that can be used for various computations, and can be further classified into many different types.

All data must be stored somewhere in the computer memory before they can be retrieved for additional manipulation or display. If a memory location with some data is expected to change as a result of operations, it is recognized as a *variable*. A variable must be given a name so that you can refer to it. Data that present themselves "as is" and are never expected to change are recognized as *constants*. A constant can go without a name but can also be given a name. A constant without a name has to be presented and used as literal. A constant that is given a name and is referenced accordingly is recognized as a *named constant*.

To summarize, data can be classified by type into numeric and string, and by variability into constant and variable. The following table shows the cross-classifications:

Classification by: Variability/Data Type	**Numeric**	**String**	**(Name Required?)**
Constant	Numeric constant	String constant	Optional
Variable	Numeric variable	String variable	Required

Declaring Constants

Before a named constant can be used in your program, you must make sure that VB can recognize it. Some named constants are predefined and automatically recognized. These are called *system constants*. You have seen some of these constants in the preceding chapters. For example, you have seen named constants for colors such as Color.Green, Color.Blue, and Color.Red. You have also seen true or false named constants—True and False. Other named constants must be declared before VB can recognize them. These are called *symbolic constants*. You use the *Const statement* to declare a symbolic constant. The syntax is as follows:

```
Const name [As data type] = literal
```

For example, the following statement declares the named constant Zero to have a constant value of 0. As the syntax in the preceding example suggests, you can also omit As Integer from the declaration without affecting the result.

```
Const Zero As Integer = 0
```

Why Name Constants?

Properly named constants can enhance the understandability of the code. When reviewing code, you may not recognize the specific meaning of a particular value, such as 0 or -1. This is particularly true when the constants are used as the property setting of a VB control. Instead, if you code these constants as False (for 0) and True (for -1) in setting the Checked property of a check box, anyone, including yourself, should be able to follow the code much more easily. In addition, in some cases, you may find it necessary to later change the value of a constant. Using a named constant, you need to make the correction in only one place—where the constant is declared. In contrast, you will have to search the entire program for the constant to change if the constant literal was used.

Declaring Variables

Under certain option settings in your project, variables can be used without being declared. It is a good habit, however, to declare all variables used. The reasons for this are given later in the section, "Why Force Yourself into Declaring All Variables?" To declare a variable, you can use the *Dim statement*. The syntax is as follows:

```
Dim Variable1 [As data type][, Variable2 [As data type]]...
```

For example, you can code the following:

```
Dim TheName as String, SSN as Integer
Dim Rate, PresentValue As Double
```

The first statement declares a variable, TheName, as the String type variable, and another variable named SSN as an Integer type. The second statement declares both the variables, Rate and PresentValue, as the Double type. (Numeric data types are explained in Section 4.3, "Numeric Data and Types ," and the String type is discussed in Section 4.5, "String Data .")

Note that for code clarity, it is advisable to declare only one variable per line. (Many companies have adopted this coding standard.) Thus, a better way to code the preceding first line is as follows:

```
Dim TheName As String
Dim SSN As Integer
```

Initializing Value with Declaration.

When you declare only a variable per line, you can also assign an initial value to the variable as shown in the following example:

```
Dim TheName As String = "John Doe"
```

In this line, the variable TheName is declared as a String type, and is given "John Doe" as its initial value. A string variable without an initial value is initialized has a zero length string.

Using Type Suffixes

A variable or symbolic constant can also be declared with a trailing special character that signifies a data type. These special characters, such as $, %, and &, are called *type-declaration characters* or *type suffixes*, and are

listed in Section 4.3. The following statement with type-declaration characters will have the same effect as the previous example.

```
Dim TheName$
Dim SSN%
```

Recall in Chapter 2, "Introduction to Visual Basic Programming," the importance of using meaningful names for VB objects was emphasized. This observation can certainly be extended and applied to all types of names, including constants and variables. As you can see, declarations using type suffixes are not as easy to understand as those using data type names. For code clarity, always use data type names instead of type suffixes.

Tip

For program readability, always give your variables and constants meaningful names. Use the data type names instead of type suffixes to declare the data types for the variables and constants.

Checking for Variable Declarations

You have control over having VB check if the variables used in your program have been declared properly by the use of the Option Explicit statement, which has the following syntax:

```
Option Explicit {On|Off}
```

The statement must be placed before any other statement in the code window, except for other Option statements to be discussed later in this chapter. If you code,

```
Option Explicit Off
```

VB will not check whether any variable you use has been declared. The default—is Option Explicit On. It is strongly recommended that you use the default. With the default, the compiler immediately underlines any variable that is not declared.

Try This

Create a new project. Place the following code in the form Load event.

```
A = B
```

You will immediately see that both A and B are underlined. Rest the mouse pointer on the underline to see the IntelliSense indicating that the name is not declared.

Why Force Yourself into Declaring All Variables?

The advantages of declaring all variables include the following.

- A variable not declared may give you unexpected results. This kind of error is hard to uncover.
- This practice helps identify misspelled variables that can cause unexpected results.
- Declaring all variables can help you catch syntax errors that you unexpectedly make. For example, suppose you mean to assign a string "Ben" to the variable, TheName, but fail to enclose Ben in a pair of double quotes. Your statement will look like: TheName = Ben. With Option Explicit On, the editor will tell you the variable Ben has not been declared. With the option off, both variables will have a zero length string.
- If you declare a variable with proper capitalization, the same capitalization will be maintained automatically by VB throughout the program. For example, if you declare a variable as HourPay, VB will change it in your program to HourPay regardless of how you type the variable, such as hourpay or

HOURPAY. This enables you to immediately check if you have typed in a name properly. Proper capitalization also enhances the readability of your program.

Ensuring Automatic "Option Explicit On" in Your Project

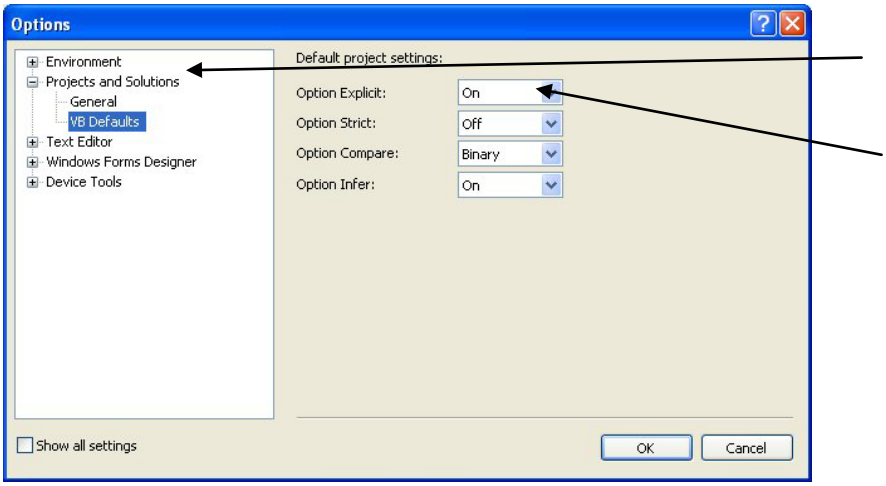
What if the compiler does not enforce variable declaration in your project? As stated previously, you can code the Option Explicit On statement in the code window, which is recognized as a module. The statement will be in effect for that particular module. If your project has more than one module, you will need to code the statement in each module. You can actually verify and opt to have the compiler enforce variable declaration for the entire project without any code. The settings for several Options are accessible in the Options dialog box through the Tools menu. To access the Option's setting, follow these steps:

1. Click the Tools menu in the IDE.
2. Click the Options option. The Options dialog box appears.
3. Select the Projects and Solutions option in the left frame.
4. Click VB Defaults under the Projects and Solutions option.
5. Select On in the combo box for Option Explicit, as shown in Figure 4-1.

Tip

Experienced programmers know the importance of having all the variable names spelled correctly. A minor error in spelling a variable name can result in mysterious errors in the program and cause numerous hours of hunting for the bug. Always ensure that Option Explicit On is the setting for your project.

Figure 4-1
Setting Option Explicit On



Rules for Variable Declaration

When you are declaring variables or named constants, some rules must be observed. Any violation of the rules will result in a syntax error. These rules include:

- A variable or constant name can contain any combination of letters, numbers and underscore; must begin with a letter or underscore; and must have at least one letter or digit if it begins with an underscore.
- If the variable is a reserved keyword, it must be enclosed with a pair of brackets, such as [Case]. (Case is a reserved keyword. You will learn about it in Chapter 5, "Decision.")

- A variable must not contain any embedded period, such as Your.Name, or embedded special characters used for data type declaration, such as The%Completed.

4.2 Scope and Lifetime of Variables

The code window associated with the form is recognized as the *class module*. When you first start a new project, its code window should contain two statements at follows:

```
Public Class Form1

End Class
```

These two statements define a class, which is the template for an object. All statements pertaining to the class must be placed between these two statements. Classes will be discussed in detail in Chapter 12, "Object Based Programming," and Chapter 13, "Object Oriented Programming." Most of the code that you write for the module should be placed between these two statements. (Option statements such as Option Explicit should be placed at the beginning before the Class statement.) All the event procedures that you have coded have been placed in this area. If you have any variable declaration statements to be placed outside of any procedure, you should place them in this area first, before any code for any procedure. This area shall be called the *general declaration area*.

Variables and constants have their scope and lifetime (duration). *Scope* refers to how widely a variable is recognized (accessible); *lifetime* refers to how long a variable remains in computer memory. The placement of a variable declaration can affect both the scope and the duration of that variable. The code snip below shows different levels of declarations, each is further explained in the following subsections.

```
Public Class Form1
    Dim ClassLevelVar As Integer 'Class level declaration
    Private Sub btnTest1_Click(ByVal Sender As System.Object, ByVal e As
      System.EventArgs) Handles btnTest1.Click
        Dim ProcLevelVar1 as Integer 'Procedure level declaration
        ' additional code lines
        If chkBreakFast.Checked Then
            Dim BlockLevelVar1 As Integer 'Block level declaration
            'Other statements
        End If
    End Sub
End Class
```

Form (Class)-Level Declaration

Variables and constants declared with a Dim statement in the general declaration area are the *class level* variables and constants. They are recognized by and accessible to all procedures in the class. In addition, they exist as long as the class does; that is, their values are preserved until the form is destroyed. In a single form application, the form is destroyed when the project ends. The life of a form is discussed in more detail in Chapter 11, "Menus and Multiple-Form Applications."

Procedure-Level Declaration

Variables and constants declared in a procedure (within the Sub … End Sub structure) are recognizable only in that procedure. They are said to be the *procedure level* or *local* variables and constants. The same names used in other procedures refer to different memory locations, and have nothing to do with those in the current

procedure. Because these variables are independent of each other, you can declare different data types in different procedures using the same name. For example, it is legitimate to have the following declarations for I:

```
Private Sub btnTest1_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest1.Click
    Dim I as Integer
   ‘ additional code lines
End Sub

Private Sub btnTest2_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest2.Click
    Dim I as Long
   ‘ additional code lines
End Sub
```

Such a practice creates confusion, however, even for yourself when you review the code later. Avoid this practice by all means.

Lifetime of Variables Declared in a Procedure with Dim

Variables declared with a Dim statement in a procedure exist as long as the procedure is in action. When the procedure ends, these variables are said to be out of scope and disappear. When the procedure is called again, these variables are reinitialized. They no longer have their previous values. They will be reset to zero if they are numeric variables, or to a zero-length string if they are string variables.

Static Declaration

If you want a local variable to preserve its value for the duration of the form, not just of the procedure, you can use the Static statement for the declaration. When you declare a variable with the *Static statement,* the value of that variable will be preserved and will not be reinitialized between each call of the procedure. The following example illustrates the difference between a variable declared with a Dim statement and one with a Static statement:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
 System.EventArgs) Handles MyBase.Click
    Dim I as Integer
    Static J as Integer
    I = I + 1
    J = J + 1
    MsgBox( “I = “ & I & “. J = “ & J)
End Sub
```

When you run this program and then click on the form repetitively, you will notice that the message box continues to display 1 for I, but increases the value for J each time.

Scope of Static Variables

If a static variable lives as long as the form, how then is it different from a class level variable? They differ in scope. A static variable, which can be declared only inside a procedure, is a local variable, recognized only in the procedure. On the other hand, a class level variable is recognized by all procedures in the same form. For example, suppose you have a form with a button named btnTest. Consider the following code:

```
Dim K as Integer ‘Place this line right after the Class statement
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
```

```
Static J as Integer
K = K + 1
J = J + 1
MsgBox( “K = “ & K & “. J = “ & J)
End Sub
```

You should be able to enter and run the routine without any problem. If you then enter the following code, however, you will notice that although you can enter the first MsgBox statement without any problem, the second MsgBox will have its variable J underlined. The compiler will tell you that J is not declared because a Static variable in another procedure is local to that procedure, and is not recognized in any other procedure.
<<Insert x mark with this code>>

```
Private Sub btnTest_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
MsgBox(“K = “ & K)
MsgBox (“J = ” & J)
End Sub
```

Block Level Declaration

Procedure level declarations can be placed anywhere in the procedure before the variables in question are used. If you declare a variable within an If block, however, the variable will be recognized only within the block. After the execution leaves the block, the variable is out of scope. If you subsequently attempt to use it, the compiler will tell you that it is not declared.

Overlapping Declarations in Form and Procedure

What happens when a class level variable is also declared in a procedure? The one declared in the procedure is a different variable, a local variable with the same name. VB recognizes the variable of the narrowest scope. This procedure recognizes the local variable, not the class level variable. To reference the variable declared at the class level, qualify the variable with Me. The following code example shows how the variable I declared both at the class level and a procedure can be referenced in the procedure:

```
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    Dim I As Integer
    I = 3
    Me.I = 12
    MsgBox(I & ", " & Me.I)
End Sub
```

Run the project. You should see that the procedure displays 3 and 12.

Note, however, that you cannot have overlapping declarations in procedures and blocks; if you declare a variable in a procedure, you cannot declare another variable with the same name inside a block, such as an If block, in that procedure.

Scope and Lifetime: A Recap

To recapitulate, variables have different scopes and lifetimes, depending on where and how they are declared. A variable declared at the form (class) level is recognized by all procedures; a variable declared in a procedure is recognized only in that procedure. If the same variable name is declared in different contexts (in the form and/or in different procedures), the variable declared in the procedure is the one recognized inside the procedure.

Besides its scope, a variable also has its lifetime. A form level variable has its value preserved until the form is destroyed. The value of a procedure level variable is reinitialized each time the procedure is called if the variable is declared with a Dim statement. If it is declared with a Static statement, however, the procedure level variable's value is preserved until the form is destroyed.
The following table provides a summary of the scope and duration of variables.

Declaration	Scope	Duration
Dim A As Integer (in the general declaration area)	Form (class) level variable recognized by all procedures	The value is preserved for as long as the form exists.
Dim B As Integer (inside a procedure)	Procedure level (local) variable recognized only in that procedure	As soon as the procedure ends, the value is gone.
Dim C As Integer (inside a block)	Block level variable recognized only in that block	As soon as the execution control leaves the block, the name is not recognized.
Static D As Integer (allowed only inside a procedure)	Procedure level (local) variable recognized only in that procedure	The value is preserved for as long as the form exists.

Declaring Constants

The value of named constants cannot be changed. Although they have similar scope and lifetime as variables, it would seem more appropriate to use the same names for the same types and values throughout the project. One potential problem is that broad scope constants declared at different places in a project may result in name collision; same names have different values and types. To guard against such a problem, declare all constants in one place in your program. This practice enhances the consistency and maintainability for the project.

Additional Note on Declaration

The preceding discussion on declaring variables and constants dealt with projects with only one form. Will there be any difference in declaration when a project has more than one form? Yes. In this case, you may need to consider whether a variable declared in one module *at the class level* (local variables are local, anyway) should be available (accessible) to the other modules. Three commonly used access modifiers (modifiers that define the scope) are ***Public, Friend*** and ***Private***. You can declare variables as follows:

```
Public Dim YourVariable As Double
Friend Dim HisVariable As Long
Private Dim MyVariable As Single
```

When a variable is declared with an access modifier, you can omit the keyword Dim. Typically, the declarations appear as follows:

```
Public YourVariable As Double
Friend HisVariable As Long
Private MyVariable As Single
```

A variable declared as Public is accessible to all other modules in the solution; a variable declared as Friend is accessible to all other modules in the project; a variable declared as Private is recognized only in that class. So is a variable declared with a Dim statement at the class level. The following two declarations have the same effect:

```
Dim MyVariable As Integer
Private MyVariable As Integer
```

Variable accessibility will be discussed in more detail in Chapter 11, "Menus and Multiple-Form Applications," which discusses multiple form projects.

The discussion of declaration of variables so far has focused on their use for elementary data types. Variables can be (and are) also used to reference other types such as enumerations (groupings of constant names, discussed in Chapter 12, "Object Based Programming") and objects such as controls. In the case of elementary data types, structures, and enumerations, variables are ready to be used after they are declared. In the case of objects, however, it usually takes two steps: one step to declare the variable as the type, and another step to create (instantiate) and associate the object with the variable. As an example, consider the case of the Random object (an object that you can use in place of the Rnd function). To use it, you will first declare a variable of the Random type as follows:

```
Dim R As Random
```

In the second step, you will create the Random object with the New keyword, and assign it to the variable R as shown below:

```
R = New Random()
```

From that point on, you will be able to use the variable R to generate random numbers that suit your need. (Note that Random is followed by a pair of parentheses.)

Understanding Objects

Technically, Random as a type is similar to an icon in the Toolbox, such as the Text Box icon. The icon in the Toolbox is not yet an object. It is simply a template from which you can create instances of objects. For examples, you can draw many text boxes onto a form, creating many instances of text boxes from the template. The formal term for the template is *class*; for the instance(s), *object*. The action of creating an object (instance) from the class is termed *instantiation.* Use these terminologies to restate the two preceding steps: You first declared the variable R as the Random class; you then created an instance of the Random class and associated it with R.

In many cases, the two steps are combined into one—that is, the object is created and associated with the variable at the same time when the variable is declared. Again, the New keyword is used to create the object. The following statement accomplishes both steps:

```
Dim R As New Random()
```

The use of the Random object will be discussed later in this chapter. This discussion of declaration of variables (and the terms, object and class) may not appear very clear to you. This is natural because of the huge body of knowledge in VB that you need to absorb in a short period. You may want to come back and revisit this section a few more times to gain additional familiarity and understanding.

Scope and Program Modularity

As you can see from the preceding discussion, a variable declared in the general declaration area is recognized by all procedures in the class. This also means that only one memory location is used to handle the variable. You may be tempted to declare all variables at the class level for convenience and to conserve memory; however, you should avoid this programming practice.

When a variable is recognized by all procedures, it is shared by all procedures. When the value of this variable is changed in one procedure, the result affects *all* procedures. In some cases (and too often), such a change may not be expected when you write your code. This coding practice makes the resulting program difficult to maintain. In addition, because of the duration of the class level variables, the memory use is not necessarily minimized; therefore, the general rule is to declare each variable with a scope as narrow as

possible—that is, to the extent possible, declare the variables you need to use in the procedures where you need them and declare variables at the class level only if they absolutely have to be shared by more than one procedure. By declaring variables with the narrowest scope possible, you keep procedures independent of each other, which is the basic foundation of modular design.

Tip

When you absolutely need for the procedures to share certain variables, try to design your code such that each of these variables is assigned/changed in as few procedures as possible. This makes it easier to trace the source of the change when there is a problem. If the value of a variable needs to be changed to perform some operations within a procedure, and such a change is not intended for all other procedures, you should take care to restore the original value of the variable. This can be done by the use of a temporary variable, for example, a properly declared variable named TempVar, as shown in the following code fragment:

```
TempVar = YourVariable 'Save the original value
YourVariable = New Value 'Change the variable to a new value
' Your code that uses the New value
YourVariable = TempVar 'Restore the value at the end
```

Constant and Variable Naming Convention

Some companies also adopt naming conventions for constants and variables in addition to those that they use for controls. The rules for variables typically stipulate that there should be a one-character prefix for the scope, followed by another three-character prefix for the data type. For example, a module level integer variable can have a prefix mint, where m indicates the scope (module level) and int represents its data type. This naming notation is recognized as the Hungarian notation. The purpose of such rules is to enhance the readability of the code.

Some well-known authors, however, maintain that such a practice appears to be overkill, and can make the code extra long by adding an overhead to the coding effort. This textbook will not follow this convention in using variables and constants; however, you should be aware that such a convention does exist.

4.3 Numeric Data and Types

VB has different ways of storing and handling numbers, depending on the *types* you declare for them. Some data types can handle only numbers without decimal points; others can handle a wide *range* of numeric values, from a very small fraction to a huge value. In addition, some data types have a low *precision*. They can be accurate for only a few significant digits. Others have a high precision, capable of holding many significant digits. Some selected numeric data types used in VB are given in the following table.

Data Type	Size in Bytes	Value Range	Type Suffix for Variable	Type Suffix for Literal Constant
Byte	1	0 to 255	(None)	(None)
SByte	1	-128 to 127		
Char	2	0 to 65535	(None)	C
Boolean	2	True or False	(None)	(None)
Short	2	-32,768 to 32,767 (singed)	(None)	S
UShort	2	0 to 65,535 (unsigned)		
Integer	4	-2,147,483,648 to 2,147,483,647 (signed)	%	I

UInteger	4	0 to 2^{32}-1 (unsigned)		
Long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed)	&	L
ULong	8	0 to 2^{64}-1 (unsigned)		
Single	4	1.401298E-45 to 3.402823E+38 (in magnitude)	!	F
Double	8	4.94065645841247E-324 to 1.79769313486231E+308 (in magnitude)	#	R
Date	8	0:00:00 on January 1, 0001 through 11:59:59 PM on December 31, 9999		# (Enclose)
Decimal	16	79,228,162,514,264,337,593,543,950,335 (in magnitude; with 28 places to the right or left of the decimal)	@	D

Notice the last two columns in the table. The Type Suffix for Variable column shows the character to append to a variable to indicate the data type in declaration. As noted previously, declaring data with type suffixes makes the code less readable. Do not use them in your own code. They are listed here for your reference in case you encounter them when reviewing someone else's code. The Type Suffix for Literal Constant column shows the character to append to a literal to force or make explicit its data type. For example, to indicate that the number 12345 is of the Decimal type, you code:

```
12345D
```

Also note that to show a date literal, you must enclosed the constant with a pair of pound (#) signs, such as #12/31/2002#.

Byte,SByte and Char Types

As you can see from the preceding table, *Byte* and *SByte* type data use one byte of storage and can handle only a small range of values with a very narrow range and low precision (fewer than three complete digits); Byte data cannot handle negative values, while SByte data can. Their use is rather limited for computational purposes. The Char type has a wider range and higher precision; however, it usually is not used for computational purposes, either. Both the Byte and Char types are typically used to handle string data.

Character strings are coded either in ASCII code or Unicode. The ASCII code uses one byte of data storage to represent a character, while the Unicode uses two bytes. The ASCII code has a limited capability in representing characters only up to 256 different symbols. This range is sufficient to handle languages such as English, but insufficient for many other languages. As a result, the Unicode is used more universally because of its capability of representing up to 65,535 different symbols. As you may have already guessed, the Byte type is suitable for handling the ASCII code data, while the Char type is suitable for the Unicode data.

Boolean Type

Boolean type data are used to store the logical state—True (-1) or False (0). In contrast to Byte data, Boolean data are often used, although there are only two values. Many properties of VB controls can accept only Boolean data. These properties, such as Visible or Enabled, must be set to the desired state for the VB controls to work properly. Also, the If statement always involves the use of Boolean data.

Short, UShort, Integer, UInteger, Long, and ULong Types

The *Short and UShort type* (U for unsigined) use two bytes to store data. These data types are often used for counters, which seldom exceed several thousands. The *Integer* and *UInteger* types use four bytes to store data and are commonly used for computations dealing with integer numbers. They are also used as the index for

elements in a list. The *Long* and *ULong* types use eight bytes to store data. All these data types use exactly the same internal coding scheme to represent numbers, but have different ranges and precisions as the table shows.

When the result of a numeric operation exceeds the capacity that a data type can handle, an *overflow error* will occur at run time. In those cases in which you are not sure how big an integer number may be, you should use the Long or ULong type to be on the safe side.

Single and Double Types

In contrast to the data types discussed in the preceding subsection, the *Single* and *Double* types can handle data with fractional numbers and decimal points. These two data types use the *floating point* coding scheme to store data. This coding scheme breaks the stored data into two portions: One portion is used to keep track of the exponent, while the other portion stores the base value (mantissa) of the data. As an illustration, the number 123.45 can be expressed as 1.2345×10^2. The mantissa value, 1.2345, is stored in one part of the storage area while the exponent, 2, is stored in another part. This form of representation can also be used in code, where the mantissa is presented first followed by a symbol E (for exponent) and the exponent value. Here are some examples:

Actual Value	Floating Point Representation
34500	3.45E4
-34500	-3.45E4
.0002345	2.345E-4
-.0002345	-2.345E-4

Both the Single and Double data types can handle a wide range of values, as shown in the previous table; however, the Single type has a precision of approximately seven significant digits. This means if you have an amount like 1234567.89 that must be stored accurately, you *should not* use the Single type because the last few digits of the amount will be lost. If you need to use a floating point type to handle your data, you should keep this in mind. Note that you can append a suffix (F for single and R for Double) to the literal to explicitly show its type (force the compiler to handle it as such). For example, if you want 3.45 to be handled as Double, you can code:

```
3.45R
```

Try This

Type the following code in the code window of a new project without any VB control. Run the project and then click the form. What does the message box display? Change Single to Double in the Dim statement. Run the program again. What do you see this time?

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim Amount As Single
    Amount = 1234567.89
    MsgBox("The amount is " & Amount)
End Sub
```

The characteristics of the 12 numeric data types discussed in this subsection can be summarized as follows:

Data Type	Value Range	Precision	Speed*	Remark
Byte, SByte	Small	Low	NA	Seldom used for computation purposes but for handling character strings in ASCII code

Char	Medium	High	NA	Seldom used for computation purposes but for handling character strings in Unicode
Boolean	Small	Low	NA	Often used to represent state, such as True or False, On or Off
Short, UShort	Small	Low	2	Often used as counters
Integer, UInteger	Medium	High	1	Used for integer computations; cannot handle decimal point
Long, ULong	Medium large	High	3	Used for computations with large integer values; cannot handle decimal point
Single	Large	Medium	4	Can be accurate for about seven significant digits
Double	Very large	Very high	5	Requires twice the storage of the Integer or Single type
Decimal	Large	Very high	6	Requires twice the storage of the double type
* 1= fastest; 6= slowest				

Tip

When you are uncertain how many significant digits the actual data may be, use the data type with higher precision for the variable. Your program may become slower and will consume more storage space. (Between Short and Integer, the latter is faster.) Nevertheless, you are assured that your program will not encounter a serious unexpected problem. Story has it that the stock market crashed once because a computer program used by a large bank in New York City failed to process all transactions when the assumed maximum number of transactions was exceeded on that day. *Program robustness should outweigh the efficiency consideration.*

Other Numeric Data Types

In addition to the numeric data types described previously, there are also data types of Decimal and Date. They use 16 and 8 bytes of storage, respectively. The Decimal type can handle 29 significant digits, and you can put the decimal point up to 28 places to the right or left of the number. This data type has the highest precision and is not subject to rounding errors. However, it requires much more storage and is the slowest in computation. Use this data type only if your application cannot tolerate rounding errors.

The Date type uses 8 bytes of storage to handle date/time data. Each unit (recognized as a tick) represents 100 nanoseconds of time beginning from 00:00:00 January 1, 0001. The value of the date such as #12/31/2008# is stored internally in terms of the number of ticks from mid-night of January 1, 0001.

Declaring Numeric Data Types

As explained in Section 4.1, "Classification and Declaration of Data ," you can declare a numeric variable using the following syntax:

```
Dim name As data type
```

or

```
Dim name+type suffix
```

You probably have noticed in the first table that not all data types can be declared with a type suffix. Again, for code clarity, avoid declaring variables with type suffixes. Here are some examples of variable declarations:

```
Dim NewRec As Boolean 'Declare NewRec as Boolean type
Dim NumberOfEmployees As Integer  'Declare # of employees as integer
Dim RecordCount As Long     'Declare RecordCount as long
Dim HourPay As Single 'Declare hour pay as single
```

```
Dim BillsIncome As Double   'Declare Bill's income as double
Dim Amount As Decimal 'Declare Amount as Decimal
```

The Object Data Type

What if you declare a variable without explicitly giving its type? The variable is then given the Object data type. This is also the data type that VB assumes for you if you use a variable without declaration and you have an Option Explicit Off statement. The Object data type can take on any type of data, numeric or string. In a way, it is a versatile type with a lot of flexibility and capability; however, this versatility can also cause unexpected results and a lot of confusion. Consider the following code:

```
Dim V As Object
V = "1.23"
V = V + "45"
```

What result do you expect? V will contain a string "1.2345" because both V and "45" are strings, and the + symbol is considered a string concatenation operator (see the string section that follows for more explanation). But what do you expect from the following?

```
V = "1.23"
V = V + 45
```

The answer is 46.23. Because 45 is a number, VB considers + as an addition operator. It thus converts the string "1.23" to the number 1.23 and then performs the addition. After the last statement, V is no longer a string but rather a floating point Double type! In addition to the potentially unexpected result and confusion, the Object type also uses more computer memory to store, and is slower because of the additional complexity involved in handling this data type. *Stay away from this data type as much as possible*.

Determining the Object Type

When a variable is declared as the object type, you may find a need in your code to determine its data type in order to manipulate the data properly. VB provides two means to handle this: The ***TypeOf*** operator and the TypeName function. The ***TypeOf*** operator returns the type of the object and has the following syntax:

```
If TypeOf(Variable) Is Type Then
```

For example, to determine whether a variable, A, is a string, you can code:

```
If TypeOf(A) Is String Then
.
.
End If
```

Notice that this operator can only be used in an If statement and must use ***Is*** to check against the type in question.

The TypeName function returns a string that represents the type of of the object. For example, assuming T has been declared as a string variable and A in the preceding code is of the string type, T will contain the value "String" after executing the following code:

```
T = TypeName(A)
```

Strict Implicit Type Conversion

The confusing results from using the Object type is partially caused by VB's permissive data type conversion rules. By default, VB allows the programmer to write code without much concern about how data should be

converted from one type to another. This convention appears to be convenient for the beginner to learn the language; however, it also can result in mysterious bugs in the program. The programmer might believe that the compiler does one thing, while it actually does another. The best way to avoid this kind of problem is to explicitly perform data conversion by code. For example, in the preceding example, if you mean for the expression to perform a string operation, you should code:

```
V = CStr(V) + CStr(45)
```

In the code, CStr is a function that converts the argument, V (and 45) into a string. Section 4.4, "Built-in Numeric Functions" presents a group of data conversion functions. Note that although at this point in the code V has functioned as a string, the CStr function is still needed under the strict conversion rule because V (being of the Object type) can be any data type at execution time.

You can have the complier force you to observe strict data conversion rules by adding a line of code at the beginning of the module:

```
Option Strict On
```

When the strict data conversion rule is on, you can implicitly convert data from a type of narrower range to that of a wider range, such as from Single to Double, but not the other way. In this case, if you really mean to do it, you will need to use one of the conversion functions explicitly. As you can infer, when your program does not have this line of code, you are in effect coding with Option Strict Off by default. It is recommended that you have this option on. Programs coded with explicit data type conversion code are easier to understand because the code explicitly shows the intended data type of each element and is less error prone. You can have the compiler enforce strict data type conversion rules for the project without any code by setting Option Strict On in the Options dialog box. The steps are exactly the same as turning on Option Explicit, as shown in Figure 4-1. This textbook will follow strict data conversion rules as soon as you learn those data conversion functions in Section 4.4.

Try This

Place the following code in a button click event.

```
Dim V As Object
V = "1.23"
V = V + 45
MsgBox("V=" & V)
MsgBox(V.GetType().ToString)
```

In the code, the GetType method returns the data type of the object. When you run the program and click the button, you should see in the message box that V = 46.23 and the data type is System.Double. Add the following line at the beginning of your code window (before any statement).

Option Strict On

Check the code in your Click procedure again. You should see V underlined. Change the code as follows and test the program again.

```
Dim V As Object
V = "1.23"
V = CStr(V) + CStr(45)
MsgBox("V=" & CStr(V))
MsgBox(V.GetType().ToString)
```

What do you see this time? The errors disappear because your code observes the strict type rules. Comment out the Option Strict On statement before proceeding further.

The Assignment Statement

You are now ready to learn about numeric operations. Before doing so, however, take another look at the assignment statement. Consider the following statement:

```
HourPay = 12.50
```

The equal sign (=) in the statement instructs the computer to move data resulting from the operation(s) on the right side to the variable on its left side. Statements with this structure are recognized as assignment statements, and are the most common statements in nearly all programs. In addition, to assigning a constant to a variable, you can also assign the value of a variable to another variable:

```
HourPay = StandardHourPay
```

It is important to note that because the result on the right side will be moved to the left, *any variables appearing on the right side must have been assigned with proper values* before execution reaches the statement. In the previous statement, you assume StandardHourPay is another variable and has been declared and assigned some value before execution reaches this line.

You can also assign data entered in the VB control to a variable. For example, the following code will assign to the variable HourPay whatever number the user enters into the text box named txtHourPay.

```
HourPay = txtHourPay.Text
```

Tip

Keep in mind that the language has strict syntax rules that govern how each type of statements should be constructed. In the case of the assignment statement, there can be only one variable on the left side. The following statements will result in *compile errors*:

```
I, J = 1
I and J = 1
```

Also, the result of the following statement may not be what you expected:

```
I = J = 1
```

?

In some languages, this may mean I and J both are assigned a value 1. But in VB, this line means to compare whether J is equal to 1. If so, assign a value True to I; otherwise, assign False.

Notice that the left side of an assignment statement must be a variable or a control property whose value can be set at runtime. Notice also that the equal sign in the statement does not mean *equal*, but represents an instruction to move the result on the right side to the variable on the left. You can write a statement such as the following:

```
I = I + 1
```

This statement says to add 1 to the value of I and then move the result to I. The effect is that I is increased by 1. Note that VB 2008 has special syntax for operations that can produce the same effect. The syntax appears as follows:

```
Variable Operation = Operand
```

where Variable = the variable to appear on the left side of the assignment statement (the first operand)
Operation = the mathematical operation such as + (addition), - (subtraction), * (multiplication), or / (division)
Operand = the other (second) operand in the operation.

For example, the same effect of the preceding statement can be obtained with the following line:

```
I += 1
```

The following table shows the effects of a few examples using this special syntax:

Example	Equivalent statement	Effect
I += 2	I = I + 2	Increase the value of I by 2
I += I	I = I + I	Double the value of I
I -= 3	I = I - 3	Decrease the value of I by 3
I *= 3	I = I * 3	Triple the value of I
I /= 2	I = I / 2	Halve the value of I

Swapping Two Values

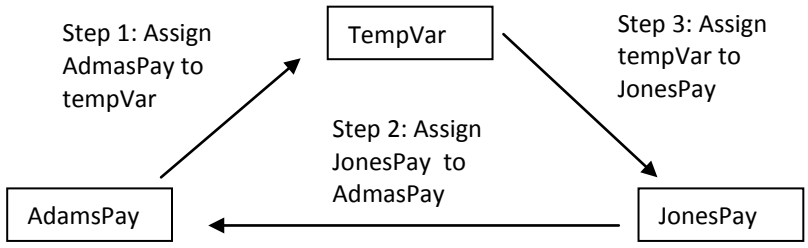
After being assigned a new value, the variable loses its previous content. It is important to remember that if you still need the original value of the variable, you will have to keep it in another variable before assigning a new value to it. For example, suppose you need to swap the values for two variables, named AdamsPay and JonesPay. The following code will *not* get the desired results:

```
AdamsPay = JonesPay
JonesPay = AdamsPay
```

This code fails because the first line assigns JonesPay (for example, 5,000) to AdamsPay. AdamsPay now has the value of 5,000, no matter what it had previously. This in effect results in both variables containing the same value, JonesPay. The second line assigns 5,000 to JonesPay, which was the original value of JonesPay anyway.

How do you solve this problem? As suggested, because you will need the original value of AdamsPay, you should find a way to keep its value before the variable is assigned with another value. You can do this by introducing a temporary variable to hold the original value of AdamsPay. After this is done, JonesPay can be assigned to AdamsPay, and the value in the temporary variable (Adams' original value) can then be assigned to JonesPay. The diagram in Figure 4-2 shows how this algorithm works.

Figure 4-2
Swapping data between two variables



The following code accomplishes swapping the values of the two variables:

```
TempVar = AdamsPay
AdamsPay = JonesPay
JonesPay = TempVar
```

In the future, you may see many situations where, at the first glance, the problem appears pretty difficult to tackle. In such cases, your solution may lie in an introduction of an additional variable.

Try This

Draw a button onto a new form. Set its Name property to **btnSwap** and its Text property to **Swap**. Enter the following code. Run the program, click the button, and observe the result. You should be able to verify the swap.

```
Private Sub btnSwap_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSwap.Click
     Dim TempVar As Single
     Dim AdamsPay As Single
     Dim JonesPay As Single
    AdamsPay = 5000
    JonesPay = 7500
    TempVar = AdamsPay
    AdamsPay = JonesPay
    JonesPay = TempVar
    MsgBox("After the swap, AdamsPay is " & AdamsPay & _
      " JonesPay is " & JonesPay)
End Sub
```

Numeric Operations

Refer back to the discussion of the assignment statement, the code that appears on the right side of the equal sign is recognized as *expression*, which can be a constant, a variable, or any operations on any combinations of constants and variables that result in a value. The symbols used to express *arithmetic operations* in VB are similar to the daily arithmetic symbols and are listed in the following table.

Symbol	Arithmetic Operations	Example	Meaning
-	Negation (unary)	-A	Negative value of A
+	Addition	A + B	A plus B
-	Subtraction	A – B	A minus B
*	Multiplication	A * B	A times B
/	Division	A / B	A divided by B
\	Integer division	A \ B	A divided by B
^	Power	A ^ B	A^B
Mod	Modulus	A Mod B	Remainder of A divided by B

The following are examples of valid expressions:

```
Salary + Commission
Mph * Hours
(Fahrenheit – 32) * 5 / 9
3.1416 * R ^ 2
20 Mod 3
```

The Mod operation divides the first operand by the second operand, and returns the remainder. The last expression, 20 Mod 3, will give 2 as a result because the remainder of 20/3 is 2.

Operational Precedence

When you combine several arithmetic operations in one expression, the order of execution resembles the ordinary arithmetic rules of precedence as follows (from highest to lowest):

Power (^)
Unary negation (-)
Multiplication and division (*, /)
Integer division (\)
Mod (Mod)
Addition and subtraction (+, -)

When the expression involves two or more operations of the same level of precedence, the execution goes from left to right.

Use of Parentheses

In many cases, you may find this order not exactly what you want. You can use parentheses to change the order of execution. An operation enclosed in a pair of parentheses will always be performed first. You can place as many pairs of parentheses as you want in an expression. You can also nest the parentheses. In this case, operations in the innermost pair will be performed first. Here are some examples of expressions using parentheses:

```
(7 + 8) * 3
(1 + Rate) ^ N ‘ compound interest for n periods
H * (A * X ^ (1 + B)) ‘total costs for the learning curve effect
```

Data Type Conversions

If an arithmetic operation in an expression has two operands that are of different data types, VB will have to perform conversion to make them the same type before it can proceed with the computation. In addition, if the result of the expression has a different data type from that of the variable on the left side of the assignment statement, VB will also make a conversion before moving the result to the variable. For example, consider the following code:

```
Dim I As Integer
    I = .5 * 20
```

There will be two conversion operations in executing the preceding code. In the code, .5 is a floating point constant; 20 is an integer constant. To make the two operands compatible, the number with smaller range or precision will be converted to the one with higher range and precision. Before the multiplication operation, 20 is converted to a Double floating point number. The multiplication is then carried out in a Double floating point operation. The result is a Double floating point value (10), which is then converted back to an integer number before it can be stored in the Integer variable, I.

As pointed out previously in this chapter, while VB will carry out these conversions automatically by default, it is advisable that you write your code explicitly to take care of the data conversions for semantic clarity and error minimization.

Computing Net Sale: An Application Example

You have learned a lot about numeric data in this chapter. These are in addition to those VB controls you learned in Chapter 3, “Some Visual Basic Controls and Events.” Have you been wondering how all these can be put together into an application? The purpose of this example is to show you how VB controls and numeric data can be used together for a simple but practical application. Please work along as you read.

The Net Sale Project

A retailer’s gross receipts consist of cash and credit cards. Both cash and credit card receipts are deposited daily in the bank. Before making the deposit, the owner needs to figure out her net receipts (which will be her actual amount of deposit). Cash is deposited on a one-for-one basis, meaning there is no discount or fee; however, she has to pay the bank a processing fee of 2.5% for all credit card receipts. You are to develop a project to compute the net receipts that she can use to prepare her deposit slips.

Input, Process, and Output

One way to analyze the project is to examine the input, process, and output of the project. It is usually easier to consider the output first. What the retailer needs to know is the net receipts that include both cash and net credit card receipts. It would also be nice to show the amount of credit card processing fees. The input should include the amount of cash and gross credit card receipts.

The process (computation) should be fairly simple. There will be no change in cash receipts between the gross and net amount. The credit card processing fee can be computed from the gross credit card receipts. The formula should be as follows:

```
ProcessingFees = GrossCreditCardReceipts x .025
```

This analysis can be shown as in the following table.

Input	Processing	Output
Cash receipts	None	Cash receipts
Gross credit card receipts (GCCR)	GCCR × .025	Credit card processing fees (CCPF)
	GCCR – CCPF	Net credit card receipts

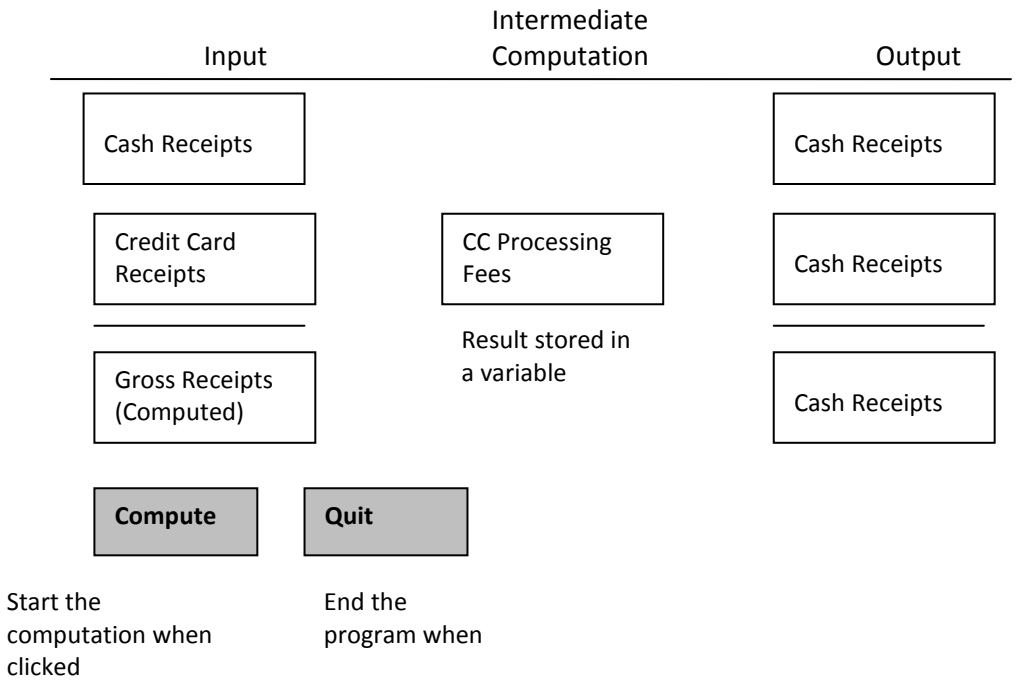
In addition, it would be nice to show total gross and net receipts. The retailer can use the total gross amount to verify the entered data. The total net receipt amount can be used for the preparation of deposit slips.

The Visual Interface

What VB controls should you use to obtain input from the user? Both cash receipts and credit card receipts are numeric. The exact amounts vary from day to day. It would appear that text boxes are the most suitable VB controls to use. Output can also be displayed in text boxes; however, you do not want the user to accidentally change the results. One possible solution is to use labels and set their properties to make them appear like text boxes. You can make the labels look similar to text boxes by properly setting their BackColor and BorderStyle properties. Alternatively, you can use texboxes and set their ReadOnly properties to True. Labels use less computer resources. However, they do not allow the user to copy and paste their text contents, while texboxes do. Assume a need for the copy and paste operation. Text boxes will be used here. You can also set text boxes to display amounts right-justified so that they will feel and appear right. For code clarity, you can use a named constant for the processing fee rate. You will also use variables to store computational data before the values are displayed in the output labels.

When should the computation be carried out and displayed? For simplicity, the computation will be done when the user clicks on a button with a text, Compute. It would also be nice to provide the user a button to quit. A sketch of the interface design is presented in Figure 4-3.

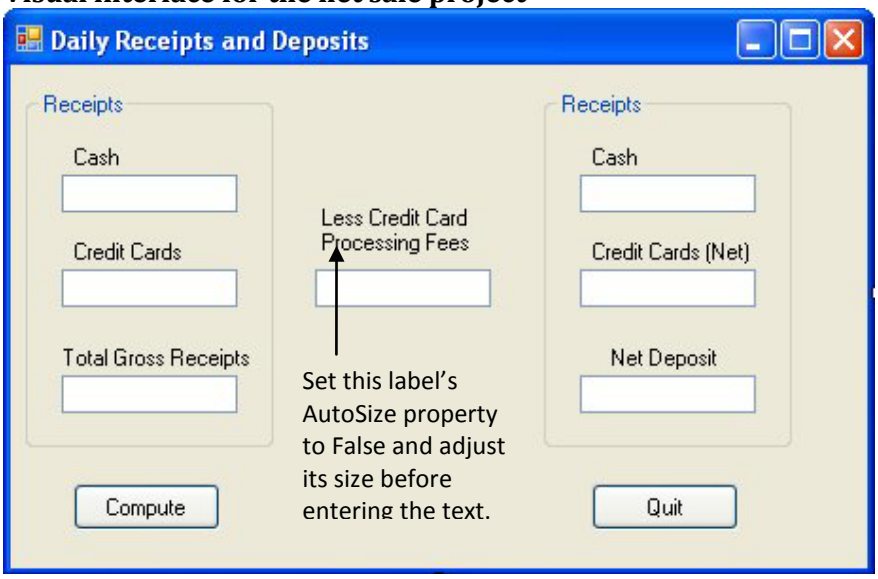
Figure 4-3
A sketch of input, process, and output of the receipts project



Of course, when you translate this sketch into a VB form layout, you must also add proper labels. The resulting visual interface appears as in Figure 4-4. The settings of selected properties of the VB objects used are summarized in the following table. Notice the property settings for the text boxes which are used to display the results of computations. To prevent these fields from accidental changes and preserve the accuracy of the results, these text boxes' ReadOnly properties are set to True. Once this property is set to True, the control's BackColor is automatically set to Control by the IDE. To make the appearance of these controls consistent with other text boxes, you can set their BackColor properties back to White through the Properties window.

Named Object	Description	Property	Setting
txtCash	For cash input	TextAlign	Right
txtCreditCards	For credit card input	TextAlign	Right
txtGrossReceipts	To display gross receipts	ReadOnly	True
		BackColor	White
		TextAlign	Right
txtProcessingFees	To display processing fees	ReadOnly	True
		BackColor	White
		TextAlign	Right
txtNetCash	To display cash receipts	ReadOnly	True
		BackColor	White
		TextAlign	Right
txtNetCreditCards	To display net credit cards	ReadOnly	True
		BackColor	White
		TextAlign	Right
txtNetReceipts	To display net receipts	ReadOnly	True
		BackColor	White
		TextAlign	Right
btnCompute	To perform computations	Text	Compute
btnQuit	To end the program	Text	Quit

Figure 4-4
Visual interface for the net sale project



In this application, the user will enter Cash and Credit Card Receipts on the left column. When he or she clicks the Compute button, all other amounts will be computed and displayed. To prevent accidental changes by the user in these computed fields, these controls' ReadOnly properties are set to True and their BackColor properties are set to White for consistent appearance.

Declaring the Variables and Constant

Before writing any code, try to identify the variables to use and their data types. In addition, you should also consider whether to name any constant to be used in the project. From your analysis, you know that:

- Cash and credit cards entered will be used in computations. You can use variables to hold the data entered into the text boxes. For computations, variables are faster and easier to reference than the properties of controls.
- You need variables to hold credit card processing fees and the net credit card receipts.

Tip

If you need to refer to a property several times in a procedure, assign it to a variable first; then use the variable from then on. The code executes faster with a variable than with a property of an object. Of course, if the property setting changes, the variable will need to be reassigned with the new setting.

The next question is: What data type should each variable be? In the previous discussion, it was suggested that the same data type be used for all the variables in a statement to avoid the problem with conversion. Because all these variables will be used to handle amounts, they will involve decimal points. This excludes the Integer and Long types from consideration. The floating point (Single and Double) and Decimal types are the potential candidates. Recall that the Single type can handle only about six or seven significant digits. If the retailer has a gross sale in excess of $100,000.00, the Single type will not have enough precision (field width) to handle the amount. The choice should be between Double and Decimal. The Decimal type has the advantage of being accurate, but is slower in speed than the Double type. Assume that the processing speed is the most important criterion for this project. Your choice will then be the Double type for all variables. With all these considerations, you will now place the following code in the *btnCompute Click event* procedure:

```
Dim Cash As Double
Dim CreditCards As Double
Dim ProcessingFees As Double
Dim NetCreditCards As Double
```

Because a variable should be declared before it is used, it makes sense to place all the preceding lines at the beginning of the procedure. Doing so also has another advantage. When you need to check the declaration for a particular variable, it is much easier to locate at the predetermined location.

How do you handle the credit card processing rate? Using a named constant for the rate appears to have some advantages. The name itself can give a clear purpose for the computation. More importantly, if the rate literal is used in several places in the project, when there is a change in the rate, you may have to search over the entire project to make the corrections. On the other hand, if you use a named constant, you define it at only one place and, therefore, revise the rate only once at the same place. With this consideration, you will place the following line in the general declaration area in the code window:

```
Const CreditCardRate As Double = 0.025
```

Again, the constant is declared as the Double type to be consistent with all the variables used to avoid data type conversion. Placing the declaration in the general declaration area will make the constant recognizable in all procedures.

Computations

Before considering any code, please note that you will not deal with strict type conversion at this point. After you have the variables declared, you can assign the content of the text boxes to them:

```
'Store user entered data in variables
Cash = txtCash.Text
CreditCards = txtCreditCards.Text
```

These variables can then be used in the computations. The credit card processing fees are computed by multiplying the (gross) credit card receipts by the processing fee rate:

```
‘ Perform fees computations
ProcessingFees = CreditCards * CreditCardRate
```

The net credit card receipt is the difference between the gross credit card receipts and the processing fees.

```
‘ Compute net credit card deposit
NetCreditCards = CreditCards – ProcessingFees
```

Displaying the Results

You are now ready to display the results. Net cash will be the same as its original amount. Both the computational results will be displayed in their corresponding textboxes.

```
‘ Display results
txtNetCash.Text = Cash
txtProcessingFees.Text = ProcessingFees
txtNetCreditCards.Text = NetCreditCards
```

The totals can be displayed by adding cash and gross/net credit card receipts.

```
‘ Display totals
txtNetReceipts.Text = Cash + NetCreditCards
txtGrossReceipts.Text = Cash + CreditCards
```

You also need to provide code to end the program execution. This is done in the btnQuit Click event procedure. As explained in Chapter 2, “Introduction to Visual Basic Programming,” a good way to handle this is to close the form:

```
   Me.Close
```

The Complete Code

Putting everything together, the complete code appears as follows:

```
‘Place the following constant declaration in the general declaration area
Const CreditCardRate As Double = 0.025
Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
System.EventArgs) Handles btnCompute.Click
Dim Cash As Double
Dim CreditCards As Double
Dim ProcessingFees As Double
Dim NetCreditCards As Double
‘Store user entered data in variables
    Cash = txtCash.Text
    CreditCards = txtCreditCards.Text
    ‘ Perform fees computations
    ProcessingFees = CreditCards * CreditCardRate
    ‘ Find net credit card deposit
    NetCreditCards = CreditCards - ProcessingFees
‘ Display results
    txtNetCash.Text = Cash
    txtProcessingFees.Text = ProcessingFees
    txtNetCreditCards.Text = NetCreditCards
‘Display totals
    txtNetReceipts.Text = Cash + NetCreditCards
    txtGrossReceipts.Text = Cash + CreditCards
End Sub
Private Sub btnQuit_Click(ByVal Sender As System.Object, ByVal e As
System.EventArgs) Handles btnQuit.Click
    Me.Close()
End Sub
```

An Issue of Presentation

Have you been working on our example? If you have (and you should), it is time to test run the program to see if it gives the correct and desired results. The program should work. You might, however, discover one minor

problem. When the credit card receipts have a fractional number, the resulting amounts on the form do not appear to be neatly presented; some may have no decimal points, and others may have a decimal point with several digits to the right. Is there a way to force the computer to display two decimal places for all amounts so that they appear neatly aligned by the decimal point? The answer lies in the Format function.

The Format Function

The Format function provides a wide variety of formatting capabilities. The syntax is as follows:

```
TextStringVariable = Format(Expression, Formatting String)
```

where Expression represents any valid expression that evaluates to either a string or a number to be formatted and Formatting string represents a string that specifies the format.

The formatting string can be a user-defined string or a name recognized by VB. For example, VB recognizes the name Standard, which will show the number with commas (as thousand separators) and a period with two decimal places; for example, Format(3000, "Standard") will give the result: 3,000.00. The following table lists selected named numeric formats.

Format Name	Meaning	Example Code	Result
Currency, C, or c	Display the dollar sign before the number with thousand separator and two digits to the right of the decimal point.	`Format(3000, "Currency")`	$3,000.00
Fixed, F, or f	Display at least one digit to the left and two digits to the right of the decimal point.	`Format(3000.1, "Fixed")`	3000.10
General Number, G or g	Format(3000, "G") without thousand separator	`Format(3000,"G")`	3000
Percent	Display number multiplied by 100 with a percentage sign (%) appended to the right with two digits to the right of the decimal point.	`Format(0.25, "Percent")`	25.00%
Scientific E, or e	Use standard scientific notation.	`Format(1234, "Scientific")`	1.234E+03
Standard, N, or n	Display numbers with thousand separator and two digits to the Right of the decimal point.	`Format(3000,"Standard")`	3,000.00

You can also provide your own user-defined formatting string. The following table gives a selected list of characters that you can use to form the formatting string.

Character	Explanation	Code example	Result
#	Place holder; either the number or nothing	`Format(5, "###")`	5
0	Place holder; either the number or 0	`Format(5, "000")`	005
.	Decimal place holder	`Format(5,"###.00")`	5.00
,	Thousand separator	`Format(5000, "#,###.00")`	5,000.00

+, -, $, (,)	Literal	`Format(5000,"$#,###.00")`	$5,000.00
%	Percent (multiplied by 100)	`Format(5, "###.00%")`	500.00%

Back to the example, you can use the Format function with the Standard format to display all the amounts in the form. To do so, the statement to display (net) cash should be changed to the following:

```
txtNetCash.Text = Format(Cash, "Standard")
```

All other statements to display the amounts should be changed similarly. You will notice that "Standard" appears in several lines of code. You can, instead, use a symbolic constant, such as FmtStandard, in place of the literal "Standard." You can declare the constant next to the declaration of credit card processing rate:

```
Const FmtStandard As String = "Standard"
```

The statement to display (net) cash can then be changed to the following:

```
   txtNetCash.Text = Format (Cash, FmtStandard)
```

All other amounts, including the two text boxes, can be formatted in a similar fashion. In case you decide to change the format later, such as to Currency, all you will have to do is to change the constant value. Notice that because FmtStandard is a constant name, it should not be enclosed in a pair of double quotes when placed in the Format function as the second parameter.

Look It Up

You can also use the FormatNumber function to format a number. Use the keyword FormatNumber in the Index tab of the help file for details on how to use the function.
For a complete list of the characters used for the user defined format string for numbers, use the keyword Format function to search the Index tab; then select user-defined numbers to view the table.

A Side Note on Design

In the previous example, the credit card rate was treated as a constant. In a real application, this may not be a good design because the rate can change. Each time the change occurs, you will need to change and recompile the program. A well thought out design should call for the rate to be saved in some file/database, and read in when your program starts. I have seen a commercial program with the social security tax rate hard-coded in the program. When the client company called for an update, the software company charged the client service fees. Don't you think this practice is taking advantage of their poor design?

4.4 Built-In Numeric Functions

VB 2008 also provides many functions and objects that can be used to handle conversion, mathematical, and financial operations. This section discusses selected conversion and mathematical functions that are commonly used.

Mathematical Functions

The following table lists three mathematical functions.

Function	Use	Code Example	Result
Fix(*x*)	Returns the integer portion of x by truncation	Fix(3.6)	3
Int(*x*)	Returns the integer portion of x by truncation	Int(3.6)	3

Rnd(x)	Returns a random number in the range of 0 to 1 (but less than 1) (See the following for additional explanation)	Rnd()	Fractional random number

Fix and *Int* differ in their way of handling negative values. Fix truncates the fraction; Int gives the next negative integer smaller than the parameter. For example, Fix(-3.3) will return -3, whereas Int(-3.3) will return –4.

Rnd is a random number generator that returns a fractional number of the Single type between 0 and 1 each time it is called. Depending on the value of the parameter given, it returns different results. If the parameter is a negative value, that value is used as the seed and results in the same random number if the same parameter is used. If the parameter is equal to zero, it returns the most recently generated random number. If no parameter is given, or if the parameter is greater than zero, it returns a random number in the sequence.

Rnd uses a seed to start the random number sequence. To avoid repeating the same sequence each time you run a program, you can use the *Randomize statement* to set a different seed as follows:

```
Randomize
```

The Randomize statement uses the system timer as the seed for the random number sequence. Because it is virtually impossible to have the same timer value repeated, there is no chance that the same sequence of random numbers will be generated.

The Random Object

In the preceding section, you saw the Random object used as an example to illustrate how a variable should be declared and associated with an object. The use of this object, which provides two methods of random number generation, is now explored. The *NextDouble method* takes no parameter, and returns a random number (of the Double type) that is greater than or equal to 0.0 and less than 1.0, providing the same functionality of the Rnd function. The *Next method* takes up to two integer parameters. When no argument is specified, the method returns an integer random number. When one argument is specified, the method returns a random number less than the argument. When two arguments are specified, the method returns a random number that is greater than or equal to the first argument, but less than the second argument. The second argument must be at least equal to the first argument; otherwise, you will encounter an argument out of range exception. The following statements will generate a random number between 1 and 100, inclusive:

```
Dim Rand As New Random()
Dim R As Integer
R = Rand.Next(1, 101)
```

As you can see, when you need to generate a series of integer random numbers within a range, this method is more convenient to use than the Rnd function. Note that the Random object randomizes its seed when it is instantiated. There is no need to use the Randomize statement.

Methods in the Math Object

VB 2008 also provides a Math object that contains many methods for mathematical computations. The following table gives a selected list of these methods:

Function	Use	Code Example	Result
Abs(*x*)	Returns the absolute value of x	`Abs(-3.2)`	3.2
Ceiling(x)	Returns the smallest whole number Larger than or equal to x	`Ceiling(2.3)`	3
Cos(*x*)	Returns the cosine value of x, where x is an angle in radians	`Cos(3.14159265359)`	-1
Exp(*x*)	Returns the value of e^x	`Exp(1)`	2.71828182845905
Floor(x)	Returns the largest whole number smaller than or equal to x	`Floor(2.3)`	2
Log(*x*)	Returns the natural logarithm of x	`Log(2.7182818284590`	1

		`5)`	
Log10(x)	Returns the common logarithm of x	`Log10(100)`	2
Max(a, b)	Returns the larger of the two numbers	`Max(10, 100)`	100
Min(a, b)	Returns the smaller of the two numbers	`Min(10, 100)`	10
Round(x)	Returns a whole number nearest to x	`Round(5.6)`	6
Sign(*x*)	Returns the sign (1, 0, or –1) of x	`Sign(-9)`	-1
Sin(*x*)	Returns the sine value of x, where x is an angle in radians	`Sin(0)`	0
Sqrt(*x*)	Returns the square root of x	`Sqrt(100)`	10

To use these methods, precede the method name with Math and a dot, similar to any typical method in an object. For example, to obtain the square root of 100, you code:

```
Math.Sqrt(100)
```

Tip

The Math object also provides two commonly used constants in math: E and PI. You can conveniently use PI to perform trig computations. For example, you can code the following to obtain the cosine value at 180 degrees:

```
Math.Cos(Math.PI)
```

Use the keyword Math to search the Index tab of the help file for all members in the Math objects.

A Side Note on Referencing Object Methods

Are you wondering about the difference in the treatment of objects between Random and Math? To use the Random object, you have to create (instantiate) and associate it with a variable. But to use the Math object, you have not done so. What causes the inconsistency—or more appropriately, difference—in treatment? The difference has to do with the types of methods in these objects. Technically, Math is a class. Methods provided in the Math class are static. These methods exist without having to be instantiated. You directly reference *static methods* and properties by using the class name, instead of the name of the instance (object) instantiated from the class.

On the other hand, the methods in the Random class are instance methods. You must have an instance of the class before you can reference these methods. And you must qualify the method name with the instance (object) name, not the class name.

Conversion Functions

The following table lists selected data conversion functions.

Function	Use	Code Example	Result
CBool(*x*)	Returns a Boolean type value; all nonzero values converted to True	`CBool(1)`	True
CByte(*x*)	Returns a Byte type value	`CByte("12")`	12
CDate(*x*)	Returns a Date/Time type value	`CDate("31-Dec-98")`	12/31/1998 12:00:00 AM
CDbl(*x*)	Returns a Double type value	`CDbl("12.34")`	12.34
CDec(*x*)	Returns a Decimal type value	`CDec("12.34")`	12.34
CInt(*x*)	Returns an Integer type value	`CInt("12.34")`	12
CLng(*x*)	Returns a Long type value	`CLng("12.34")`	12
CShort(x)	Returns a Short type value	`CShort("12.34")`	12
CSng(*x*)	Returns a Single type value	`CSng("12.6")`	12.6

In all the conversion functions listed in the table, *x* is an expression that can be evaluated to a numeric value. These functions typically are used to convert from one numeric data type to another; however, they can also be

used to convert strings that appear to human beings as numbers. For example, the expressions CLng(''3,456'') and CLng(''$3,456'') will both yield a result of 3456. The expression CLng('' '') will cause an error because a blank space is not considered a number.

You may have noticed that the preceding example program to compute net receipts behaves exactly as described here although the content of the text box is assigned to the numeric variable directly without calling any conversion function. The Text property of the text box is of the String type; however, those variables are declared to be the Double type. As noted previously in this chapter, in the absence of Option Strict specification, when the data type of the expression (source) is different from the variable on the left side (target) of an assignment statement, a data conversion will occur. One of these conversion functions is automatically called.

Note also, the Int and Fix functions differ from CInt and CLng in that the former pair truncates the fractional portion from the result, and the latter round the number. Using the number 5.5 in Int or Fix will have a result of 5, but will have a result of 6 with CInt or CLng.

Date/Time Functions

Date/Time functions are listed in the following table.

Function	Use	Code Example	Result
Today *	Returns current date as set in the computer	`Today`	Today
Now*	Returns current date and time as set in the computer	`Now`	Current date and time
TimeOfDay*	Returns current time as set in the computer	`TimeofDay`	Current time
Day(*Date*)	Returns the day of the Specified date	`Day(#12/31/2001#)`	31
Month(*Date*)	Returns the month of The specified date	`Month(#12/31/2001#)`	12
Year(*Date*)	Returns the year of the Specified date	`Year(#12/31/2001#)`	2001
DateSerial(year,month,day)	Returns a date given the year, month, and day in the parameter	`DateSerial(2002, 5, 31)`	5/31/2002
DateValue(*Date*)	Returns a date given a string such as “02-28-1998” as the parameter	`DateValue(“May-31-2002”)`	05/31/2002
DateAdd(interval, number, date)	Returns a Date/Time value after adding the specified date by the specified number of intervals	`DateAdd(DateInterval.Month, 2 ,#03-31-2002#)`	05/31/2002
* Technically, these three are properties of the System.DateTime object rather than functions. You can use them not only to return values, but also to set the date/time for your system.			

Note that the Day function must be qualified by Microsoft.VisualBasic for the IDE to accept your code.
The DateAdd function is a versatile function in handling Date/Time data. You specify the interval parameter by using the DateInterval enumeration. For example, to indicate an interval in year, you code DateInterval.Year. The number parameter requires an integer, while date can be any Date/Time value (Note that a date literal must be enclosed by a pair of the # sign). For example, the following expression:

```
DateAdd(DateInterval.Second, 5, Now)
```

will return a Date/Time value that represents 5 seconds from now.

Look It Up

If you need to perform Date/Time calculations, use the keywords "DateAdd function" and "DateDiff function" to search the index tab of the help file. These two topics give you specific details and fine points for various computations.

Formatting Date Data

Recall that you can use the Format function to format numeric data. You can also use the same function to format date/time. The following table shows a list of selected predefined date/time formats.

Format Name	Meaning	Code Example	Result
General Date, or G	Display a date and/or time	`Format(#12/31/2002#, "General Date")`	12/31/2002 12:00 AM
Long Date, or D	Display a date in the computer's long date format	`Format(#12/31/2002#, "Long Date")`	Tuesday, December 31, 2002
Medium Date	Display a date in the medium date format	`Format(#12/31/2002#, "Medium Date")`	Tuesday, December 31, 2002
Short Date or d	Display a date in the computer's short date format	`Format(#12/31/2002#, "Short Date")`	12/31/2002
Long Time or T	Display a time using the computer's long time format, including hours, minutes, and seconds	`Format(#3:23:25 PM#, "Long time")`	3:23:25 PM
Medium Time	Display a time using the computer's medium time format	`Format(#3:23:25 PM#, "Medium time")`	3:23:25 PM
Short Time or t	Display a time using the computer's short time format	`Format(#3:23:25 PM#, "Short time")`	3:23 PM

Look It Up

You can also use the FormatDateTime function to format date/time data. Use the keyword FormatDateTime to search in the index tab of the help file for information on how it can be used.

Of course, you can also create your own user-defined formatting string to format date/time. The following table gives a selected list of examples.

Code example	Result	Comment
`Format(#2/23/2002#, "MM/dd/yyyy")`	02/23/2002	M for month, d for day, y for year; number of characters defines maximum number of output digits
`Format(#2/23/2002#,"M/d/yy")`	2/23/02	
`Format(#2/23/2002#,"d-MMM")`	23-Feb	MMM specifies 3 characters for month

`Format(#2/23/2002#,"MMMM yy")`	February 02	MMMM specifies full month name
`Format(#2:35:6 pm#, "HH:mm:ss")`	14:35:06	H for hour (24 hour clock), m for minute, s for seconds; number of characters defines maximum number of digits displayed
`Format(#2:35:06 PM#, "h:m:s tt")`	2:35:6 PM	h for hour (12 hour clock); tt shows AM or PM

Look It Up

Use the keyword Format function to search the Index tab of the help file; then double-click user-defined dates/times under the format function heading for a complete list of characters for date/time formatting.

VB also provides financial functions that can be used to compute annuity, mortgage payments, and depreciation amounts. You can find these functions by searching the online Help file with the keyword Financial functions.

Using Functions

You can use functions in any expressions in the same the way you use a constant or variable. Here are some examples, assuming all variables involved have been properly declared and assigned with proper values:

```
TheWidth = Math.Abs (X1 – X2) 'width of a rectangle
GrossPay = CDbl (HourRate) * CDbl(HoursWorked) 'compute gross pay
' Compute total time under the learning curve model
TotalTime = InitTime * Units ^ _
    (1R + Math.Log(LearningRate) / Math.Log (2R))
Sample = Sqrt(100R * Rnd()) 'square root of 100 times a random number
MsgBox("Today is " & Today) ' Display today
```

4.5 String Data

As explained previously, a string consists of zero or more characters. The String data type is usually used to handle text such as names and addresses. It is also used for data code. For example, the characters M and F can be used to represent male and female. A string literal must be enclosed in a pair of quotes. String variables are used to store string data as shown in the following:

```
   LastName = "Smith"
```

The code moves the string literal "Smith" to the variable named LastName. You can also assign data in a control or a variable to another variable. For example, the following code will assign to the variable LastName the text that the user has entered in the text box named txtLastName.

```
LastName = txtLastName.Text
```

Declaring String Variables

You can declare a string variable with the following syntax:

```
Dim VariableName As String
```

For example, you can code:

```
Dim Address As String
```

All strings in VB 2008 are *variable-length strings*. A variable-length string variable can contain any number of characters. Its length depends on what is assigned to it.

String Operations and Built-In Functions

You cannot perform computations on string data; however, you can perform concatenation on strings. The concatenation operation joins two separate strings. You can use either + or & as the symbol for the concatenation operation. Here are two examples of concatenations:

```
EmployeeName = “John ” + “Smith”
EmployeeName = txtFirstName.Text & “ “ &  txtLastName.Text
```

The first line will result in the string “John Smith” being assigned to the variable EmployeeName. In the second line, the content of the text box named txtFirstName is first concatenated with a blank space. The result is then concatenated with the content of the text box named txtLastName. Finally, this result is assigned to the variable EmployeeName. If txtFirstName contains the string “John” and txtLastName contains the string “Smith,” EmployeeName will have the string “John Smith” as the result.

The Ambiguous + Operator

Both + and & symbols can be used for the concatenation operation; however, the + operator can be ambiguous. This is particularly true when one of the operands involved is a string, and the other is a number. Consider these lines:

```
Dim Text As String
Text = “123” + “123”
Text = “123” + 123
Text = “ABC” + 123
```

The first line will result in Text containing the string “123123” as expected. The second line, however, will result in text containing a string value of “246.” Because 123 is a number, VB considers + an addition operation, instead of concatenation; therefore, “123” is first converted to a number 123. The two numbers are then added together. The resulting value is converted to a string before it is assigned to the variable Text.

The third line will cause a run time Invalid cast exception. Again, VB treats the expression as a numeric addition first. When it is unable to successfully convert the string “ABC” to a number, it issues an error message. As you can see, using the + operator for string concatenation can cause confusing results. Avoid using it by all means.

If you mean to have string concatenations for the previous examples, the following lines will produce the correct results:

```
Dim Text as String
Text = “123” & “123”
Text = “123” & 123
Text = “ABC” & 123
```

Because VB knows the & operator is only used for string concatenation, it will first convert 123 into a string “123” in the last two lines. The first two lines will result in a string of “123123,” and the third line results in “ABC123.”

On Strict Conversion Again

If you observe strict data type conversion, you would not have coded some of the above lines as they were. In each of those lines that involved mixed types of data, you would convert one of the operands into the type of the other operand to ensure that the meaning is clear. For example, line 2 in the above two cases can be coded as follows (imagine the value 123 is actually represented by a numeric variable):

```
Text = “123” + CStr(123)
Text = “123” & CStr(123)
```

No ambiguity will result as to what kind of operation you expect to perform. Observing strict data type conversion is a good practice to follow. Set Option Strict On in your project to make the compiler force you to observe strict conversion rules.

Try This

Type the following code in the code window of a new project with Option Strict Off. Run the project, click the form, and observe the results; then replace the + symbol with the &" symbol. Repeat the test, and you should be able to get a good feel about how VB treats the two symbols.

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Click
Dim Text123 As String
Dim TextABC As String
    Text123 = "123" + 123
    MsgBox("Text123 is " & Text123)
    TextABC = "ABC" & 123
    MsgBox("TextABC is " & TextABC)
End Sub
```

Tip

Because the & symbol is also used as a type suffix, you must be sure that there is a space between this symbol and the variable before it to avoid a syntax error. For example, the following expression will cause an error.

```
FirstName& " " & LastName
```

But the following expression (notice the space before &) will produce no error.

```
  FirstName & " " & LastName
```

Built-In String Functions

In addition to the concatenation operation, many built-in string functions can be used to manipulate string data. The following table lists selected functions.

Function	Use
Asc(*char*)	Returns the ASCII key code value of a character (one byte); for example, Asc("A") returns 65
AscW(*char*)	Returns the key code value of a character (two bytes); for example, AscW("A"C) returns 65.
Chr(*n*)	Returns a character with an ASCII value n; for example, Chr(65) returns "A".
ChrW(*n*)	Returns a character in Unicode with a value n; for example, ChrW(65) returns "A"C.
Val(*S*)	Converts the string S to a numeric value; for example, Val("-23.5") returns –23.5.
Str(*n*)	Converts a number n into a string; for example, Str(-23.5) returns "–23.5".
StrReverse(*S*)	Returns a string of the mirror image of the string S; for example, StrReverse("AB") returns "BA".
LCase(*S*)	Returns the lowercase of the string S; for example, LCase("AbC") returns "abc".
UCase(*S*)	Returns the uppercase of the string S; for example, UCase("AbC") returns "ABC".
Space(*n*)	Generates a string with n blank spaces; for example, Space(1) returns " ".

StrDup(*R*, *C*)	Returns a string of R number of repeated characters, C. For example, StrDup(3,"A") returns "AAA".
Left(*S*, *n*)	Returns a string with the first n characters of the string S; for example, Left("My Name", 2) returns "My".
Right(*S*, *n*)	Returns a string with the last n characters of the string S; for example, Right("My Name", 4) returns "Name".
Mid(*S*, *b*, [*n*])	Returns a string of n characters that starts at the bth position in S; for example, Mid("My Name", 2, 3) returns "y N".
InStr([*p*],*S*, *c*)	Returns the position in S at which the content matches the string c. The comparison will start at position p. If p is omitted, it will start at position 1; for example, InStr("My Name", "y") returns 2.
InStrRev(*S*,*c*,[*p*])	Returns the position in S at which the content matches the string c. The comparison goes backward from position p. If p is omitted, it will start from the end; for example, InStrRev("02-02-43", "-") returns 6.
Len(*S*)	Returns the length of the string; for example, Len("My Name") returns a value 7.
LTrim(*S*)	Returns a string with all leading blank spaces trimmed off; for example, LTrim(" My ") returns "My ".
RTrim(*S*)	Returns a string with all trailing blank spaces trimmed off; for example, RTrim(" My ") returns " My".
Trim(*S*)	Returns a string with all the leading and trailing spaces trimmed off; for example, Trim(" My ") returns "My".
Replace(*S*, *a*, *b*)	Replaces all a's in S with b; for example, Replace("Jeff", "f", "s") returns "Jess".

Note that both the Left and Right functions must be qualified by Microsoft.VisualBasic; the actual code to reference to the Left function should be as follows:

```
Microsoft.VisualBasic.Left(MyString, Number)
```

Tip

If you encounter a design time error when coding a built-in function and you are sure the function name is correct, try to qualify the name with Microsoft.VisualBasic. The error may be caused by name conflicts. Examples of built-in functions requiring the qualifier include Left, Right, Timer, and Day.

Asc and Chr Functions

The Asc function returns the ASCII key code value of a character. For example, the ASCII key code value for A is 65; therefore, Asc("A") will return a value of 65. The Chr function converts a key code value to a character and can be considered the complement of the Asc function. Chr(65) returns a character A.

What do you need this pair of functions for? These functions provide a convenient way to perform computation on characters, which can become pretty tedious otherwise. For example, suppose you are developing a Scrabble game. You probably would need to provide a capability to generate a string of random letters. The Chr function makes this fairly easy.

You know there are 26 letters in the English alphabet; therefore, you need to generate a random number within the range of 26 so that each number will correspond to a letter. The following code should accomplish this:

```
Dim Rand As New Random()
Rand.Next(0, 26)
```

Recall that the Next method of the Random object returns an Integer number greater than or equal to the first argument but less than the second argument. The second statement should return an integer in the range of 0 and 25. Because A has an ASCII value of 65, adding 65 to the result of the Next method should produce the ACSII value of the random letter. You can then use the Chr function to convert this ASCII value to display the letter.

```
MsgBox("The next random letter is " & Chr(Rand.Next(0, 26) + 65))
```

As another illustration, suppose you want to display the next letter following the one that the user has entered in a text box named txtLetter when the user clicks a button named btnNext. What code will you provide in the button's Click event procedure? You will first obtain the keycode value of the current letter (using the ASC function). Add 1 to that value to get the keycode value of the next letter and then convert the result to the letter (using the Chr function). The code can appear as follows:

```
Private btnNext_Click(ByVal Sender As System.Object, ByVal e As System.EventArgs)
  Handles btnNext.Click
     Dim KeyAscii As Integer
    'Get the ASCII value of the letter
    KeyAscii = Asc(txtLetter.Text)
    MsgBox("The next letter is " & Chr(KeyAscii + 1))
End Sub
```

In this illustration, you assume the user will not enter the letter Z or more than one letter in the text box.

The preceding discussion deals with the ASCII code. If you are dealing with characters in Unicode code, the proper functions to use are the AscW and CharW functions. The only difference between these two functions and the two just discussed is that the AscW and ChrW pair deals with the Unicode (two bytes), while the Asc and Chr pair deals with the ASCII code (one byte).

Val and Str Functions

The Val function converts a string into a numeric value. For example, Val("123") will return a value 123. The Str function does the opposite. It converts a number into a string. For example, Str(123) will return a string " 123". Note that there is a leading blank space before the string "123". This can be important if you intend to check the length of the resulting string. If you do not want the leading blank space for any reason, you should use the Format function instead. For example, Format(123,''General Number'') will return a string "123".

Note that the Val function ends its attempt to convert a string as soon as it encounters a nonnumeric character, thus Val("1B2") will return 1 with no error. In addition, Val("3,456") will return a value 3; however, CLng(''1B2 '') will result in an error, whereas CLng(''$3,456'') will return a value of 3456. In a text box, such as, txtUnitsSold, you may expect that the user will enter numbers with commas; therefore, one of the conversion functions should be used. You may also anticipate that the user will leave the box blank or enter an invalid string. In such a case, the Val function will be safer. Thus, you are faced with a dilemma of choosing the appropriate function to use. One possible solution is to use yet another function to decide which function to use. The function ***IsNumeric(S)*** can be used to test whether the string parameter S is a valid numeric string. If it is, the function returns a value True; otherwise, False. Your solution can look like the following:

```
If IsNumeric(txtUnitsSold.Text) Then
    UnitsSold = CLng(txtUnitsSold.Text)
Else
    UnitsSold = Val(Val(txtUnitsSold.Text))
End If
```

You can even insist that the user enter a valid number before accepting the data. In this case, your code in the Else block will be a message requesting the user to enter a valid number.

LCase and UCase Functions

The LCase function returns a lowercase string. For example, LCase("ABC") will return a string "abc." The UCase function returns the uppercase string, so UCase("abc") will give "ABC".

Space Function

The function Space(n) will generate a string with n blank spaces.

Left, Right, and Mid Functions

The Left, Right, and Mid functions deal with the substring of a string. Left(S, n) returns the first n characters of the string S; therefore, Left("Upper",2) returns "Up". Note that this function has a name conflict with the Left

property of the form. To differentiate properly between this function and the Left property, this function must be coded as:

```
Microsoft.VisualBasic.Left(S, n)
```

Right(*S, n*) returns the last n characters in string S; therefore, Right(”Upper”,3) will return “per.” Note that this function also has a name conflict, and must be qualified with Microsoft.VisualBasic. The Mid(S, b, n) returns a string of n characters in S beginning from the bth position. Mid(”Upper”, 2, 3) will return “ppe”. If the third parameter is omitted, all the remaining string starting from the bth position will be returned. For example, Mid(”Upper”, 2) will return “pper”.

Mid can also be used as the target of a string assignment. For example, assume the string variable MyStr contains a string “Uppercase.” The following code will change the string to “Upperhand”:

```
Mid(MyStr, 6, 4) = “hand”
```

In this statement, the four characters in MyStr starting from the sixth position are replaced with a string “hand”. The last argument in Mid is an optional argument. If it is omitted, either the remaining length of MyStr or the length of the expression on the right side of the assignment (whichever is shorter) is used. The following table summarizes the examples discussed concerning these functions.

Code	Result (Value of A)
`A = Microsoft.VisualBasic.Left(''Upper'', 2)`	“Up”
`A = Microsoft.VisualBasic.Right(''Upper'', 3)`	“per”
`A = Mid(''Upper'', 2, 3)`	“ppe”
`A = Mid(''Upper'', 2)`	“pper”
`A = ''Uppercase''`	"Uppercase"
`Mid(A, 6, 4) = ''Hand''`	“UpperHand”

The InStr and InStrRev Function

The InStr(*S, ss*) function returns the position in S where its substring matches the string ss. For example, InStr(''ABCDE'', ''CD'') will return a value 3. Actually, the complete syntax for the function is as follows:

```
InStr([p], S, ss, [Compare])
```

where *p* = the position in S at which to begin the search
S = the string to be searched on
ss = the substring to search for
Compare = a value to specify the type of comparison and should be either CompareMethod.Text or CompareMethod.Binary. If CompareMethod.Text is specified, the search is not case sensitive; that is, “abc” and “ABC” are treated as equal.

Using this syntax, the expression InStr(''12-31-1999'', ''-'') will return a value 3 and the expression InStr(4, ''12-31-1999'',''-'') will return a value 6.

The InStrRev function has the following syntax:

```
InStrRev(S, ss, [p], [Compare])
```

where all the parameters have exactly the same meaning as those parameters in the InStr function. The positions in which the parameters are placed in the two functions are different, however. Also note that to specify the fourth parameter for the InStr function, you must also provide the value for the first parameter, as shown in the second through fifth code examples in the following table.

Code	Result (Value of Pos)
`Pos = InStr(“Containing”, “in”)`	6
`Pos = InStr(1, “Containing”, “in”)`	6

`Pos = InStr(7, "Containing", "in")`	8
`Pos = InStr(7, "Containing", "In")`	0
`Pos = InStr(7, "Containing", "In", CompareMethod.Text)`	8
`Pos = InStrRev("Containing", "in")`	8
`Pos = InStrRev("Containing", "In", ,CompareMethod.Text)`	8

These functions and the substring functions (Left, Right, Mid) together form a very useful group of functions in parsing strings. An illustration is provided in the next subsection.

The Len Function

The Len function returns the length of the string, including blank spaces. The string "" containing nothing between the quotes has a length zero; therefore, Len("") will return a value of 0.

Tip

If you need to check whether a string (say, MyStr) is of zero length, your code is more efficient with:

```
If Len(MyStr) = 0 Then
```

than with:

```
If MyStr = "" Then
```

Trim, Ltrim, and Rtrim Functions

If you are interested only in the length of the visible characters in a string, you can use the Trim function to trim off the spaces and obtain the length accordingly. Len(Trim(MyString)) gives the length of the string with no leading or trailing spaces. If you are interested in trimming off only the leading spaces of any string, you can use the LTrim function to do the job. Furthermore, if you are interested in trimming off only the trailing spaces, you should use the RTrim function.

Look It Up

Search the Contents tab of the help file. Expand the following items (book icons) in sequence: Visual Studio .NET -> Visual Basic and Visual C # -> Reference -> Visual Basic Language -> Visual Basic Language and Run-Time Reference -> Keywords and Members by Task; then locate and click String manipulation keywords summary. You will find a list of functions organized by purpose and use. These functions can make your job of manipulating strings much easier. You should find Replace, Split, as well as those format functions useful.

The Calculating Vending Machine: An Example

To see how some of the previously mentioned string handling functions can be used to solve a programming problem, let's consider an example. In this project, a list box is used as the display of a vending machine. The customer buys an item by clicking on it. The visual interface of the project in action is given in Figure 4-5. At run time, this program behaves in the following manner:

- When the user clicks an item in the list box (named lstVending), he or she buys it. Clicking on the same item multiple times means buying the same item multiple times. (No confirmation. No refund. It's a vending machine, after all.) The price is added to the total. (*Note:* Make sure that the SelectionMode property is set to One.)
- When the user finishes the purchase, he or she clicks a button (named btnShow) with the text, "That's all." The computer will then display all the items purchased as well as the total, as shown in Figure 4-5.
- When the user clicks another button (named btnQuit) with the text Quit, the program ends.

Figure 4-5
The vending machine in action

Setting Up the Vending Machine

The list box will display the items available as the program starts. So, how do you add the items to the vending machine? You can use the list box's Items.Add method to add the food items along with their prices. (In a real application, food items and their prices can be read from a file or database.) You would also like to align the prices in a column. This can be accomplished by inserting a *tab character* between the food item and its price. The key code constant for the tab character (Chr(9)) is defined in VB as vbTab. You will have six items in the vending machine. The code should be placed in the Form Load event procedure:

```
Private Sub frmVendingMachine_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    lstVending.Items.Add("New York Top Sirloin" & vbTab & "16.95")
    lstVending.Items.Add("Boston Red Lobster" & vbTab & "26.95")
    lstVending.Items.Add("Alaska King Salmon" & vbTab & "15.95")
    lstVending.Items.Add("Home Made Sandwich" & vbTab & "12.95")
    lstVending.Items.Add("Washington Red Apples" & vbTab & "4.95")
    lstVending.Items.Add("Florida Orange Juice" & vbTab & "5.95")
End Sub
```

Notice that the length of each item varies slightly. The price of each item has a slightly different length, too. However, the use of the tab character (vbTab) helps align the prices fairly evenly.

Look It Up

Use the keyword, "constants" to search the Index tab of the help file. You will find all kinds of predefined constants that you can use in your program. The subtopic, miscellaneous, gives constants that are used in printing and display, such as tab, linefeed, and so on.

A Side Note on Coding Mechanics

Notice the repetitive references to lstVending.Items in the previous procedure. A coding structure exists that allows you to use a shorthand reference to the same object or collection. This involves the use of the With…End With structure, which has the following syntax:

```
With Object
    Statements
End With
```

Inside the With block, any statement that begins with a dot will be automatically qualified with the object. For example, the preceding Form Load event procedure can be rewritten as follows:

```
Private Sub frmVendingMachine_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    With lstVending.Items
```

```
        .Add("New York Top Sirloin" & vbTab & "16.95")
        .Add("Boston Red Lobster" & vbTab & "26.95")
        .Add("Alaska King Salmon" & vbTab & "15.95")
        .Add("Home Made Sandwich" & vbTab & "12.95")
        .Add("Washington Red Apples" & vbTab & "4.95")
        .Add("Florida orange Juice" & vbTab & "5.95")
    End With
End Sub
```

All the .Add statements will be interpreted as lstVending.Items.Add because the With statement refers to lstVending.Items. This structure is not only convenient for coding, but also efficient for execution. The reference to the object is set for all statements in the block, thus avoiding the need to locate the object for each statement individually.

Tip

When you have a block of code that refers to an object several times, use the With…End With structure to reference the object. Your coding will be more efficient. The code executes faster, and the code is easier to read.

Determining the Item the Customer Has Chosen

When the user makes a selection by clicking an item in the vending machine, what does the computer know? You may recall that the item is identified by the list box's SeletectedItem property, as well as the SelectedIndex property. This item will be referenced several times in the program. You will assign the text to a variable, which is named TheItem. You can code the following:

```
TheItem = lstVending.SelectedItem ‘store the clicked item in a variable
```

Obtaining the Item Name

How do you get the food item name from this string? Recall that the string has both the name and the price. You know that the tab character separates the item and its price. If you know the position of the tab character, you should be one step closer to solving your problem. You can use the InStr function to find this position. If you use the variable TabPos to keep track of the position, you can code the following:

```
‘ Find the position of vbTab in the selected item
TabPos = InStr(TheItem, vbTab)
```

The item name starts at the beginning of the string, and ends one position before the tab character position. It can be computed as follows:

```
Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1)
```

Putting Items Bought Together

The question then is what should you do with this result? Consider this list:

One, two, three, …

When you have only one item such as One, you will show only the item One. If you have more than one item, you want to insert a comma between the two items; that is, in addition to adding the item, you also add a comma and a space. If you use a variable named ItemsBot to track the items purchased, you will assign the item name to ItemsBot when this item is the first one; that is:

```
ItemsBot = Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1) ‘For the first item
```

If the item is not the first one, you will add a comma (with a space) as well as the item name to the existing string. The code should be as follows:

```
ItemsBot = ItemsBot & “, “ & Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1)
```

Showing Items on Separate Lines

Here is an additional thought. What if you want to show each item on a separate line? This ensures that the text to be displayed is wrapped properly. The character combination Chr(13) & Chr(10)—recognized as carriage

return and line feed—will make the computer display what follows it in the next line. This constant has a predefined name vbCrLf. To make each item appear on a separate line, you can insert this named constant to the preceding code:

```
ItemsBot = ItemsBot & “, “ & vbCrLf _
  & Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1)
```

Telling the First Item from Others

Finally, you can tell whether the current item is the first item purchased by checking the length of ItemsBot. If the current item is the first one, nothing has been bought before. The length of ItemsBot will be zero. The code to add the item name to the list ItemsBot should be as follows:

```
If Len(ItemsBot) = 0 Then
    ‘First item, just keep the item name
    ItemsBot = Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1)
Else
    ‘Not the first item; add a comma, and concatenate the new item
    ItemsBot = ItemsBot & “, “ & _
        vbCrLf & Miscrosoft.VisualBasic.Left(TheItem, TabPos - 1)
End If
```

Obtaining the Price

So far, you have taken care of the list of items bought. How do you get the price from TheItem? Recall that the price starts from one position beyond the tab character, and extends to the end of TheItem. The Mid function can be used neatly to find the price string:

```
   Mid(TheItem, TabPos + 1)
```

Recall that the Mid function takes three parameters. When the last parameter is omitted, it returns the remainder of the string beginning with the position specified in the second parameter.

Accumulating the Total

You will use the variable Total to keep track of the total purchases by the customer. The price string obtained previously should be converted to numeric data before it is added to Total. You can use the Val function to convert the price string, such as:

```
‘ Add the purchase price to total
Total = Total + CSng(Val(Mid(TheItem, TabPos + 1)))
```

The Complete Code for a Selected Item

As described in the project requirements, the customer buys an item when he or she clicks on it; therefore, all the preceding code should be placed in the lstVending’s Click event procedure. The complete event procedure appears as follows:

```
Private Sub lstVending_Click(ByVal sender As System.Object, ByVal e
  As System.EventArgs) Handles lstVending.Click
    Dim TabPos As Integer
    Dim TheItem As String
    TheItem = lstVending.Text 'store selected item in a variable
    ' Find the position of vbTab in the clicked item
    TabPos = InStr(TheItem, vbTab)
    If Len(ItemsBot) = 0 Then
        ' first item, just keep the item name
        ItemsBot = Microsoft.VisualBasic.Left(TheItem, TabPos - 1)
    Else
        'Not the first item; add a comma, a CrLf
        'and concatenate the new item
        ItemsBot = ItemsBot & ", " & vbCrLf & _
        Microsoft.VisualBasic.Left(TheItem, TabPos - 1)
    End If
```

```
    ' add the purchase price to total
    Total = Total + CSng(Val(Mid(TheItem, TabPos + 1)))
End Sub
```

Declaring the Variables Used

Notice that four variables have been used: ItemsBot, Total, TheItem, and TabPos. You need to consider their data types and scope. In terms of their data types:
Both ItemsBot and TheItem are used to handle string data. They should be declared as the String type.

- Total is used to accumulate the purchase amount. Because the total does not require a number of many digits but can be a fractional number, it can be declared as a Single floating point variable.
- TabPos is used to keep the position of vbTab in the string TheItem, which does not have many characters; therefore, TabPos can be declared as an Integer variable. (A Short variable is also appropriate, but the Integer type computes faster.)

In terms of their scope:

- Both TheItem and TabPos are used only in the lstVending Click event procedure. You should declare (and have declared) them in that procedure.
- Both ItemsBot and Total, however, are used in both lstVending Click procedure and btnShow Click procedure (discussed next). They should be placed in the general declaration area:

```
Dim ItemsBot As String
Dim Total As Single
```

Tip

Beware that you are not allowed to break a string constant into multiple lines as shown below:

```
lblDemo.Text = “This is a demonstration of _
    an extremely long string message”
```

In the first line, the compiler considers the space and the underscore as a part of the string literal; it fails to recognize it as an underscore for line continuation. The correct code should be:

```
    lblDemo.Text = “This is a demonstration of  ” _
    & “an extremely long string message”
```

Handling “That's all."

Finally, consider handling the event when the user clicks the button with the text, ”That’s all”. This is the event where your program should tell the user what he or she purchased as well as the total amount. The list of purchased items has been collected in the lstVending Click procedure. You would like for the computer to show a message such as the following:

```
You have purchased
Boston Red Lobster,
Florida Orange Juice.
The total is $32.90.
```

More Text on Separate Lines

If you examine the tentative output carefully, you should notice that both the first line and the last line of output are on separate lines from the purchase list. In addition, there is a period at the end of the purchase list as well as the end of the total. This means vbCrLf and the period should be added at the proper places. Notice also the amount displayed has a $ sign, and there should be two decimal places, suggesting that the total should be formatted with the Currency format. The code should appear as follows:

```
MsgBox(“You have purchased” & vbCrLf & ItemsBot & “.” & __
  vbCrLf & “The total is “ & Format(Total, “Currency”) & “.”)
```

A Finishing Touch

There is another fine point you should consider. What if the user clicks the “That’s All button before clicking any item? Using the previous line will result in the following message:

```
You have purchased
```

```
    .
The total is $0.00.
```

Although that's not too bad, it does not appear very neat. You would rather inform the user that he or she has not yet selected any item. Recall that you can test whether any item has been purchased by checking the length of ItemsBot; therefore, the more complete code is as follows:

```
If Len(ItemsBot) = 0 Then
    MsgBox("You have not selected any item yet.")
Else
    MsgBox("You have purchased" & vbCrLf & ItemsBot & "." & _
    vbCrLf & "The total is " & Format(Total, "Currency") & ".")
End If
```

Reinitializing Data for the Next Customer

After the message is displayed, you should also clear ItemsBot and Total so that the vending machine will be ready for another customer. To do this, you can assign a zero-length string to ItemsBot and a value 0 to Total:

```
ItemsBot =""
Total = 0
```

Complete Code, "That's all."

Recall that the name of the button is btnShow. The complete procedure to handle the event when the customer clicks the That's All button should appear as follows:

```
Private Sub btnShow_Click(ByVal Sender As System.Object, ByVal e As
System.EventArgs) Handles btnShow.Click
        If Len(ItemsBot) = 0 Then
            MsgBox("You have not selected any item yet.")
        Else
            MsgBox("You have purchased" & vbCrLf & ItemsBot & "." & _
              vbCrLf & "The total is " & Format(Total, "Currency") & ".")
            ItemsBot = ""
            Total = 0
        End If
End Sub
```

Finally, you should also take care of the event when the user clicks the Quit button:

```
Private Sub btnQuit_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnQuit.Click
    Me.Close()
End Sub
```

Additional Remarks

The vending machine example involved four event procedures:

- The Form Load event procedure to prepare the list box to show food items
- The lstVending Click procedure to accumulate the customer's purchases
- The btnShow Click procedure to display the result
- The simple btnQuit Click procedure to close the form

Run and test the program. If everything works correctly, you should be able to verify that the computer:

- Displays in the list box all the items with their prices as soon as the program starts
- Displays a reminder message if you click the That's All button before making any selection
- Accumulates and displays correctly the items and the total prices you have purchased when you click the That's All button after you have made selections
- Ends the project when you click the Quit button

Summary

- Data can be classified by use in computation into two broad categories: numeric and string.
- Data can also be classified by variability into variables and constants. Each variable must have a name, while a constant may or may not have a name—although in general, it is advisable to give a name to a constant.
- Constants are declared with the keyword Const. Variables are declared with the keyword Dim (or Static for local static variable).
- Variables and constants declared with the keyword Dim or Private at the class level are recognized by all procedures in the class and are of the class level scope.
- Variables and constants are accessible to all modules in the project when declared with the modifier Friend; and are accessible to all modules in all projects of the solution when declared with the modifier Public.
- Variables and constants declared inside a procedure (but outside of all code blocks) are recognizable only in that procedure and have the local scope. Variables and constants declared within a block, such as an If block, are recognizable only inside the block and have the block scope.
- Variables and constants of the class scope exist in the program for as long as the class exists.
- Procedure level variables declared with Dim exist for as long as the procedure comes into action. Each time the procedure is called, the variables get reinitialized. To preserve the values of previous calls, declare the variables with Static.
- Block level variables are out of scope and no longer exist outside of the block.
- Variables should be declared with a scope as narrow as possible. Constants should be declared with a broad scope. All constants used in a project should be declared in the same area for ease of code management.
- Numeric data can be further classified by internal code scheme into Byte, SByte, Char, Boolean, Short, UShort, Integer, UInteger, Long, ULong, Date, Single, Double, and Decimal. These data types have different ranges and precisions, and each is suitable for different type of computational need. Of these types, Byte and Char are typically used in handling string data, and are seldom used for computation.
- In terms of speed, the Integer and UInteger types are the fastest.
- The Object data type is capable of handling any type of data and, therefore, is very flexible; however, it is slow and can produce results that may be unexpected. Beginners are advised to stay away from this data type.
- You can use the Next method of the Random object to generate integer random numbers conveniently. To use the object, declare a variable of the Random type (class) and then associate it with an instance (object) of the Random class created (instantiated) by the use of the New keyword.
- The symbols for mathematical operations used in VB code are similar to those used in math. The operational precedence in VB is the same as in math.
- The Format function can be used to format numeric data as well as date and time.
- VB provides many functions and methods to handle mathematical computations as well as conversion of data from one type to another.
- To minimize unexpected results caused by automatic data conversion provided by VB, always explicitly use data conversion functions to convert data so that operands in an expression are compatible. It is strongly recommended that Option Strict On is set for the project.
- VB also provides many functions for string manipulation.

Explore and Discover

4-1. Declaring Static Variables in the General Declaration Area. Place the following line in the form's general declaration area. What do you see? Static declarations at the class level are not allowed.

```
Static A As Integer
```

4-2. Declaring with Public or Private modifier. Place the following code in the general declaration area. Do you see any problem?

```
Public Salary As Decimal
Private Wage As Decimal
```

Now cut and paste the code into an event procedure—say, form click. What do you see? You can't declare variables with these modifiers inside a procedure. They are for module/class level declarations only.

4-3. Implicit Type Declaration. With Option Strict off, place the following code in the form click event:

```
    Dim A
    A = "a"
    MsgBox(A.GetType().ToString)
```

Run the program and then click the form. What do you see? (The GetType method gets the data type of the object and ToString gives the string of the data type.) A variable declared with no explicit type is given the Object type whose specific type depends on the type of data actually assigned to the variable.

Now set Option Strict On. Check the code the again. An implicit type declaration is not allowed. If you really mean for the variable to be of the Object type, you must code:

```
  Dim A As Object
```

4-4. Data Type: Comparing Single, Double, and Long Types. Enter the following code in a new form. Run the project and then click any part of the form. Examine the results in the immediate window.

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim D As Double
    Dim C As Long
    Dim S As Single
    D = 123456789
    S = CSng(D)
    C = CLng(D)
    Console.WriteLine("D = " & D)
    Console.WriteLine("S = " & S)
    Console.WriteLine("C = " & C)
End Sub
```

What results do you see? Especially, what value does S have?

Now change the line for D to:

```
  D = 0.0000123456789
```

Repeat the testing process. What results do you see? Particularly, what value does C have?

4-5. The Rnd Function. Draw three buttons on a new form and name them btnForward, btnRecent, and btnSeed. Set their Text properties to Forward, Recent, and Seed, respectively; then type in the following code:

```
Private Sub btnRecent_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnRecent.Click
    Console.WriteLine(Rnd(0))
End Sub
Private Sub btnForward_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnForward.Click
    Console.WriteLine(Rnd(1))
End Sub
Private Sub btnSeed_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSeed.Click
    Console.WriteLine(Rnd(-1.23))
End Sub
```

Run the program. Click each button repetitively a few times and then click each button once. Have you got a good feel about the effect of each of the parameter values for Rnd? If not, look up in the Index tab of the online Help file using Rnd as the keyword.

4-6. Some Date/Time Functions. Draw a button on a new form. Name it btnToday, and set its Text property to Today. Type in the following code:

```
Private Sub btnToday_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnToday.Click
    Dim D As Date
    D = Now
    Console.Write("Year = " & Year(D))
    Console.Write("Month = " & Month(D))
    Console.Write("Day =" & Microsoft.VisualBasic.Day(D))
End Sub
```

Run the project and then click the button. What do you think the following functions do? Now, Year, Month, and Day.

4-7. The DateAdd Function. Draw a button on a new form, name it btnNextMonth, and set its Text property to Next Month. Type in the following code.

```
Dim D As Date = #07/01/2009#
Private Sub btnNextMonth_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles.btnNextMonth.Click
    D = DateAdd(DateInterval.Month, 1, D)
    Console.WriteLine(D)
End Sub
```

Run the project. Click the Next Month button repetitively. What results do you see? Isn't this a convenient way to find the same day of the next month; such as, 07/01/2009, 08/01/2009, 09/01/2009, and so on?

But beware. Change the date in the Dim statement to:

```
  #01-31-2009#
```

Repeat the experiment. What results do you get? Does everything turn out to be what you expected? Were you expecting to see month end dates?

There are many other uses of the DateAdd function, depending on the first parameter you specify. Check out the online Help file, using the keyword DateAdd function.

4-8. Problem with Broad Scope Variables. Draw a list box on a new form and name it lstWeekDays. Set its Items property (collection) to Sunday, Monday, Tuesday, through Saturday. Set its SelectionMode property to MultiSimple. Draw two buttons on the form. Name them btnChangeColor and btnShowSelected and then set their Text properties to Change Color and Show What's Selected, respectively. Declare I in the general declaration area as follows:

```
Dim I As Integer
```

In the btnChangeColor click event, enter the following code:

```
I + = 1
If I > 1 Then I = 0
If I = 0 Then
    LstWeekDays.BackColor = Color.Red
Else
    lstWeekDays.BackColor = Color.Blue
End If
```

In the btnShowSelected click event, enter the following code:

```
For I = 0 To lstWeekDays.SelectedItems.Count - 1
    Console.WriteLine(lstWeekDays.SelectedItems(I))
Next
```

Run the project, and click the Change Color button. What do you see? Each time you click, the list box's color changes from blue to red and from red to blue. Keep it red. Select one or two items in the list box and then click the Show What's Selected button. After all selected items are displayed, click the Change Color button again. Does it change the color? Click the button one more time. Does it change this time? What happened?

When you have items selected in the list box and clicked the Show what's Selected button, I is left with a value of 1 or greater after the click event is executed. Because it is a class level variable, the value is preserved. When you click the Change Color button, that value of I is increased by 1, resulting in a value greater than 1. The If statement changes it to 0; therefore, the list box's BackColor is set to Red, the same as what is already there (showing no change). Do you see how a broad scope variable can cause problems or confusion?

4-9. The SelectedText Property of the Text Box. Draw a text box and a button on a new form. Name the text box txtTest, and set its HideSelection property to False. Name the button btnHighLight and set its Text property to Highlight. Enter the following code:

```
Private Sub Form1_Click(ByVal sender As System.Object,  ByVal e As
  System.EventArgs) Handles MyBase.Click
    Console.WriteLine(txtTest.SelectedText)
End Sub
Private Sub btnHighLight_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnHighLight.Click
    txtTest.SelectionStart = 2
    txtTest.SelectionLength = 3
 End Sub
```

Run the project. Enter some long text in the text box and then do the following:

A. Click the form. Do you see anything?
B. Highlight a portion of the text box. Click the form. What do you see? Whatever you highlight should appear in the immediate window. Highlight a few different portions, and click the form to get a good feel.
C. Click the Highlight button. What do you see in the text box? The setting of the HideSelection property makes the highlight show.
D. Click the form. What do you see?

You should have a good feel about the read capability of the SelectedText property.

4-10. The SelectedText Property of the Text Box (continued). Add a button to the preceding project. Name it btnInsert and set its Text property to Insert. Enter the following code:

```
Private Sub btnInsert_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnInsert.Click
    txtTest.SelectedText = "***Inserted***"
End Sub
```

Run the program. Enter some text into the text box, and do the following:

A. Place the cursor anywhere in the text box and then click the Insert button. What do you see?
B. Highlight a portion of the text and then click the Insert button. What do you see this time? Is the highlighted text still in the text box?
C. Click the Highlight button and then click the Insert button. What do you see?

The write capability of the SelectedText property can be conveniently used to replace or insert text into a text box.

4-11. Scope of Variables. Type in the following code:

```
Dim A As Integer = 1
Dim B As Integer = 2
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim B As Integer
    Console.WriteLine("A =" & CStr(A))
    Console.WriteLine("B =" & CStr(B))
End Sub
```

Run the project and then click the form. Why is A equal to 1, but B equal to 0?

4-12. Scope and Lifetime of Variables. Enter the following code, and run the program. Click the form a few times. What do you observe? Why is B always 1, while A and C have different values after each form click?

```
Dim A As Integer
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim B As Integer
    Static C As Integer
    A = A + 1
    B = B + 1
    C = C + 1
    Console.WriteLine(“A =” & CStr(A))
    Console.WriteLine(“B =” & CStr(B))
    Console.WriteLine(“C =” & CStr(C))
End Sub
```

4-13. Lengths of Different Types of Variables. Do you want to know the length (number of bytes) a type of variable uses? Insert this code in the form click event. Run the program and then click the form. What do you see?

```
Dim A As Date
Console.WriteLine(“Length of A is “ & CStr(Len(A)))
```

Change the type in the Dim statement to another to find the length of the type of interest.

4-14. Which way is Faster? Insert the following code in a button click event. Run the program and then click the button. How much time does it take?

```
Dim I As Integer
Dim S As String
Dim T As Double
T = Microsoft.VisualBasic.Timer()
S = Space(32000)
For I = 1 To 32000
    Mid(S, I, 1) = "a"
Next I
MsgBox("Time used " & CStr(Microsoft.VisualBasic.Timer() - T))
```

Now try the following code:

```
Dim I As Integer
Dim S As String
Dim T As Double
T = Microsoft.VisualBasic.Timer()
For I = 1 To 32000
    S = S &      "a"
Next I
MsgBox("Time used " & CStr(Microsoft.VisualBasic.Timer() - T))
```

How much time does it take this time? (Note: In VB6, the first method is much faster, but in VB 2008 the reverse is true. There are many changes between the two versions. You just can't assume that what works better in one version will work equally well in the other.)

Exercises

4-15. Converting Temperature. The relationship between the two measurement systems of temperature can be expressed as follows: Fahrenheit = 32 + Celsius * (9/5). Design a project with two text boxes (with proper labels) and two buttons. When the user clicks the Convert to Fahrenheit button, your program will convert the number in the text box containing the Celsius data into Fahrenheit degrees and show the result in the text box for the Fahrenheit data. When the user clicks the Convert to Celsius button, the opposite conversion takes place.

4-16. Converting Measurements. A mile is equal to 1.60935 kilometers. One kilogram is equal to 2.20462 pounds. One acre is equal to 4046.85 square meters. On a new form, draw six text boxes with proper labels such as Miles, Kilometers, and so on, and two buttons. Set one button's Text property to Convert to Metric. Set the other button's Text property to Convert to British.

Provide the code so that when the user clicks the Convert to British button, your program will convert the data in the text boxes containing the metric data into British/American measurements and show the results in the proper text boxes. When the user clicks the other button, the opposite conversion occurs.

4-17. The Area and the Length of the Hypotenuse of a Right Triangle. The area of a right triangle is computed by multiplying the two sides of the right angle, divided by 2. The length of the hypotenuse is equal to the square root of the total of the sides squared. Design a project to compute the area and the hypotenuse. The user is expected to click a Compute button after entering the values of the sides. There should also be a Quit button.

4-18. Testing Divisibility. Develop code to test whether a number, N, can be evenly divided by another number, D. Use two text boxes for the user to enter N and D, respectively, with a proper label for each. After the user has entered the numbers, he or she clicks a button with the text Test. Your program will display a message indicating whether N can be evenly divided by D. (*Hint:* Use the Mod operator.)

4-19. Applying Strict Type Conversion. The net receipts example in the text (p. 24) assumes Option Strict Off. Modify the project by setting Option Strict On; then revise the code so that there is no syntax error.

4-20. Next Letter. The Next Letter example in this chapter (p. 40) assumes that the user does not enter a letter Z. Modify the program so that if the user enters Z, your program will show A as the next letter.

4-21. Last Day of the Previous Month. Write an expression that will give the last day of the previous month. Show the result in a text box. (*Hint:* The last day of the previous month is one day before the first day of the current month. Use the Month and DateAdd functions.)

4-22. Automatically Filling the Date. Write a statement that will fill a masked edit control named mskDateofOrder with today's date with a mask "##-##-####." Suppose you want this effect each time the control has focus. In which event should the statement be placed?

4-23. Julian Date. The Julian date represents a date by a five-digit number. The first two digits represent the year, and the last three digits represent the date sequence number in that year. For example, 99001 represents January 1, 1999. Write two short routines—one to convert a Julian date to a regular (Gregorian) date/time value and another to convert a date/time value to a Julian date. (*Hint:* You should find DateAdd and DatePart useful.)

To test your routines, draw two text boxes and two buttons on a new form. Set the Text property of one button to Julian and another to Gregorian. Use one text box (with a Julian Date label) to enter a Julian date; the other (with a Gregorian Date label) to enter the date in the mm/dd/yyyy format. When the user clicks the Gregorian button, your program will convert the Julian date in the Julian Date text box and display in the Gregorian Date text box a date in the mm/dd/yyyy format. Conversely, when the user clicks the Julian button, the regular (Gregorian) date is converted to a Julian date.

4-24. Rotation Encryption. A plain text string can be encrypted by rotating each letter in the text by 13 positions in the alphabet. For example, A (first position in the alphabet) will be rotated into N (fourteenth position in the alphabet); and N to A (13 + 14 = 27, which goes back to 1). An interesting feature of this encryption algorithm is that you can use it to both encrypt and decrypt. For example, the string POKE will be encrypted into CBXR, but applying the same algorithm on CBXR will result in POKE.

Draw a text box and a button onto a new form. Provide the code so that when the user enters a text string in the text box and clicks the button, the text will be encrypted by the rotation algorithm just described. The result should be displayed in the same text box. The text should be restored when the user clicks the button again (if your routine works properly). Assume the user is allowed to enter only uppercase letters. (*Hint:* Use a For loop. You should find the functions Mid, Asc, Chr, and the Mod operator useful.)

4-25. Elapse Time. Draw two buttons on a new form. Name them btnStart and btnStop and set their texts to Start and Stop, respective. Provide the code so that when you click the Start button, your program starts to measure the elapse time; when you click the Stop button, your program will tell you how long it has been since you clicked the Start button. Note that when your program starts, you should disable the Stop button. When you click the Start button, the Stop button should be enabled and the Start button disabled.

4-26. The Name Enforcer. Nearly all words in a name, such as personal name, street name, and so on, start with an uppercase letter. Provide the code to enforce the rule that all words begin with an uppercase letter followed by lower case letters, while the user is entering the name in a text box named txtName; that is, perform the uppercase or lowercase conversion automatically based on the letter's position in a word no matter which case—upper or lower—the user enters. Note that there can be more than one word in the text box. (*Hint:* Place the code in the KeyPress event. The first letter of a word is either the first

character in the text box, or one whose left side is a space. You need to use the built-in functions Asc, Chr, Ucase, and Lcase.)
Note: The built-in function, StrConv can perform this process on an entire string; but here, the problem requires the immediate conversion as each character is being entered.

4-27. Computing the Economic Order Quantity. The economic order quantity can be computed by the following formula:

Q* = Square Root of (2*P*D/S)
Where P = cost to place an order
D = annual demand
S = annual cost to carry an item

Design a project that will take P, D, and S as input (use text boxes); compute the EOQ; and display the result in a label. (*Note:* Include two buttons: one to handle computation and another to quit.)

4-28. Future Value of a Deposit. The future value of a deposit (loan) compounded annually can be computed as follows:

$F = D\ (1 + Rate)^n$
Where D = amount of deposit (loan)
Rate = annual percentage rate
N = number of years

Design a project to compute the future value of a deposit.
Question: Can the user use your program to determine how long it will take to double his or her money? How?

4-29. Finding Last Name and First Name. A label has a Text, Name (Last, First). A text box is on its right for the user to enter the name. When the user clicks the Show button, your program will compute the last name and first name, and display the results using the MsgBox statement; such as "Your name is John Smith." (Note that the user is expected to enter a name in the format like, "Smith, John.") When the user clicks the Quit button, your program unloads.

4-30. Handling the Esc Key. Suppose in exercise 4-29 you want the text box to restore its previous content when the user presses the Esc key while entering a new name. Provide the code to implement this requirement. (*Hint:* You need to place the code in the Enter event and the KeyPress event. The keycode value for the Esc key is Keys.Escape. Use the AscW function to find the keycode value for the key pressed [KeyChar].)

4-31. Replacing characters. Write a program that will display the date for today in short time format, such as 12/31/2009; however, you want the separators between month, day, and year to be dashes (-) instead of slashes (/).

4-32. Searching and Highlighting. Set up a form with two text boxes with proper labels. The first one will be used to enter the search word, and the second one will contain a long string of text to be searched on (name it txtLongText). This text box has several lines. (Set its MultiLine property to True.) When your program starts, the user enters a long text into the second box; then each time the user enters a search word and clicks on the Find Next button, your program will look for that word in the second text box. If the search word is found, it is highlighted in the second text box; otherwise, the message, "Word Not Found" will be displayed (use the MsgBox statement). (*Hint:* Set the HideSelection property of the

second text box to False. Use CompareMethod.Text as the fourth parameter for the InStr function to make the search not case sensitive.)

4-33. Repetitive Searching and Highlighting. Continuing from exercise 4-32, you would like to allow the user to click the Find Next button repetitively. Each time the user clicks the button, your program will find and highlight the next matching word in the second text box. When there are no more matches, the message, "Word Not Found" will be displayed. (*Hint:* Use the first optional parameter of the InStr function to specify the position to start searching. If the current search is a new search, it should start from position 1; otherwise, it should start from the previous position plus the length of the word being searched. You will need static variables to keep track of the previous position and word being searched.)

4-34. Repetitive Search and Replace. Modify the project in exercise 4-33. Add a text box, with proper label, and a button. Set the button's Text property to Replace. Provide the code so that when this button is clicked, the text found in txtLongText will be replaced with the text in the new text box. Make sure that when no matching text is found, or when the new text box contains no replacement text, a proper error message is displayed. (*Hint:* See exercise 4-10 for string replacement.)

Projects

4-35. Computing Daily Receipts. At the end of each day, a restaurant owner needs to determine the amount of sales, credit card processing fees, sales taxes, and net receipts. The cash register gives a tally of the amounts received by category. A sample of the tally appears as follows:

Cash	$1,569.82
American Express	4,031.99
Other Credit Cards	5,903.22
Total	11,505.03
Food	$8,135.03
Liquor	2,320.00
Beer	1,050.00
Total	11,505.03

American Express (AE) credit card receipts are mailed to AE, which will deduct a 4.25% of processing fee and make a direct deposit in the restaurant's bank account in two weeks. Statements are mailed to the restaurant. Cash and other credit card receipts are deposited in the bank the same night. These credit cards are subject to a 2.45% processing fee, which will be subtracted from the restaurant's account balance and shown in the monthly bank statement.

The food sales include a 7.25% sales tax. The liquor sales are subject to 10% liquor tax, and beer sales are subject to a 7% beer tax. Both liquor and beer taxes are expenses of the restaurant, whereas sales taxes are withholdings; therefore, the amount of food sales to be reported for this sample should be $8,135.03/(1 + .0725). The restaurant owner will report and remit all these withholdings and taxes to the state tax authorities every month.

Required: Design a project to compute the net receipts (monetary), bank deposit, credit card processing fees, net sales, net revenues (net of fees and taxes), and taxes (withholdings and expenses) owed to the state tax authorities. Your visual interface should be neat, clear, and uncluttered.

4-36. The Virtual Vending Machine. Imagine that on the third floor of the Business School building there is this Do-It-Yourself Vending Machine. On the top row of its display screen, it clearly identifies itself as the Almighty Virtual Vending Machine. This sign is shown as blinking light. (Make it look like a neon sign.) On the left side of this sign, there's a picture (image of your choice). On the right side of

the sign, there's a clock that shows the current date and time, and ticks every second. (*Hint:* Use one timer for the blinking light, and another timer for the date/time clock. Set the fonts of the labels to make them look neat.)

In the middle of the screen is a list of the menu choices. It shows six of your favorite dishes with prices; however, this list box is tall enough to show only four items. From this list, a customer can select any number of items to purchase. (*Note:* Use the list box. Set its foreground and background colors such that it will look like a vending machine. Set its SelectionMode property such that it will accept multiple selections. Use the Items.Add method to create the menu list. Concatenate a tab character between each item and its price.)

On the left side of this list box, there's a check box for the customer to indicate whether the customer is a student at your college. If so, the box (with proper label) below the check box is enabled so that the customer can enter the student ID; otherwise, this box and its related label are disabled. (*Hint:* Use a group box to contain the check box and the ID field, which should be the user's student number.)

On the right side of the list box, two radio buttons allow the customer to identify the payment method: cash or credit card. If the choice is credit card, the combo box below it (with proper label) is enabled to allow the customer to indicate the type of the credit card: Visa, Diners Club, Master Card, Discover, or American Express; otherwise, the combo box and its label are disabled. (*Hint:* Use another group box to contain the radio buttons and the combo box. Set the default credit card to Visa. Set the combo box's DropDownStyle property such that it will not accept any other kind of credit cards.)

At the bottom of the screen, there are two buttons. The left one has a text Buy; the right one, Bye. When the Buy button is clicked, the computer should display the items the customer has chosen, total purchase, and indicate the payment method. If the customer is a student at your college, the customer is entitled to have a 20% discount. The following is a sample display (use the message box for this purpose):

```
You have purchased chicken noodle soup, cheesecake, and beer.
The total purchase is $12 with a 20% discount. Net amount is $9.60.
You paid with a Visa credit card.
```

If the customer is not a student of your college, the middle line will read as follows:

```
The total purchase is $12 with no discount.
```

The machine will reset to its original state after the message is displayed. When the Bye button is clicked, the program ends. (*Note:* This project is different from the example in the chapter. You can no longer use the list box's SelectedIndexChanged event to keep track of what the customer has bought. Instead, you should figure out what has been selected in the Buy button's Click event. You will need to use a For loop to test what have been selected in the list box.)