

3 VARIABLES, ASSIGNMENT STATEMENTS, AND ARITHMETIC

LEARNING OBJECTIVES

After reading this chapter, you will be able to:

1. Understand a four-step process to writing code in VB .NET and its relationship to the six operations a computer can perform.
2. Declare and use different types of variables in your project.
3. Describe the various types of data used in a program.
4. Use text boxes for event-driven input and output.
5. Write VB .NET instructions to carry out arithmetic operations.
6. Discuss using VB .NET functions to carry out commonly used operations.
7. Use buttons to clear text boxes and exit the project.
8. Describe the types of errors that commonly occur in a VB .NET project and their causes.

WORKING WITH VARIABLES

Recall from Chapter 1 that a computer can carry out six basic operations:

1. Input data
2. Store data in internal memory
3. Perform arithmetic on data
4. Compare two values and select one of two alternative actions
5. Repeat a group of actions any number of times
6. Output the results of processing.

In Chapter 2, we created a VB .NET project that carried out only two of these operations by responding to input in the form of a mouse click to output a message. In this chapter, we are going to expand that project to handle input in the form of the name of the customer, the name of the DVD being rented, and the rental price. The project must then use arithmetic operations on data to compute the sales tax on the DVD and add this amount to the rental price to compute the amount due. The project must output the sales tax and amount due that are calculated and respond to a mouse click to print the form, clear text boxes, and terminate execution of the project.

To do this, we will need to carry out Steps 1–3 and 6; that is, we need to store both data and instructions in the computer's memory, perform arithmetic on the data,

and output the results of the processing. In this chapter we will not need to compare two values or repeat a group of actions.

Four-Step Coding Process

After the logic for the problem has been developed using IPO Tables and pseudocode, the next step is writing the program in VB .NET or another language. Creating a program in VB .NET follows a four step process:

1. Decide what variables and constants will be needed and declare them.
2. Input data from text boxes or other input controls.
3. Process data into information using arithmetic or string operations and store in variables.
4. Output values of variables to text boxes or other output controls.

We will follow this four-step process for a simple calculator, the Vintage DVDs example from Chapter 2, and, finally, a monthly payment application. First, we will discuss the use of variables in programming.

Variables

To be able to carry out any type of arithmetic operation, you need to understand the manner in which data are stored in the computer's internal memory. Internal memory can be thought of as a very large number of boxes called **memory cells** arranged in a manner similar to post office boxes. In each memory cell, one and only one data item can be stored at a time. To store a new value in the memory cell, you must destroy the old value. However, you can retrieve and use a value from a memory cell without destroying it.

Because memory cells can have different values stored in them, they are commonly referred to as **variables**. The location of the memory cell remains the same, but different values may reside there at different times during the execution of the program. For example, a memory cell (variable) called `Number` may begin with a value of 10, have this changed to 0, and then to -20.

Changing the value of a variable involves destroying the contents of the memory cell and resupplying the cell with a new value. So, in a computer program when we speak of a variable or a variable name, these terms refer to a memory cell in the computer's memory. The *value of a variable* in a program is the *current value* in the memory cell with the same name.

Naming Variables

Because data are stored in variables, in order to perform a desired operation, both you and the computer must be able to refer to a variable. To identify variables, you must assign names to them. Once this is done and a variable name is used in the program, there is no uncertainty as to which variable is being referenced. For example, if you have given a variable a name of `Number` and it has a value of 10 stored in it, then you can change the value stored in `Number` by using the appropriate instructions and referring to `Number`. Note that the name of the memory cell and its contents are *not* the same. If a variable is named `B`, this does *not* mean it has the symbol "B" stored in it.

Variables are named in VB .NET through a combination of the letters A through Z, the digits 0 through 9, and the underscore (`_`). Variable names are not allowed to contain any other *special* characters, and they must *always* start with a letter. VB .NET variable names are virtually unlimited in their length, but you will probably never want to create a very long variable name due to the problems in retyping it many times. VB .NET is *case-insensitive*; that is, case is not considered in variable names. For example, a variable named `NUMBER` is the same as a variable named `Number` or `number`. How-

ever, it is often useful to use upper- and lowercase letters in variable names, especially when words are combined to create a variable name. Doing this helps you understand the purpose of the variable. For example, `InterestRate` is easier to understand than `interestrate`. You should avoid using all capitals for variable names, since they can make the name difficult to understand.

A naming convention is also considered standard when using variables. The standard convention is to prefix the variable name with a three-letter indicator related to the variable data type. Variable data types will be discussed shortly. For example, a variable representing a tax amount that is declared as a decimal data type may be named `decTaxes` following this convention. This convention will be followed throughout the text.

In addition to these rules about variable names, another rule is that a variable name may not be one of the restricted **keywords** in VB .NET. These are words that VB .NET uses as a part of its language. For example, you could not use the keyword `Sub` as a variable name. However, it is valid to imbed a keyword in a variable name. For this reason, it would be valid to use `SubThis` as a variable name. VB .NET is very helpful in alerting you to keywords and errors in variable names. Keywords are displayed in blue and errors with a "wavy" underline. For example, if you tried to use `Sub` as a variable name, VB .NET would display an error message and show the word `Sub` with a wavy underline. Positioning the cursor over an error generates a ToolTip that explains the problem. If this happens, change the variable name and try again.

A common practice is to choose a name that in some way helps the programmer to remember the quantity being represented by the variable. Such devices that aid the programmer's memory are termed **mnemonic variable names**. The use of mnemonic variable names is considered good programming practice. The only problem with using long variable names is that they can be cumbersome to work with in the programming process. For that reason, it is a good idea to make sure the variable name represents the quantity but does not become too long. Table 3-1 shows quantities and a typical name for the variable corresponding to each of those quantities.

TABLE 3-1: Example variable names

Item	Variable Name
DVD rental price	<code>decDVDPrice</code>
Amount due	<code>decAmountDue</code>
Taxes due	<code>decTaxes</code>
Interest on home mortgage	<code>decMortgageInterest</code>
Year-to-date earnings	<code>decYTDEarnings</code>
Employee's last name	<code>strEmpLastName</code>

Data Types

Different types of data can be stored in a computer's internal memory. VB .NET supports 13 standard data types as well as user-defined data types. The data types you will use most often for your work include `String`, `Single`, `Double`, `Integer`, `Long Integer`, `Decimal`, `Date`, and `Boolean` (true or false).

The **String data type**, prefixed `str`, is used to store any string of ASCII symbols or characters that is enclosed in quotation marks. For example, a `String` variable might hold "120 Hilltop Rd." All of the remaining data types listed above are considered

numeric data types, because they store numeric data and can be used in arithmetic processing. It is important to remember that Numeric variables are very different from the numbers stored as String variables. Numerical variables can be used in arithmetic operations—addition, subtraction, multiplication, and so on. String variables, on the other hand, *should not* be used in such operations. The primary purpose of String variables is to provide for the input, output, and manipulation of sets or *strings* of characters.

Integers and **Long Integers** are designed to be used only with whole numbers, and they cannot have any fractional portion. **Single** and **Double precision** numbers can contain a decimal point and are used for calculations that require fractional values. They are used **floating point operations**, since the position of the decimal point is not fixed. If you need to work with currency or other values in which you need up to 28 decimal places of accuracy, then you would use the **Decimal** data type. You might not think of the **Date** or **Boolean** data types as being Numeric, but both can be used in arithmetic processing. For example, you would store today's data in a variable of the **Date** data type. Similarly, the Boolean type corresponds to values of 0 (false) or -1 (true), so if a value is multiplied by a Boolean variable, depending on the value of the Boolean variable, its sign may be changed.

Good programming practice calls for the type of variable to be shown as a prefix to the variable name just like prefixes are used as a part control names. For example, we used *dec* as a prefix for a variable that will hold decimal data. We will use these prefixes whenever we declare variables. Table 3-2 displays the characteristics of each Numeric data type along with the standard three letter prefix for each data type and other pertinent information about the data type.

In Table 3-2, the *Range* refers to the values that can be represented by this type of constant. A value involving an E followed by a number means that value raised to that power of 10. For example, 2.5E2 is the same as 2.5×10^2 . *Precision* in the table refers to how accurately this type of constant can store numbers. For example, Integers and Long Integers cannot store numbers with fractions, so they are accurate to only the whole number part of a value. Similarly, a Single variable can be accurate to only the first seven digits that are stored in it. So, 23,561.1903 is accurate to only the first seven digits, or 23,561.19, with the last two decimal places being lost.

Finally, the *Number of Bytes* refers to the amount of internal memory required to store this constant, where one byte equals eight bits. In general, the greater the range and precision, the greater the number of bytes required to store the data type.

In addition to the String and Numeric data types, VB .NET offers the **Object** data type. An Object data type can take on any of the other data types as needed. This generally occurs in the coding process when you fail to declare the type of variable you are using and VB .NET has to “guess” at the data type by using an Object type variable. In general, it is not a good idea to have VB .NET do this, so you should always “declare” the data type for each variable you use in your project. We will discuss this process in more detail next.

Using the coding convention prefixes for objects and variables can provide several advantages including:

- * Your code follows a standard that allows anyone who reads the code to immediately determine the data type of the variable;
- * Words that are normally reserved as VB .NET keywords may be used as part of an object or variable name, for example `strSub`;

TABLE 3-2: Characteristics of numeric data types

Numeric Data Type	Prefix	Range	Precision	Number of Bytes
Single	sng	1.4E-45 to 3.4E38 and -3.4E38 to -1.4E-45	Seven significant digits	4
Double	dbl	4.9E-324 to 1.8E308 and -1.8E308 to -4.9E-324	Fifteen significant digits	8
Decimal	dec	Very large positive or negative numbers to very small positive or negative numbers (too large and small to be shown here)	28 places to right of decimal	16
Integer	int	-2,147,483,648 to 2,147,483,647	Whole numbers	4
Long	lng	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Whole numbers	8
Short	sht	-32,768 to 32,767	Whole numbers	2
Date	dtm	Dates between January 1, 0001 and December 31, 9999	Not applicable	8
Boolean	bln	True or False	Not applicable	2

- * You can name different objects and variables with practically the same name. For example, imagine that you have a form with a label and a text box used to enter a user's age. If prefixes are used, you may name your text box *txtAge*, the corresponding label *lblAge*, and even an integer variable that may receive the value of the text box as *intAge*.

Declaring Variables

Good programming form requires that you always inform VB .NET of the data type you wish to be used with a given variable. This operation, termed *declaring variables*, is usually the first code instruction that is entered into an event procedure for a control. While variables can be declared in a number of ways, the most common declaration form uses the Dim keyword in combination with the variable name and the data type; that is,

Dim variable name as data type

where you supply both the variable name and the data type (use the data types given in Table 3-2 or, for a String variable, use the keyword String.)

For example, to declare a variable called *decMyIncome* as decimal, the statement would be:

```
Dim decMyIncome as Decimal
```

Similarly, to declare a variable called *strHerName* as a String type, the statement would be:

```
Dim strHerName as String
```

You may combine multiple declarations on one line by separating them by commas. For example, to combine the two previous declarations on one line, you would enter:

```
Dim decMyIncome as Decimal, strHerName as String
```

Multiple declarations of the same type need only have a data type after the last one. For example, the declaration:

```
Dim decMySalary, decYourSalary as Decimal
```

will result in both variables being declared to be the type Decimal.

Failure to include any data type declaration with one or more variables on a declaration line will result in all of them being declared as Object type variables. For example,

```
Dim decBigSalary, decSmallSalary
```

will result in both decBigSalary and decSmallSalary being declared as Object type variables.

In VB .NET, you can also *initialize* a variable to some value as well as declaring it in one statement. For example, if you wanted to declare a variable, intCounter, as an Integer and set it to zero in one statement, it would appear as:

```
Dim intCounter as Integer = 0
```



TIP: However you declare a variable in terms of upper- and lower-case letters, VB .NET will always change the case of the variable in later references to fit the declaration. This can be useful for a quick check on the variables that you are using.

Using the Option Explicit Statement

Throughout this book, we will always declare variables since it is our belief that the programmer should always be in control. Failure to declare variables leaves it up to VB .NET to use the Object type, which can lead to problems when VB .NET makes the wrong decision as to how a variable should be used. To ensure that you always declare variables, VB .NET automatically turns **Option Explicit** ON in every project. This option requires that all variables be declared and generates an error message if any are not. You could turn this option off in every project you create by typing in "Option Explicit Off" *before any other code in the module*, including the Public Class statement..

Mini-Summary 3-1: Variables and data types

1. A four-step coding process should be used to write programs in VB .NET.
2. Variables correspond to memory locations in the computer in which data and results are stored. They are assigned names by the programmer and used in the program.
3. A variety of data types can be stored in the memory locations; all are referred to by their variable names.
4. It is a good idea to declare variables as to the type of data on which they correspond. Variables are declared with the Dim statement.
5. Variables are automatically forced to be declared in VB .NET. To stop this, you must use the Option Explicit Off statement before the Public Class statement (not a good idea.)



It's Your Turn!

1. Give the most appropriate data type for each of the following values:
 - a. 23
 - b. 7.56
 - c. True
 - d. \$100.87

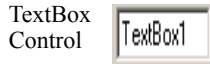
- e. 9534
 - f. 10089.34512
2. Create a variable name for each of the following quantities and give the appropriate data type:
 - a. Time until arrival
 - b. Take-home pay
 - c. Discount rate
 - d. Shipping rate per pound
 - e. Failure rate per 1000
 - f. Pass or fail?
 3. Write instructions to declare the variables for which you created names in the previous exercise.
 4. Rewrite your declaration from Exercise 3 to initialize the Discount rate to 0.06 as well as declaring it as a single type variable. Do the same for the shipping rate to initialize it to \$50 as well as declaring it to be a decimal type variable.
 5. For each of the following short scenarios, list the variables that you would need to solve the problem. In your list include the variable name, its data type, and the declaration statement that you would use.
 - a. W. Loman makes a living as a traveling salesman. In order to properly maintain his vehicle, Mr. Loman faithfully records the odometer reading (total miles a car has traveled) and the gallons of gas purchased. He then uses these values to calculate the miles per gallon for his vehicle since the last fill-up.
 - b. Joe “Bull” Cigar occasionally checks the prices of the shares of stock that he owns in the financial pages of his local newspaper. When he checks, he records the date, the closing price of the stock, and the number of shares traded. Bull would like to be able to save this information in a file on his computer. In addition, he would like to be able to calculate the percent change in stock price since his initial investment.
 - c. Tiger “Golden Bear” Shark is new to the professional golf circuit. He would like a simple program for his palm device in which he can keep a record of his performance by hole on the various courses that he plays. He will need to enter the name of the course, the date, the hole number, the hole par, his score on the hole, and comments about the hole. In addition, he would like the program to calculate his total score for a round and how far over/under par.
 6. Why is it not a good idea to include the **Option Explicit Off** statement in your projects?
-
-

EVENT-DRIVEN INPUT

Now that you understand the use of variables to represent the contents of a memory cell, the next question is: How do we get data into these variables? This is accomplished through some type of input. By **input** we mean using the keyboard, mouse, or other means to enter data into a variable. There must *always* be some form of input for processing of data into information to take place. In procedural languages, there are some types of input or read statements that enter data directly into a variable. However, for input to occur in event-driven languages like VB .NET, where the user typically enters data into a control on the screen, an event must transfer the data from the control to a variable. This is important to note: While you can enter data into a control,

if the appropriate event does not take place, then the data will never be used in the program. (This is not to say that we will never input data directly into variables; it does mean that transferring data from a control to a variable is more in keeping with the event-driven nature of VB .NET).

The control commonly used for event-driven input is the **TextBox control**, into which data can be easily entered into a text box. Once data have been entered into a text box, you can use an event like clicking a button to transfer the data to a variable, which can then be used in some type of computation. The TextBox control can also be used to display processed information. The property of the TextBox control that enables it to be used for input and output is its **Text property**. The Text property is *always* equal to whatever is displayed in the **Edit field** of the TextBox, that is, the area in the text box in which you may enter, edit, or display text. To enter text in a text box, you simply click the mouse pointer inside the blank area to change it to a vertical line and type the text. You may edit existing text by placing the pointer in the text box at the desired location and using the word processing editing keys to change the text.

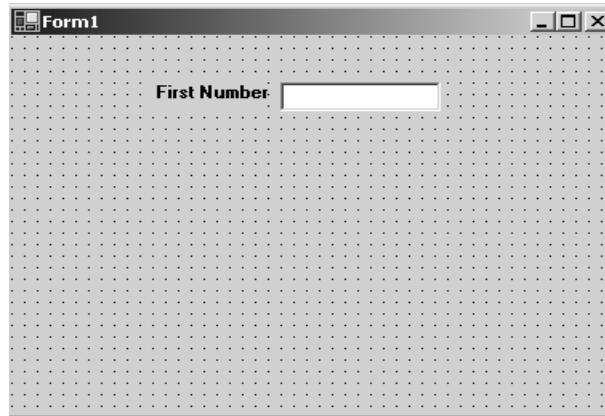


TIP: To improve the appearance of the labels by right-justifying them, you can change the TextAlign property for the label control from Left to Center or Right as well as controlling the vertical location.

To add a TextBox control to a form, you select it from the Toolbox, place it on the form, and change its Name and other properties, just like you did with the Label, PictureBox, and Button controls in Chapter 2. The Name property of the TextBox should be changed to begin with a prefix of *txt*. The default Text property that is initially displayed in a text box is *text* combined with a number, say, *textbox1*. Figure 3-1 shows a form with a text box centered in the upper half. Note that we have also added a Label control with a Text property that describes this text box as *First Number*. Note also that the text box is empty because we deleted the default value for the Text property. The name of the text box has been changed to *txtFirstNum* and the name of the label to *lblFirstNum*.



FIGURE 3-1. Text box added to form





TIP: You can easily change or delete the Text property for a highlighted text box by double-clicking it in the Properties window and entering new text or pressing the Delete key (**Del**). If you then highlight another text box, the focus will still be on the Text property, enabling easy changing or deletion of it.

Creating a Simple Calculator

As an example of using the TextBox control for input (and output), let's add two more text boxes and accompanying labels to the form shown in Figure 3-1. Let's also add two buttons and a label for which the Text property can be set equal to the underscore to form a line, to create a very simple calculator that sums the contents of the top two (input) text boxes and displays the result in the third (output) text box. Clicking the Sum button transfers the contents of the two input text boxes to variables, carries out the addition, and transfers the results to the output text box. Similarly, clicking the Clear button clears the text boxes and places the cursor back in the top text box. The resulting form is shown in Figure 3-2 and the Name and Text properties (where appropriate) are shown in Table 3-3. The form has also been made the Startup object for the project and saved as *Simple.vb*. In all cases, the Font properties for the texts have been set to 10 points with Bold style .



TIP: The order in which you add text boxes to the form controls the order that the cursor will follow when the **Tab** key is pressed. If the order is incorrect, you can change it in the Properties window with the **TabIndex** property of the text box. Make the TabIndex property for the first text box = 0, the second = 1, and so on.

FIGURE 3-2. Simple calculator design

The screenshot shows a window titled "Simple Calculator" with a standard Windows-style title bar (minimize, maximize, close buttons). The window background has a light gray dotted grid. On the left side, there are three text boxes stacked vertically. The first is labeled "First Number", the second "Second Number", and the third "Sum". To the right of these text boxes are two buttons: "Sum" and "Clear". The "Sum" button is currently selected, indicated by a dotted border around it.

TABLE 3-3: Controls for Calculator

Control	Name	Text
Form	frmSimple	Simple Calculator
TextBox	txtFirstNum	
TextBox	txtSecondNum	
TextBox	txtSum	
Label	lblFirstNum	First Number (right-justified)
Label	lblSecondNum	Second Number (right-justified)
Label	lblSum	Sum (right justified)
Label	lblLine	_____ (multiple underscore)
Button	btnSum	Sum
Button	btnClear	Clear

Using Assignment Statements to Input Data

To transfer the contents of the text boxes to variables, you must use an assignment statement. An **assignment statement** gives a variable a value by setting it equal either to an existing quantity or to a value that is computed by the program. In the case of transferring the contents of a text box to a variable, the variable is set equal to the Text property of the text box. The general form of an assignment statement is:

Control property or variable = value, variable, expression, or property

Note that the control property or variable being assigned a value appears on the left side of the equals sign, with a value, variable, expression, or control property appearing on the right. This is an important rule that must always be followed in the assignment statement. For example, an assignment statement might be:

```
decAmountDue = decPrice + decTaxes
```

where `decAmountDue` is a variable to which the sum of the `decPrice` and `decTaxes` variables is being assigned.

To input data from a control property, we simply set a variable equal to the control property. For example, to input the value for `intFirst` from `txtFirst.text`, the assignment statement is:

```
intfirst = txtFirst.text
```

We will discuss other applications of assignment statements in more detail later in this chapter, but for now we are interested in assigning the Text property of the text boxes to variables, summing those variables, and transferring the sum to a third text box. Since all of this is accomplished by the `btnSum` button, we need to double-click this control to open its Code window and declare three variables—`intFirst`, `intSecond`, and `intSum`—that will be used in the actual summation. Next, we need to use assignment statements to transfer the Text property of the text boxes to the variables `intFirst` and `intSecond` as shown above. These statements are shown in VB Code Box 7-1.

Using Functions

In looking at the assignment statements in VB Code Box 7-1, notice that `intFirst` and `intSecond` are Integer variables, but the Text property of any text box is a character string. This means that we have Numeric variables being set equal to String quantities—something that should be avoided since it requires VB .NET to decide how to

VB CODE BOX 7-1.
Code to input two numbers

```
Private Sub btnSum_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSum.Click
    Dim intFirst, intSecond, intSum As Integer
    intFirst = txtFirstNum.Text
    intSecond = txtSecondNum.Text
End Sub
```



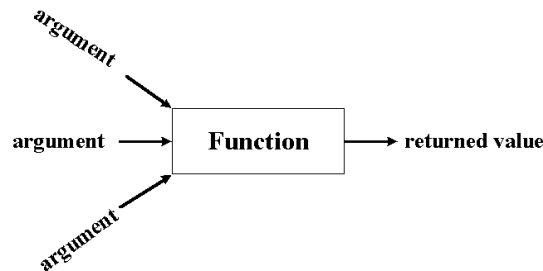
TIP: In most places in your project, you can position the pointer on a control, property, function, or other entity and press **F1** to display Dynamic Help about it.

handle the mismatch of types. While VB .NET *usually* handles this appropriately by converting the quantity on the right to the variable type on the left, it is not wise to leave this conversion up to VB .NET. Instead, we need to use a built-in VB .NET function to carry out the conversion. A **function** is an operation that takes one or more arguments and returns a single value. A common form of a function is:

variable = functionname(arg1, arg2, ...)

where arg1, arg2, and so on are the arguments of the function and the value is returned through the function name in an assignment or other statement. You might want to think of any function as a *black box* into which arguments are fed and from which a single value is returned, as shown in Figure 3-3. We do not need to know how it works to use it; we only need to know the name of the function, the appropriate form of the arguments and the type of value to be returned.

FIGURE 3-3.
Function as black box



Not all functions require arguments. For example, the **Today** function returns the system date on your computer with no arguments. Regardless of whether a function does or does not require arguments, all are used in a similar manner.

In our case, we will use the **CInt()** function to convert the Text property from a string to an Integer value number, which is then assigned to a Numeric variable. For example, instead of

```
intFirst = txtFirstNum.Text
```

the appropriate statement is

```
intFirst = CInt(txtFirstNum.Text)
```

The revised code for the btnSum button using the CInt function is shown in VB Code Box 7-2:

You might ask, why not use the text boxes directly for this computation? The answer to this question involves the nature of the Text property of a text box: It is a String, and as such it must be converted into a Numeric variable before being used in any type of calculation. If you try to use the Text property in a calculation, in some

VB CODE BOX 7-2. Code to use Val() function in inputting numbers	<pre> Private Sub btnSum_Click(ByVal sender as System.Object, _ ByVal e as System.EventArgs) Handles btnSum.Click Dim intFirst, intSecond, intSum As Integer intFirst = CInt(txtFirstNum.Text) intSecond = CInt(txtSecondNum.Text) End Sub </pre>
--	---

cases it will be converted into an Object data type while in others it will remain as a String. In either case, the results can end up surprising and *wrong!*

Computing and Displaying the Sum

Now that we have the contents of the two text boxes transferred to variables, the next step is to sum these two variables and assign that sum to a new variable, `intSum`, which must be declared. The value of `intSum` is then displayed in the output text box, `txtSum.text`. Once again, we use declaration and assignment statements to carry out these operations. First, we declare `intSum` to be of type `Integer` and then we use the summation sign (+) to sum the variables `intFirst` and `intSecond`, with the result being assigned to the variable `intSum`. This assignment statement will appear as:

```
intSum = intFirst + intSecond
```

In the second case, it might appear that we simply assign the value of the variable `intSum` to the `Text` property of the text box `txtSum`. However, we have the reverse of the problem we had before; we are now assigning the value of an `Integer` variable to a text box that is a `String`. To make sure that this conversion is handled correctly, we need to use the **CStr() function**, which converts the `Numeric` argument into a character string. In this case, the assignment statement is:

```
txtSum.Text = CStr(intSum)
```

If we add these two statements to the Code window for the `btnSum` button, we arrive at the code shown in VB Code Box 7-3. This is *all* the code you need to use our simple calculator. If you enter an `Integer` value in the top text box and a second `Integer` value in the second text box, and click the `btnSum` button, the sum of the two values will be displayed in the bottom text box. For example, if we enter 20 and 30 and click the button, the sum, 50, will be displayed.

VB CODE BOX 7-3. Code for btnSum button	<pre> Private Sub btnSum_Click(ByVal sender as System.Object, _ ByVal e as System.EventArgs) Handles btnSum.Click Dim intFirst, intSecond, intSum As Integer intFirst = txtFirstNum.Text intSecond = txtSecondNum.Text intSum = intFirst + intSecond txtSum.text = str(intSum) End Sub </pre>
--	---

Properties and Methods

Note that the `Text` property for the `txtSum` text box is shown appended to the text box name with a period. This **dot notation** is the way that properties are set at run time. The general form for setting a property at run time is:

object.property = value

and for using a property is:

value = object.property

The same notation is used to invoke a method for a control. Recall that methods define the actions that a control can carry out. The general form for using a method at run time is:

object.method

For example, a method that is associated with a text box is the **Focus()** method. This method shifts the cursor to the text box whenever it is invoked. If we wanted to put the cursor in the txtFirstNum text box, the statement would be:

```
txtFirstNum.Focus()
```

We will use this method later in the Calculator example.

In comparing the use of properties and methods, you should note one crucial difference: *A control **property** can be used in an assignment statement, but a control **method** can never be used in an assignment statement.* So, if you ever see a control followed by a period and name in an assignment statement, the name must refer to a property. Conversely, if you see a control followed by a period and name by itself, then the name must refer to a method.



TIP: Note that when you type the name of an object in your code, an Auto List pop-up menu appears just after typing the period. This menu provides a listing of all properties and methods available for the object that you typed.

Clearing Text Boxes

After we use this calculator for one set of numbers, to use it again we need to clear the text boxes and set the focus back to the first text box. To do this, we need to add code to the btnClear button that sets the Text property of each text box to an empty string ("") and uses the Focus method to position the cursor in the txtFirstNum text box. The code for this button is shown in VB Code Box 7-4.

VB CODE BOX 7-4.
Code for btnClear button

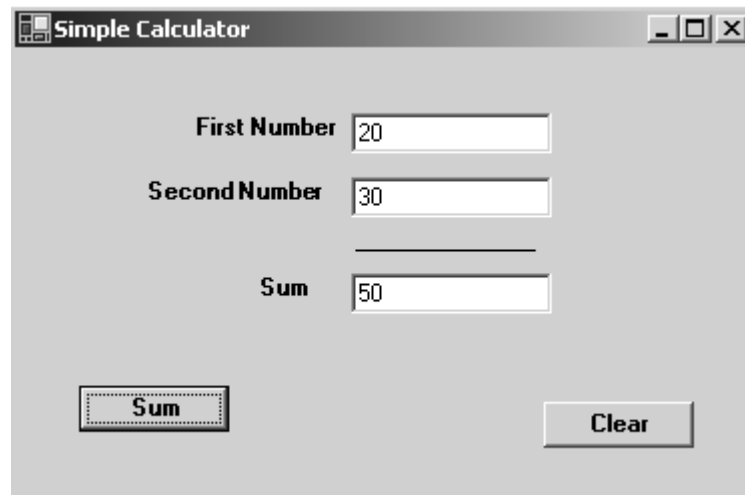
```
Private Sub btnClear_Click(ByVal sender As System.Object,  
ByVal e As System.EventArgs) Handles btnClear.Click  
    txtFirstNum.Text = ""  
    txtSecondNum.Text = ""  
    txtSum.Text = ""  
    txtFirstNum.Focus()  
End Sub
```



TIP: The null string ("") does *not* contain any spaces between the pair of quotation marks.

If 20 and 30 are entered in the top two text boxes of the Calculator project and Sum is clicked, the result is shown in Figure 3-4. If the Clear button is then clicked, all three text boxes will be cleared and the focus will be set back to the top text box. The steps to create the simple calculator project are summarized in Step-by-Step Instructions 3-1.

FIGURE 3-4.
Running the simple
calculator



Step-by-Step Instructions 3-1: Creating a simple calculator

1. In Window Explorer, create a new folder in the Visual Studio Projects folder named **Chapter3**. Then:
 - a. Start VB .NET and select **New Project** from the Start page (or **File | New | Project** if you are already in VB .NET) to create a new project.
 - b. Give the new project a name of **Simple** and make sure it is being created in the Chapter3 folder you just created.
 - c. Change the Text property of the form to **Simple Calculator** and change the Name property of the form to **frmSimple**.
 - d. In the Solutions Window, *right-click* on the *Simple.vbproj* project file and selecting **Properties**. In the resulting Properties page, select **frmSimple** as the Startup Object.
 - e. Finally, use the **File | Save Form1 as ...** menu option to save the form as **Simple.vb**
2. In the Solution Explorer window, *right click* on the **Simple.vbproj** project file (the second one from the top) and select **Properties**. In the Properties dialog box, change the *Startup Object* from **Form1** to **frmSimple**.
3. To the frmSimple form, add three text boxes, four corresponding labels (one for a series of underscores) as shown in Figure 3-2. (You will need to experiment with the number of underscores in the Label control to make it match the width of the text boxes.) Change the font for the labels to **10 point bold**.
4. Add buttons in the lower lefthand and righthand areas of the form. Use the names and texts (where appropriate) shown in Table 3-3 for the controls on this form.
5. Open the Code window for the **btnSum** button and enter the code shown in VB Code Box 7-3.
6. Open the Code window for the **btnClear** button and enter the code shown in VB Code Box 7-4. Change the font for both buttons to be **10 point bold**.

7. Click the **VCR Run** button. Enter **20** in the top text box and **30** in the second text box. Click the **Sum** button. Your result should look like Figure 3-5. Now click the **Clear** button to clear all text boxes and set the focus back to the top text box. Test your project with a few other combinations of Integers, both positive and negative. When these are completed, click the **VCR Stop** button.
8. Note: if you receive an error message like "Sub Main was not found ..." when you run the program, double-click on the error message. A window will be display from which you can form name (**frmSimple**) and click OK to continue. This will occur if you fail to make frmSimple your Startup Object.
9. Click the **Save All** icon to save the files in this project..



Mini-Summary 3-2: Event-driven input

1. Input in a VB .NET program is often event-driven. A value is entered into a text box and then transferred to a variable with an event such as clicking a button.
2. An assignment statement is used to transfer the contents of the text box to a variable and to make arithmetic calculations with the variables.
3. The CInt() function is used to convert the string contents of the text box to an Integer variable. The CStr() function converts numeric variables into strings for storage in text boxes.
4. Control properties can be used in assignment statements, but control methods cannot.
5. Text boxes may be cleared by setting equal to a null or empty string. The Focus method positions the cursor in a text box.



It's Your Turn!

1. Assume the following variable declarations have been made:

```
Dim intN As Integer
Dim sngPi As Single
Dim sngAlpha As Single
```

Determine which of the following are valid VB .NET assignment statements. If they are not valid explain why they are not.

- a. sngPi = 3.141592
- b. intN + 1 = intN
- c. sngAlpha = 1
- d. sngAlpha = sngAlpha
- e. 3 = intN
- f. intN = intN + 1
- g. intN = "Five"

2. For each of the following, what is the value displayed in txtAnswer after the code completes execution (where we have not shown all of the first statement in each event procedure)?

```
a. Private Sub btnCalculate_Click()
    Dim intFirst, intSecond, intAnswer As Integer
```

```

intFirst = 5
intSecond = 3
intAnswer = intFirst + intSecond
txtAnswer.Text = intAnswer
End Sub

```

```

b. Private Sub btnCalculate_Click()
    txtR.Text = CStr(0.07)
    txtS.Text = CStr(4.5)
    txtAnswer.Text = Str(CInt(txtR.Text) + CInt(txtS.Text))
End Sub

```

```

c. Private Sub btnCalculate_Click()
    Dim intA As Integer
    Dim intB As Integer
    Dim intAnswer As Integer
    intAnswer = intA + intB
    txtAnswer.Text = ""
End Sub

```

3. Explain what is wrong with the following statement:

```
txtName.Focus = On
```

4. Complete Step-by-Step Instructions 3-1 to create the Simple Calculator example.

5. Experiment with the use of the Text property by adding a new button to the Simple Calculator example with a Text of **Test** and a Name of **btnTest**. The Code window for this button should have only one line:

```
txtSum.Text = txtFirstNum.Text + txtSecondNum.Text
```

6. Run the project again with values of **20** and **30**. Click the **Test** button. What happens? Note that the two numbers have been concatenated (combine into one string) rather than being summed. Close the project *without* saving the revised version of this project

USING ASSIGNMENT STATEMENTS FOR CALCULATIONS

Now that you have seen and used assignment statements to transfer input and output to and from text boxes and to carry out a simple calculation, you are ready to consider more complex uses of assignment statements. However, in all cases, the same idea as before applies; that is, you always have the variable or control property on the left side of the equal sign and a value, variable, expression, or Control property on the right side. While the term *expression* in our discussion of assignment statements is a common one in algebra, it needs to be defined for programming: An **expression** is a *combination of two or more variables and/or constants or functions with operators*. We already saw one example of an expression when we summed the two variables in the Simple Calculator, that is, `intFirst + intSecond`. In this discussion, we will cover numerous other situations involving expressions.

To understand our definition of an expression, we need to understand two new terms in it: constants and operators. A **constant** is a quantity that does not change. Numbers are constants, as are strings enclosed in quotation marks. For example, 73 and -0.453 are numeric constants and “VB .NET” is a string constant. Constants are used frequently in expressions when the same value applies to all situations. For example, the circumference of a circle is always 3.14157 times the diameter of the circle (where 3.14157 represents Pi).

Operators are symbols used for carrying out processing. The plus sign we used in the Simple Calculator example is an operator. There are four types of operators: arithmetic, concatenation, comparison, and logical. Arithmetic operators are used to carry out arithmetic calculations. Concatenation operators are used to combine String variables and constants. Comparison operators are used to compare variables and constants. Finally, logical operators are used for logical operations. For the time being, we will concentrate on the arithmetic operators shown in Table 3-4.

TABLE 3-4: Arithmetic Operators

Operator	Function	Example	Result
()	Grouping	(A+B)	Groups summation operation
^	Exponentiation	Radius^2	Squares Radius
-	Negation	-Amount	Changes sign of Amount
*	Multiplication	3*Price	Multiplies Price by 3
/	Division	PayRaise/Months	Divides PayRaise by Months
\	Integer Division	Number\3	Performs integer division of Number by 3
Mod	Modulus	15 Mod 2	Remainder from dividing 15 by 2
+	Addition	Price + Taxes	Sums Price and Taxes
-	Subtraction	Salary - Deductions	Subtracts Deductions from Salary

You should already be familiar with all of these operations, with the possible exception of integer division and the use of the Modulus operator. Integer division is differentiated from standard division in that with integer division both the divisor and the dividend are rounded to integers and the quotient is truncated to an integer. For example, if $A = 7.111$ and $B = 1.95$, then $A \setminus B$ will result in 7.111 being rounded to 7, B being rounded to 2, and the quotient of 7 divided by 2 will be truncated to 3. As a result, $7.111 \setminus 1.95$ yields an integer value of 3.

The Modulus operation finds the integer remainder that results from integer division of the two operands. For example, if we use the same two values of A and B as above, $A \text{ MOD } B$ will yield a value of 1 ($7.111 \setminus 1.95 = 7 \setminus 2 = 3$ with remainder of 1)

We can construct arithmetic expressions by combining variables, constants, and arithmetic operators. Examples of valid arithmetic expressions in assignment statements are shown in Table 3-5.

All of the examples in Table 3-5 should be clear, with the possible exception of the examples involving the variable `intCounter`. In these two examples, we first *initialize* the variable to zero and then increment it by one. These two instructions provide two key concepts to remember. First, you should never assume that the value of any variable is automatically zero or anything else. It may retain a value from a previous use, so any variable that will appear later should always be initialized on the *right* side of an assignment statement.

Second, whenever the same variable appears on both sides of an assignment statement, the value of a variable on the *right* side of the equals sign is the *current value* of that variable and the value of the variable on the *left* side of the equals sign is the *new value* of the variable. Having two different values of the same variable in the same

assignment statement does not confuse the computer, because it makes any needed calculations on the right side of the equals sign using current values. It then takes the resulting value and places it in the variable on the left side of the equals sign. This way there is no confusion between old and new values of the same variable

At this point, we need to consider a major syntax error that can occur when in the use of an assignment statement: An expression can never appear on the left side of the equals sign. For example, an *invalid* assignment statement is:

$$X + Y = Z + Q$$

Note that the expression consisting of two variables and an operator on the left of the equals sign violates the rule about expressions on the left side of the equals sign..

TABLE 3-5: Examples of Valid Expressions in Assignment Statements

Expression	Result
<code>decTakeHome = decSalary - decDeductions</code>	TakeHome value is equal to Salary minus Taxes and Deductions.
<code>decInterest = decPrincipal * sngInterestRate</code>	Interest value is equal to Principal times Interest Rate.
<code>sngArea = 3.14157*sngRadius^2</code>	Value of Pi is multiplied by Radius squared to compute area of a circle.
<code>decUnitCost = decTotalCost/intUnits</code>	Unit Cost is equal to Total Cost divided by number of Units.
<code>intRoundUp = intBig\intTiny + intBig Mod intTiny</code>	The remainder of integer division of Big by Tiny is added to the result of integer division of Big by Tiny to compute the Roundup value.
<code>intCounter = 0</code>	The variable Counter is set equal to zero.
<code>intCounter = intCounter + 1</code>	The variable Counter is set equal to <i>old</i> value of Counter plus 1.

The Hierarchy of Operations

An important question may come to mind when you're using arithmetic operators: In what order will the operators be used? For example, consider the following expression in an assignment statement:

$$\text{Cost} = F + V * D + S * D ^ 2$$

Note that there are two summations, two multiplications, and an exponentiation operation. The order in which these operations are carried out will determine the value of the expression, which in turn will be assigned to the variable, Cost, on the left side of the equals sign. However, there is no ambiguity about the order of arithmetic operations in this or any expression on the right side of an assignment statement, because the hierarchy of operations will control the order in which the operations are performed. For VB .NET, the **Hierarchy of Operations** is:

1. Parentheses
2. Raising to a power
3. Change of sign (negation)
4. Multiplication or division

5. Integer division
6. Modulus
7. Addition or subtraction

Note that the arithmetic operators were listed in Table 3-4 in the same order as the hierarchy of operations. In case of a tie, work from left to right.

Returning to our example, since there are no parentheses in the expression $F + V * D + S * D^2$, according to the hierarchy of operations the arithmetic operations will be carried out in the following order:

1. D will be raised to the second power.
2. V will then be multiplied by D, and S will be multiplied by D-squared.
3. F and the two products found in Operation 2 will be summed.

If $F = 100$, $V = 200$, $D = 30$, and $S = 2$, this expression will be evaluated as follows:

1. $D^2 = 900$
2. $V * D = 6,000$ and $S * D^2 = 1,800$
3. $F + V * D + S * D^2 = 100 + 6000 + 1,800 = 7,900$

and the variable Cost will be equal to 7,900.

You should be aware that parentheses can have a dramatic effect on the result of evaluating an expression. For example, consider the same expression as before but with parentheses around $F + V$; that is, the assignment statement is now:

$Cost = (F + V) * D + S * D^2$

In this case, F and V will be summed before being multiplied by D, yielding a result that is very different from the one we got before. Using the same values as before, we now have:

1. $F + V = 300$
2. $D^2 = 900$
3. $(F + V) * D = 9,000$ and $S * D^2 = 1,800$
4. $9,000 + 1,800 = 10,800$

and Cost equals 10,800 instead of 7,900.

String Operators

For String variables and constants, the only valid operation is that of combining two strings into one. This operation, which is performed using the plus sign (+) or the ampersand (&), has the effect of adding the second String variable or constant to the end of the first. For example, if we have the assignment statement `strBigDay = "May" + " Day"` (or `"May" & " Day"`), the result is that `strBigDay` is now equal to "May Day." None of the other operators has any meaning for operations with strings.



TIP: While both the ampersand (&) and plus (+) symbols do the same thing when combining strings, they work differently when used with numeric values. This may cause some confusion if you wish to concatenate two numeric values. It may be best to use the ampersand (&) only for concatenation.

Symbolic Constants

In creating an application in VB .NET or any other language, you never work entirely with variables only. Often you are working with constants in expressions or by themselves. While it is possible to use the actual Numeric constant or String constant

(enclosed in quotation marks), if a quantity is not going to change in your project it is advisable to give this quantity a name and data type just like you do for variables. For example, if you are working with an interest rate that is going to remain the same throughout your program at say, 0.07, you might want to give this value a name of `sngIntRate`, define it as a `Single` data type, and use it, rather than the actual value, in your processing. If you decide at a later time that you want to change the interest rate from its current value, you simply change it at the point in your project where you have named it. The same rules and conventions apply to names for constants as apply to names for variables. Named constants are often referred to as **symbolic constants** since you are using a symbolic name for the actual value or string.

Assigning a name to a constant is usually done at the beginning of an event procedure—even before you declare variables. The form of the constant definition statement is much the same as initializing a variable at the same time as declaring it:

Const *constant name as variable type = value or expression*

where you supply both the constant name and value portions of the statement. For example, to create a symbolic constant called `IntRate` with a value of 0.07, the statement would be:

```
Const sngIntRate as Single = 0.07
```

and if you also wanted to define a constant for the number of years in the investment as being 12, the statement would be:

```
Const intNumYears as Integer = 12
```

As with declaring variables, it is possible to define more than one symbolic constant on a line by separating them with commas. For example, the two constant definitions shown above could be combined as:

```
Const sngIntRate as Single = 0.07, intNumYears as Integer = 12
```

While the constant and variable declarations look quite similar, they have very different results; a constant cannot be changed in the code while a variable is *meant* to be changed.

Mini-Summary 3-3: Using assignment statements for calculations

1. Assignment statements are used for calculations by setting variables equal to expressions where an expression is a combination of variables, constants, values from functions, and operators.
2. Arithmetic operators include grouping, exponentiation, negation, multiplication, division, integer division, modulus, addition, and subtraction. The Hierarchy of Operations controls the order in which these operations are carried out.
3. String operators include the plus sign and ampersand for concatenation.
4. Symbolic constants can be defined at the beginning of the program to store values that will not change during the program.



It's Your Turn!

1. Write appropriate assignment statements for the following situations:
 - a. The total cost for the sale of multiple items is the unit price times the number of units sold.

- b. The net sales price after applying a discount is equal to the gross sales price times $(1 - \text{discount rate})$. For example, if the gross price is \$500 and the discount rate is 0.15, the net price is equal to $500 \times (1 - 0.15)$.
 - c. The value of an amount of money some number of years in the future is equal to the amount of money times $(1 + \text{rate of return})$ raised to the number-of-years power. For example, if the amount of money is \$1,000, the rate of return is 0.12, and the number of years is 5, the future value is equal to $1000 \times (1 + 0.12)^5$.
 - d. The depreciated value of a piece of machinery using straight line depreciation is equal to the original value minus the depreciation, where the depreciation is equal to the original value divided by the life of the machinery times the number of years since it was put into service.
 - e. The amount due for a sale is equal to the sales price times $(1 + \text{tax rate})$.
2. Evaluate the following expressions to determine the value assigned to the variable on the left of the equals sign.
 - a. $Y = 3^2 * 4 - 1 * 2 + 3$
 - b. $X = 3^{(2 * 4)} - 1 * (2 + 3)$
 - c. $\text{sngAverage} = ((70 + 80) / 2 + 65) / 2$
 - d. $\text{strState} = \text{"New York"}$
 - e. $\text{strCity} = \text{strState} \ \& \ \text{" City"}$
 3. Declare constants for the following values:
 - a. Pi (3.14157)
 - b. The current exchange rate between British Pounds and U.S. Dollars (1.62)
 - c. The number of feet in a mile (5,280)
 4. Given that $\text{sngTwo} = 2.0$, $\text{sngThree} = 3.0$, $\text{sngFour} = 4.0$, $\text{intNum} = 8$, and $\text{intMix} = 5$ and that the appropriate variable declarations have been made, find the value assigned to the given variable for each of the following.
 - a. $\text{sngW} = (\text{sngTwo} + \text{sngThree}) \wedge \text{sngThree}$
 - b. $\text{sngX} = (\text{sngThree} + \text{sngTwo} / \text{sngFour}) \wedge 2$
 - c. $\text{sngY} = \text{intNum} / \text{intMix} + 5.1$
 - d. $\text{intZ} = \text{intNum} / \text{intMix} + 5.1$
 5. Write variable declaration statements and assignment statements for the following that calculates the given expression and assigns the results to the specified value.
 - a. Rate times Time to DIST
 - b. Square root of $(A^2 + B^2)$ to C
 - c. $1 / (1/R1 + 1/R2 + 1/R3)$ to Resist
 - d. P times $(1 + R)^N$ to Value
 - e. Area of triangle (one-half base times height) of base B and height H to Area
-

APPLICATION TO VINTAGE DVDS



So far you have learned about text boxes, assignment statements, and expressions. We are now ready to apply them to the Vintage DVDS example using the six-step development process presented in Chapter 1; that is, define problem, create interface, develop logic for action objects, write and test code for action objects, test overall project, and document project in writing.

Define Problem

Assume that the Vintage DVDs store owner wants to extend the project created in Chapter 2 to input the renter's name, the DVD rented, and the price for the DVD and then use this information to calculate the taxes due on the rental price and add these taxes to the rental price to compute the amount due. He also wants to have a way of printing the result of the computations, clearing the text boxes, and exiting the project. To ensure that we understand his request, we need to sketch the interface. In this case, this involves adding additional features to the sketch created in Chapter 2 (Figure 2-8). The resulting sketch is shown in Figure 3-5.

FIGURE 3-5.
Revised Vintage
DVDs sketch

Create Interface

Note that, in the sketch in Figure 3-5, that five text boxes for input and output and five corresponding labels have been added to the form along with a line control to separate the amount due text box from the others. Three additional buttons—to calculate the taxes and amount due, to clear the text boxes, and to exit the project—are required in addition to the existing button that displays a welcoming message. No other controls are required for this project. To add new controls to the existing Vintage form, we must first create the Vintage3 project by copying the Vintage2 project files to a new folder. The steps to do this are summarized in the Step-by-Step Instructions 3-2.



Step-by-Step 3-2: Modifying an Existing Project

1. Start Windows Explorer and create a new folder within the **Chapter3** folder (or wherever you have been instructed to save your projects) named **Vintage3**.
2. **Copy** all the files in the **Chapter2\Vintage2** folder to the new **Chapter3\Vintage3** folder. Caution: do not drag them from the Vintage 2 folder as that moves them instead of copying them. Use the **Edit|Copy** menu command to copy them from Vintage2 folder and the **Edit|Paste** command to paste them to the Vintage3 folder.

3. Start VB .NET and select **Open Project** from the Start Page (if you are already in VB .NET, select **File | Open | Project** menu option.) Now *double-click* the **Vintage3** folder you created in Step 1 to open it. From the resulting dialog box, open the *Vintage.sln* file by *double-clicking* it also. (it should already be highlighted.)
4. You should now see the Form Design window with the Solution Explorer Window showing the Vintage2 files you copied into this folder, *Vintage.sln*, *Vintage.vbproj*, and *Vintage.vb*.
5. In the Solution Explorer window, *right-click* the **Vintage.vbproj** project file and choose **Properties**. The Vintage Property Pages dialog box will be displayed. Select **frmVintage** from the Startup Object listbox and click **Ok** to save this choice.
6. Double-click the **Vintage.vb** file in the Solution Explorer window to display the *frmVintage* form and make it necessary changes. .

Once you have completed the Step-by-Step 3-2 instructions to create the new Vintage3 folder with the Vintage3 files in it, you are now ready to add the new controls that are shown in Table 3-6

TABLE 3-6: New Controls for Vintage DVDs Application

Control	Name Property	Text Property
text box	txtCustName	
text box	txtDVDName	
text box	txtDVDPrice	
text box	txtTaxes	
text box	txtAmountDue	
label	lblCustName	Customer Name
label	lblDVDName	DVD Name
label	lblDVDPrice	DVD Price
label	lblTaxes	Taxes
label	lblAmountDue	Amount Due
label	lblLine	_____
button	btnCalc	Calculate
button	btnClear	Clear
button	btnExit	Exit

Once all of the controls shown in Table 3-6 are added to the existing Vintage3 project, it should appear as shown in Figure 3-6 with input text boxes and related labels and three new buttons to calculate the total amount due, clear the input, and to exit the application..



*Develop Logic for
Action Objects*

On the revised Vintage DVDs form, there are three new action objects: the Calculate, Clear, and Exit buttons. Of these, the Calculate button is the only one that involves input, processing, and output. As such, it is the only object for which we need to

FIGURE 3-6.
Expanded interface
for Vintage DVDs



develop the logic using an IPO Table and pseudocode. The Clear button clears the text boxes and sets the focus back to the Customer Name text box. The Exit button involve only single instructions to exit the application.

Input to the btnCalc button includes the customer name, DVD name, and DVD price. Processing includes computing the tax on the DVD price and adding it to the DVD price to determine the total amount due. Output should include the tax and the total amount due. The IPO Table for the Calculate button is shown in Figure 3-7.

FIGURE 3-7. IPO
Table for Calculate
button

Input	Processing	Output
Customer name	Taxes = 0.07 x DVD Price	Taxes
DVD Name	Amount Due = Price + Taxes	Amount Due
DVD Price		Amount Due

The pseudocode for the Calculate button shown below converts the IPO into structured English. Note that there is no mention of text boxes in the input and output statements in the pseudocode, because the pseudocode is intended to present the logic of the operation with no concern for the details.

```

Begin Procedure Calculate
    Input customer name
    Input DVD name
    Input DVD price
    Taxes = DVD price times tax rate
    Amount due = DVD price + taxes
    Output taxes
    Output amount due
End procedure
    
```

Write and Test Code

As noted earlier, there are three new action objects for which we need to write code: the click events for the Calculation, Clear, and Exit buttons. The code for the Calculation button should follow the pseudocode, except that it uses text boxes for input and output. If we apply the four-step process for writing code to this problem, the resulting code is as shown in VB Code Box 7-5.

VB CODE BOX 7-5. Code for Calculate button	<pre>Private Sub btnCalc_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles btnCalc.Click Const sngTaxRate As Single = 0.07 'Use local tax rate Dim decPrice, decAmountDue, decTaxes As Decimal decPrice = CDec(txtDVDPrice.Text) decTaxes = decPrice * sngTaxRate 'Compute taxes decAmountDue = decPrice + decTaxes 'Compute amount due txtTaxes.Text = CStr(decTaxes) txtAmountDue.Text = CStr(decAmountDue) End Sub</pre>
--	---

Note in the code that a symbolic constant, `sngTaxRate`, is declared to be equal to the local sales tax rate (0.07). If the sales tax rate changes or the application is used in another jurisdiction, we can easily change the sales tax rate by changing this statement. Since the tax rate does not normally change from use to use, we declare it as a constant rather than inputting it for each use. Note also that the variables are declared as `Decimal` rather than `Single`, since the values stored in them will be in dollars and cents. Also, the `CDec` function is used to convert between the `String` data type and the `Decimal` data type, and the `CStr()` function is used in the other direction. Some of the more commonly used conversion functions are shown in Table 3-7.

TABLE 3-7: Commonly Used Conversion Functions

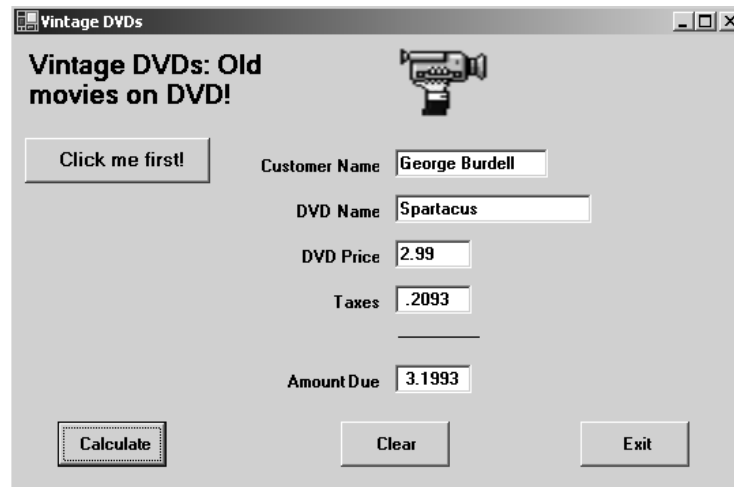
Conversion Function	Purpose
<code>CStr</code>	Converts argument to <code>String</code> data type
<code>CBool</code>	Converts argument to <code>Boolean</code> data type
<code>CDec</code>	Converts argument to <code>Decimal</code> data type
<code>CDate</code>	Converts argument to <code>Date</code> data type
<code>CInt</code>	Converts argument to <code>Integer</code> data type
<code>CSng</code>	Converts argument to <code>Single</code> data type
<code>Cdbl</code>	Converts argument to <code>Double</code> data type

Note that in VB Code Box 7-5 we have also added a **comment** to the statement defining the tax rate by beginning it with an apostrophe. Comments are a form of **internal documentation** that is used to explain part of a program. Comments can be on a line by themselves or added to the end of another statement as was done here. Comments are displayed in green on the screen to distinguish them from executable code. Any text begun with an apostrophe or the keyword *Rem* is ignored by the computer and is there for explanation purposes only. You should include comments in your code wherever it will help explain the purpose of the program or a specific statement. As we noted in Chapter 1, since we explain all code in the text, we will add comments here

only to explain code that might otherwise be misunderstood. Your instructor may want you to add more comments, and, almost certainly, if you do any coding in your professional life, comments will be *required*.

Figure 3-8 shows an example of running the project with sample data (Customer Name = *George Burdell*, DVD Name = *Spartacus*, and DVD Price = *2.99*) and clicking the Calculate button. The resulting Taxes value is 0.2093 and the Amount Due is 3.1993.

FIGURE 3-8. Vintage DVDs application



Formatting Output

Note that the values output in the Taxes and Amount Due text boxes are shown not as dollars and cents but with four decimal places. To control the form of *numeric* items or dates in output, we need to use the **Format(expression, format) function**. In this function, *expression* is any valid expression that is to be formatted and *format* is a valid **format expression**. The more commonly used Numeric format expressions are shown in Table 3-8, where each format expression is enclosed in quotation marks in the Format function. Note that the Format function has no effect on non-numeric variables.

For example, to format a number as currency with two decimal places, the format expression would be “currency”; to format it as percent, the expression would be “percent.” These formats can also be abbreviated as “c” for currency or “p” for percent.



TIP: If you do not include the format expression, the Format() function will return the same result as the CStr() function.

In the Vintage DVDs project, we will format the txtTaxes text box as currency by replacing the statement involving the CStr() function with a statement using the Format() function, as shown below:

```
txtTaxes.text = Format(decTaxes, "currency")
```

Similarly, to format the txtAmountDue text box as currency, the statement would be:

```
txtAmountDue.text = Format(decAmountDue, "C")
```

It is also possible to format the numeric values using the FormatCurrency function. The **FormatCurrency** function performs the same operation as the Format



TABLE 3-8: Numeric Format Expressions

Format Expression	Abbreviation	Result
Currency	“c” or “C”	Display number with dollar sign, thousands separator, and two digits to the right of the decimal point.
Fixed	“f” or “F”	Display number with at least one digit to the left and two digits to the right of the decimal point.
Standard	“s” or “S”	Display number with thousands separator and at least one digit to the left and two digits to the right of the decimal point.
Percent	“p” or “P”	Display number multiplied by 100 with a percent sign (%) on the right and two digits to the right of the decimal point.
Scientific	“e” or “E”	Use standard scientific notation.

function with the “currency” or “c” parameter. For example, the input textbox can be formatted by including the same text box on the left side of the assignment statement and on the right side in the FormatCurrency function. To format the contents of txtDVDPrice as currency, the statement might be:

```
txtDVDPrice.Text = FormatCurrency(txtDVDPrice.Text)
```

If we replace the two statements in the btnCalc button Code window that use the CStr() function with statements that use the Format() function and we then add a statement that formats the txtDVDPrice textbox, the final code for this object will appear as shown in VB Code Box 7-6. If we run the revised project with the same sample data as before, the new result will be as shown in Figure 3-9. Note that all monetary values are now shown as currency.

<p>VB CODE BOX 7-6. Code to compute and display taxes and amount due</p>	<pre>Private Sub btnCalc_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles btnCalc.Click Const sngTaxRate As Single = 0.07 'Use local tax rate Dim decPrice, decAmountDue, decTaxes As Decimal decPrice = CDec(txtDVDPrice.Text) decTaxes = decPrice * sngTaxRate 'Compute taxes decAmountDue = decPrice + decTaxes 'Compute amount due txtTaxes.Text = Format(decTaxes, "c") txtAmountDue.Text = Format(decAmountDue, "c") txtDVDPrice.Text = Format(decPrice, "c") End Sub</pre>
---	--

Clearing entries and exiting the application

In addition to computing the taxes and amount due on a DVD rental, we want to make it possible to clear the previous entries in preparation for the next customer, set the focus to the customer name text box, and exit the project. This means that the Text property of the text boxes must be set to an empty string, as was done with the Simple Calculator example, and the Focus method used to place the cursor in the txtCustName text box so the next name can be entered. The complete code for the btnClear button is shown in VB Code Box 7-7. When the btnClear is clicked, all text boxes are blanked out and the focus set to txtCustName.

FIGURE 3-9. Result of running revised project

The code for the Exit button is very simple; it consists of one word: **End**. This will cause the project to terminate, just as if you had clicked the VCR Stop button. The steps to create this project are summarized in Step-by-Step Instructions 3-3.

<p>VB CODE BOX 7-7. Code to clear entries</p>	<pre>Private Sub btnClear_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles btnClear.Click txtCustName.Text = "" txtDVDName.Text = "" txtDVDPrice.Text = "" txtTaxes.Text = "" txtAmountDue.Text = "" txtCustName.Focus() End Sub</pre>
--	---



Step-by-Step Instructions 3-3: Adding controls to the Vintage DVDs Project

1. Open the **Vintage3** project that resulted from carrying out the Step-by-Step 3-2 instructions and add the five text boxes, five labels, four buttons, and one line control as shown in Figure 3-5.
2. Change the Name properties for all of the new controls to match those shown in Table 3-6. Delete the Text property for the text boxes and add the appropriate texts to the label and button controls. Your resulting form should look like that shown in Figure 3-6.
3. Add the code shown in VB Code Box 7-5 to the Code window for the Click event for the btnCalc button. Add the End statement to the Code window for the Click event for the btnExit button.
4. Run the project and test the **Calculate** button by entering a customer name of **George Burdell**, a DVD name of **Spartacus**, and a DVD price of **\$2.99** (don't include the dollar sign.) The result should be the same as shown in Figure 3-8.

5. Change the statements that use the CStr() function to use the Format() function with the Currency ("c") format expression. Also, format the txtDVDPrice text box with the FormatCurrency function. Your code should be the same as that shown in VB Code Box 7-6. Run your project again with the same data as above. Your result should look like Figure 3-9.
6. Open the Code window for the Clear button and add the code shown in VB Code Box 7-7.
7. Exit the project and save the files.



Mini-Summary 3-4: Developing the Vintage DVDs Application

1. The Vintage DVDs application can be developed using the six-step process discussed in Chapter 1. The code is written using the four-step process discussed earlier in this chapter.
2. IPO Tables and pseudocode are useful for developing the logic of the code.
3. In addition to the Val function, there are a number of other conversion functions that convert string data to a specific numeric data type.
4. To control the form of output, the Format function should be used with a variety of format expressions.
5. The application can be exited with the End command.



It's Your Turn!

1. Describe what appears in the txtAnswer textbox after each of the following is executed. If you are not sure, create a form and write the code in VB .NET to find out.

- a.

```
Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 56.789
    txtAnswer.Text = Format(sngValue, "c")
End Sub
```
- b.

```
Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 456.7891
    txtAnswer.Text = Format(sngValue, "f")
End Sub
```
- c.

```
Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 65.432
    txtAnswer.Text = Format(sngValue, "s")
End Sub
```
- d.

```
Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 1.543
    txtAnswer.Text = Format(sngValue, "p")
End Sub
```

```
e. Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 0.00235
    txtAnswer.Text = Format(sngValue, "e")
End Sub
```

2. Describe what appears in the txtAnswer textbox after each of the following is executed. If you are not sure, create a form and write the code in VB .NET to find out.

```
a. Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 0
    txtAnswer.Text = CBool(sngValue)
End Sub
```

```
b. Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 3.6
    txtAnswer.Text = CInt(sngValue)
End Sub
```

```
c. Private Sub btnCalculate_Click()
    Dim sngValue As Single
    sngValue = 36038
    txtAnswer.Text = CDate(sngValue)
End Sub
```

3. Complete Step-by-Step Instructions 3-2 to modify the file you created in Chapter 2 (Vintage2) to become Vintage3.

4. Complete Step-by-Step Instructions 3-3 create the Vintage3 project.

5. Run the project again with the previous customer and DVD (**George Burdell**, a DVD name of ***Spartacus***, and a DVD price of **\$2.99**.) Clear the entries and enter a new customer name of **Sam Cassell**, a DVD name of ***Sands of Iwo Jima***, and a price of **\$1.99** and calculate the result. Clear the form and add new data of your choosing and click Calculate again.

6. Save the project files and close it.

MORE ON USING FUNCTIONS

You have already used four built-in functions—CStr(), CInt, CDec(), and Format()—in creating the Simple Calculator and the Vintage DVDs application, so you have some knowledge of their use. In addition to these four, there are many other built-in functions that serve a variety of useful purposes. There are built-in functions for converting data types, working with dates and time, performing financial operations, working with strings, and carrying out different mathematical operations. Examples of commonly used built-in functions are shown in Table 3-9. Check the online Help for others. You will probably recognize many of them from working with spreadsheet software.

As an example of using one of the financial functions, we will create an application that will determine the monthly payment necessary to repay a loan. In looking at Table 3-9, you can see that we will need to use the Pmt() function for this purpose. The Pmt function has the following form: **Pmt(rate, nper, pv)**, where **rate** = the periodic interest rate as a decimal fraction, **nper** = the number of months over which the loan is to be repaid, and **pv** = the *negative* of the loan amount. For example, if you

TABLE 3-9: Commonly used built-in functions

Function	Type	Purpose
Abs	Mathematical	Returns absolute value of a number
Sqr	Mathematical	Returns square root of a positive number
FV	Financial	Returns future value of an annuity
PV	Financial	Returns present value of an annuity
IRR	Financial	Returns internal rate of return
Pmt	Financial	Returns periodic payment to pay off a loan
UCase/LCase	String	Converts a string to all upper/lower case
Len	String	Returns length of string
Datevalue	Date/Time	Returns date for string argument

borrowed \$10,000 at a 12% annual interest rate for five years, the appropriate form of the function would be: `Pmt(.01, 60, -10000)`, where we have changed the 12% annual rate to a 1% (0.01) monthly rate and have converted the number of years to 60 months. It is important that the number of periods used correspond to the interest rate being used.



TIP: Always be sure to understand completely how a function should be used, what parameters are required or optional and what it returns before using it in your code. Functions with similar names do not necessarily operate in the same way.

Creating the Interface

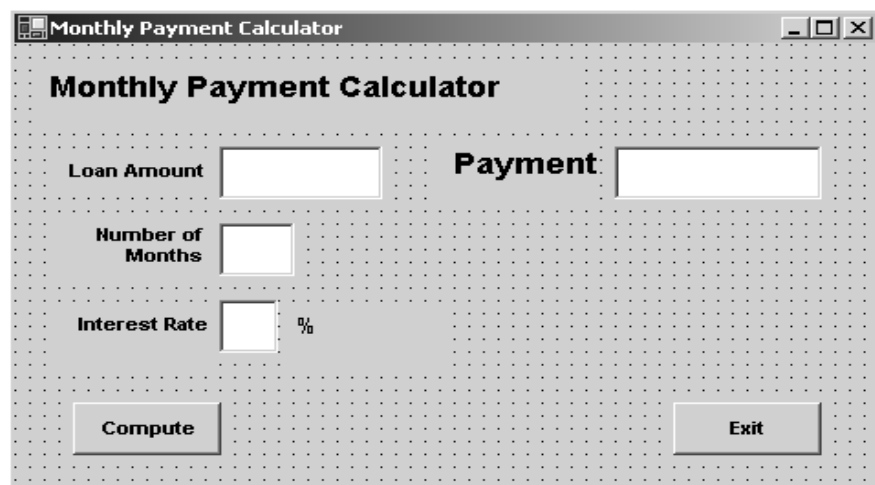
To create the interface to compute the monthly payment required to repay a loan, we need to add the three input text boxes for the amount of the loan, the duration of the loan in months, and the interest rate as a percentage. We will also need an output text box for the monthly payment and two buttons, one to compute the monthly payment and one to exit the project. All of the controls for this interface are shown in Table 3-10 and the resulting form is shown in Figure 3-10.

TABLE 3-10: Controls for Monthly Payment Interface

Control	Name	Text	Font/Other
Form	frmMonthPay	Monthly Payment Calculator	Change filename to Monthpay.vb
Label	lblHeading	Monthly Payment Calculator	14-point Bold
Label	lblLoanAmt	Loan Amount	Bold
Label	lblNumMonths	Number of Months	Bold
Label	lblIntRate	Interest Rate	Bold
Label	lblPercentSign	%	Bold
Label	lblPayment	Payment	14-point Bold

TABLE 3-10: Controls for Monthly Payment Interface (Continued)

Control	Name	Text	Font/Other
TextBox	txtAmount	N/A	
TextBox	txtMonths	N/A	
TextBox	txtRate	N/A	
TextBox	txtPayment	N/A	14-point Bold
button	btnCompute	Compute	Bold
button	btnExit	Exit	Bold

FIGURE 3-10.
Interface for
Payment Calculator

Note that we have added a label with a percentage symbol after the txtRate text box. Doing this will allow us to enter the interest rate as a whole number without formatting it. We do this because formatting a number as a percent precludes using it later in calculations.

Computing the Monthly Payment

For this project, we need to develop the logic for only one action object—*btnCompute*. For this control, the input includes the loan amount, number of months, and interest rate. The processing involves using the **Pmt function** to generate the required monthly payment, which is then output. Since this is so straightforward, we will dispense with the IPO Table and pseudocode for this situation.

For the btnCompute button, the code to compute the monthly payment is very simple: Transfer the contents of the three text boxes to variables, use the variables in the Pmt() function to determine a payment value, transfer the result of the payment value to the txtPayment text box, and format all text boxes appropriately. For the interest rate, we need to convert the contents of the txtRate textbox to a Single data type variable and divide it by 100 to convert it to a decimal fraction. We then have to divide the result by 12 to convert it into a *monthly* interest rate. With the CSng() conversion function, this can be done in one statement (where Rate is the variable):

```
sngRate = (CSng(txtRate.Text) / 100) / 12
```

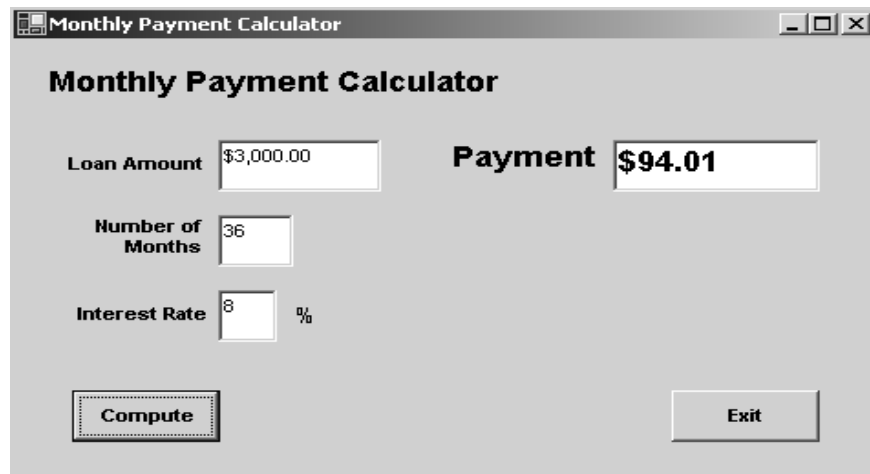
The code for the btnCompute button is shown in VB Code Box 7-8.

<p>VB CODE BOX 7-8. Code to compute monthly payment</p>	<pre>Private Sub btnCompute_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles btnCompute.Click Dim decAmount, decPayment As Decimal Dim intMonths As Integer Dim sngRate As Single decAmount = CDec(txtAmount.Text) intMonths = CInt(txtMonths.Text) sngRate = (CSng(txtRate.Text) / 100) / 12 decPayment = Pmt(sngRate, intMonths, -decAmount) txtPayment.Text = Format(decPayment, "currency") txtAmount.Text = Format(decAmount, "currency") End Sub</pre>
--	---

Several things are of note in VB Code Box 7-8. First, as in the Vintage DVDs application, we declare the `decAmount` and `decPayment` variables to be `Decimal` instead of `Single`. The `intMonths` variable is declared to be `Integer` and `sngRate` is declared to be `Single` since they are not dollars and cents. Second, we use the `CDec()`, `CInt()`, and `CSng()` conversion functions to convert the string data into the appropriate type of variable. Finally, we have formatted the contents of the `txtAmount` and `txtPayment` text boxes as `Currency`. However, we have not formatted the contents of the `txtRate` text box since we want it to remain as an *annual* interest rate instead of being displayed as a monthly interest rate. As mentioned earlier, because of the way the `Percent Numeric` format expression works, if the `txtRate` text box had been formatted as `Percent` *before* the calculation or for a new set of values with the same interest rate, it would *not* have been possible to convert the formatted result back to a numeric value and use it in the calculation.

To test the `btnCompute` button, we will run the project and enter values for the loan amount, number of months, and interest rate, and then click the `btnCompute` button. For example, for a loan of \$3,000 for 36 months at 8 percent, the monthly payment is \$256.03. The result of using this data is shown in Figure 3-12.

FIGURE 3-11.
Monthly Payment Calculator



The steps to create the monthly payment project are summarized in Step-by-Step Instructions 3-4.



Step-by-Step Instructions 3-4: Creating a Monthly Payment Calculator

1. Use **Start Page | New Project** or **File | New Project** to start a new project with a name of **MonthPay** in the **Chapter3** folder. Change the filename for the form to **MonthPay.vb** and give it name of **frmMonthPay**. *Right-Click* on the **MonthPay.vbproj** file and select **Properties**. In this dialog box, change the Startup Object to **frmMonthPay**.
2. For this project, add the controls shown in Table 3-10 to create the interface shown in Figure 3-10.
3. Open the Code window for the btnCompute button and add the code shown in VB Code Box 7-8. Also, add the appropriate code for the btnExit button.
4. Test your project for a loan amount of **\$3,000** for **36** months at **8%** interest rate. The results should look like that shown in Figure 3-11. Test it also for a loan amount of **\$10,000** for **60** months at **7.3%** (your answer should be \$199.43.)
5. If your project runs correctly, save all the files. If there are errors, correct them and then save the files.

Mini-Summary 3-6: More on using functions

1. In addition to the conversion functions (CInt, CStr, and so on), there are many other functions.
2. These include scientific, mathematical, financial, string, and date/time functions.
3. The Pmt function is useful for computing the monthly payment on a loan amount.



It's Your Turn!

1. Describe what is displayed in the text box after the following assignment statements are executed.
 - a. `txtAnswer.Text = Abs(-3)`
 - b. `txtAnswer.Text = Sqr(25)`
 - c. `txtAnswer.Text = UCase("VB is fun")`
 - d. `txtAnswer.Text = Len("VB is fun")`
 - e. `txtAnswer.Text = Sqr(Len("four"))`
2. Complete Step-by-Step Instructions 3-4 to create the monthly payment calculator in VB .NET.

ERRORS IN VB .NET

In creating the three projects discussed in this chapter, you may have encountered an error message from VB .NET, or, worse than that, you received no message but your project failed to run correctly. Ideally, in either of those cases, you were able to find your error by comparing your work to the descriptions in the text. Unfortunately, when you begin creating projects with no explicit instructions to follow, you are likely to run into a variety of errors, or as they are commonly known in the programming community, **bugs**. (There is an apocryphal story about the source of this term, which we won't bore you with here!)

Finding errors is a very important part of program development that often takes as long as all the other steps combined, even when special testing software is used to detect errors. Writing bug-free software is inherently difficult, because the logic supporting the program is inflexible. In most engineering projects, a margin of error is built into the design specifications, so a bridge, for example, usually will not collapse if an element is defective or fails. With computer software, on the other hand, *each* program instruction must be absolutely correct. Otherwise, the whole program may fail. This is a significant problem given the ever-increasing complexity of modern programs. For example, between 1983 and 1992, the average size of a typical application software package increased tenfold, from 100,000 lines of computer code to 1 million lines. Today, software like Windows XP have tens of millions of lines of code in them.

Other errors that can occur during execution include the incorrect use of data or the inadvertent request by the user that the computer perform a meaningless operation, for example, dividing by zero. The code for an object will execute until it encounters an error; then it will stop, display a message telling why it has abnormally terminated the program execution, and highlight the line of code that may be causing the error. If possible, test data that can test all portions of the code should be chosen. If this is not done, errors in any untested section of the project will not be discovered. If an error *is* detected, then the programmer must trace through both the logic of the code and the actual language statements to find it. If a logic error goes *undetected*, the results can be catastrophic.

In general, there are three types of errors that you may encounter in creating VB .NET projects: syntax errors, run time errors, and logic errors. **Syntax errors** are usually caused by incorrect grammar, vocabulary, or usage. Using a keyword for a variable name, using a keyword incorrectly, and entering an assignment statement with an expression on the left of the equals sign are all examples of syntax errors. Fortunately, VB .NET will catch almost all syntax errors when they are entered. For example, if you tried to use the Dim keyword twice in the same declaration statement, VB .NET would immediately alert you to this error when you attempted to press **ENTER** at the end of the line by underlining the error(s) with "wavy underline(s)." If you then position the cursor over the underlined error, a tooltip will show you the error; "Specifiers valid only at beginning of declaration." in this case as shown in Figure 3-13. You would receive a similar error message if you tried to use a variable name beginning with a number or a variable name with a period embedded in it. Occasionally, VB .NET will not catch your syntax error until you attempt to run the program, but the result is the same: You must correct the syntax before proceeding. The best way to avoid these errors is to adhere to the rules regarding variable names, use of keywords, and appropriate use of assignment statements and operators in expressions.

Run time errors occur when the program is running, and they are almost always associated with an error in programmer logic or input data. For example, if you have a project that requires a division operation and either the data or the logic requires a

FIGURE 3-12.
Incorrect use of
keyword

```
Public Class Form1
    Inherits System.Windows.Forms.Form

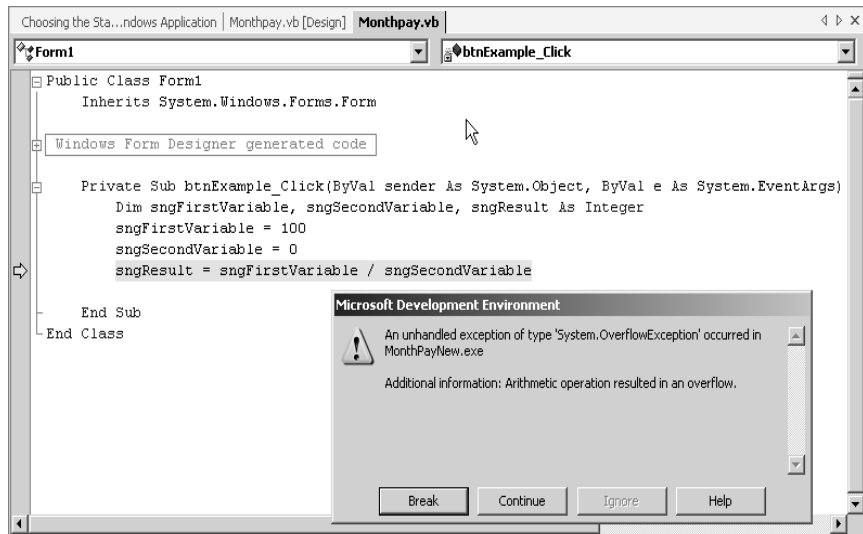
    Windows Form Designer generated code

    Private Sub btnExample_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Dim FirstVariable as Integer, Dim SecondVariable as String
        |
    End Sub
End Class
```

Specifiers valid only at the beginning of a declaration.

division by zero, a run time error will occur. When this occurs, VB .NET will terminate the execution of the project, display the Code window with the line in which the error most likely occurred highlighted, and show a dialog box with a message about the error. At this point, you must stop the program with the Stop icon and correct the error if it involves a problem with the program statement. You may then click the Run icon, and begin running the program again. If the error is one of input (long experience has shown that a large percentage of all errors are errors of input), then you must terminate execution of the project and enter correct data. Even with VB .NET's help, run time errors are more difficult to find and correct than syntax errors. For example, dividing by zero will not be caught until Run time and will result in a runtime error as shown in Figure 3-13 ("overflow" indicates that a memory location has filled up beyond its capacity and this event occurs when division by zero is attempted.)

FIGURE 3-13. Result
of dividing by zero





To avoid runtime errors, programmers will often include statements to trap errors to validate input. For example, a statement will be included that will test a divisor to determine if it is not equal to zero before it gets used in an expression

TIP: When error messages appear while you are testing your program, don't simply press End and ignore them. The message will provide valuable clues as to a problem in your code and possibly how to fix it. Selecting Debug will show the line in which your coding error first causes a problem.

Logic errors are errors that result from incorrect program design, and they usually are not caught until the program has been tested extensively or is already in use. The much-discussed Year 2000 bug was a good example of a logic error. This error was caused by the failure of designers of yesterday's programmers to consider the fact that the year 2000 would eventually arrive. Instead, the designers assumed that all date arithmetic could be handled by using just the last two digits of any year. For example, to determine the year a loan would mature, simply add the number of years for the term of the loan to the last two digits of the current year. When it was 1998 and the term of the loan was, say, five years, the maturity date became 03 causing confusion for the computer about whether this referred to 1903 or 2003. While using only the last two years of the date saved a great deal of disk space at a time when disk space was scarce, it caused a great deal of expense for organizations around the world, not to mention grief for those programmers who were charged with correcting the logic error.

Debugging Projects

Debugging is the art and science of finding run time and logic errors in computer programs. We don't include syntax errors in this search, because VB .NET will find almost all of them for you. In debugging a project, you should start by testing each object as it is created, giving special attention to the input statements. Be sure to use test data that will check all possible options in an object; this will catch many of the errors. To help in this process, VB .NET provides a group of debugging tools, which we will discuss in later chapters. Finally, when all else fails, ask someone else to look at your code. Many times fresh eyes can find errors that have evaded you for hours!

Mini-Summary 3-7: Errors in VB .NET

1. There are three major types of programming errors in VB .NET: syntax errors, run time errors, and logic errors. Programming errors are commonly referred to as *bugs*.
2. Syntax errors are usually found immediately by VB .NET. Run time errors are also often caught by VB .NET when the program is running. However, logic errors are not found by VB .NET and must be found by the programmer.
3. The process of finding and removing errors is referred to as *debugging*.

SUMMARY

At the beginning of the chapter, we said you would be able to do a number of things after reading it. Let's review those things here:

1. **Understand a four-step process to writing code in VB .NET and its relationship to the six operations a computer can perform.** To carry out any of the six operations that a computer can perform, it is necessary to write instructions or code. A four-step process for doing this includes determining the variables needed, input data, process it, and output the results.
2. **Declare and use variables to store the different types of data needed in a computer program.** In terms of storing data in internal memory, variables are used to represent data that are stored in internal memory and come in many times including integers, strings, decimal, single precision, and so on. In VB .NET, unless you turn Option Explicit Off, you must declare all variables as to their type. One way to declare variables is the Dim statement.
3. **Use text boxes for event-driven input and output.** Event-driven input involves inputting data when an event occurs, such as mouse click of a button. Often text boxes are used for event-driven input. In this type of input, the user enters data in a text box and then uses an event to transfer the data to a variable using an assignment statement. The variable is then used in the processing step. It is often necessary to use functions such as CDec() or CInt() to convert from the string form of a textbox to the numeric variables used in processing. The unformatted results of processing can output using the CStr function or formatted through the use of the Format() function. For formatting numbers there are a number of commonly used format expressions, including Currency, Fixed, Standard, Percent, and Scientific.
4. **Describe the process of carrying out arithmetic operations.** Assignment statements are used for performing calculations. Arithmetic and string operators are used to create expressions which can be on the right side of an assignment statement with a variable on the left. The hierarchy of operations controls the order in which arithmetic operations are carried out. Symbolic constants are a method of using a name to represent a constant in a program.
5. **Discuss using VB .NET .Net functions to carry out commonly used operations.** In addition to the conversion functions and the Format function, other functions enable us to carry out complex operations easily in such areas as converting data types, working with dates and time, finance, and mathematics. The Pmt function is a useful function for computing the monthly payment necessary to repay a loan amount.
6. **Use buttons to clear text boxes and exit the project.** It is possible to clear text boxes by setting their Text property to an empty value ("") and the text box Focus method can be used to position the cursor in a text box. The project can be exited through the use of the End command.
7. **Describe the types of errors that commonly occur in a VB .NET .Net project and their causes.** The various types of errors that can occur in the development of a VB .NET program—syntax errors, run time errors, and logic errors—were also discussed in this chapter. The process of finding and removing errors is referred to as debugging.

EXERCISES

1. Write a single VB .NET statement to accomplish each of the following:
 - a. Declare the variable blnStatus as Boolean.
 - b. Assign “Nicolas” to the variable strFirstName.
 - c. Assign the value 0.85 to the txtAmount text box to appear in Currency format.

NEW VB .NET ELEMENTS

Controls/Objects	Properties	Methods	Events
text box control	Name Text	Focus	
form object		PrintForm	

NEW PROGRAMMING STATEMENTS

<p><i>Statement to declare a variable</i> Dim variable name As type</p> <p><i>Statement to assign a value to a constant</i> Const constant name As type = value</p> <p><i>Assignment Statement</i> Control property or variable = value, variable, expression, or control property</p> <p><i>Statement to use a function</i> Variable = functionname(arg1, arg2, ...)</p> <p><i>Statement to end execution of a project</i> End</p>

KEY TERMS

assignment statement	function	run time errors
bugs	internal documentation	string data type
comments	logic errors	symbolic constant
constant	memory cells	syntax errors
debugging	mnemonic variable names	text box
Edit Field	numeric data type	Text property
fixed point operations	operator	variables
floating point operations	prototyping	Object data type
format expression		

- d. Set the focus to the txtScore text box.
 - e. Increment the variable intCounter by 1.
2. What happens when each of the following is executed? Why?
 - a.

```
Private Sub btnCalculate_Click()
    Dim strAddress As String
    strAddress = "543 Elm Street"
    txtAnswer.Text = CDec(strAddress)
End Sub
```
 - b.

```
Private Sub btnCalculate_Click()
    Dim strAddress As String
    strAddress = "543 Elm Street"
    txtAnswer.Text = CInt(strAddress)
End Sub
```

3. What happens when the following code is executed? Why does this occur? How can it be corrected?

```
Private Sub btnCalculate_Click()
    Dim intNumber As Short
    intNumber = 42000
End Sub
```

4. Use the VB .NET Help facility to answer the following questions.
- What can cause a Type Mismatch error?
 - What is the data type of the value returned by the MsgBox function?
 - What does the Left function do? What are the arguments required?
 - What is the value and data type of the vbOKOnly constant?
 - What methods are available for the button?
5. When are two variables that have been assigned the same value not equal? Create an interface with one text box and one button. Write the following code for the button.

```
Private Sub btnCalculate_Click()
    Dim sngA As Single
    Dim dblB As Double
    sngA = 8.05006
    dblB = 8.05006
    txtAnswer.Text = sngA - dblB
End Sub
```

What is the result in the text box after executing the code? What would it be if the values stored in memory for each variable are equal? Why do you think the result comes out this way? What does this imply for your calculations?

PROJECTS

1. Review your design for Exercise 1 in Chapter 1 and then use the **Step-by-Step 3-2** instructions to copy the **Ex2-1** files into a new folder named **Ex3-1** in the Visual Studio Project folder. Follow those instructions to also rename the files in the folder to be **Ex3-1.vb**, **Ex3-1.vbproj**, and **Ex3-1.sln**. Be sure to make **Ex3-1.vb** your startup object. Add five text boxes and corresponding labels and modify the properties for the text boxes and labels as needed. The first text box should allow the user to input the student's name, and the next three text boxes are for input of the three quiz scores. The last text box is to display the average of the three quiz scores. Add a line control above this text box.

Use a button to sum the three test scores and compute and display the average of the three scores (use the Fixed Numeric format for the average.) Add buttons to print the form, to clear the text boxes and set the focus back to the student name text box, and to exit the project. Test your project with a student name of **Chris Patrick** and test scores of **71**, **79**, and **85**. Click the **Save All** icon to save the resulting project.

2. Review your design for Exercise 2 in Chapter 1 and then use the **Step-by-Step 3-2** instructions to copy the **Ex2-2** files into a new folder named **Ex3-2** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **Ex3-2.vb**, **Ex3-2.vbproj**, and **Ex3-2.sln** and to make **Ex3-2.vb** the startup object.

Now add four text boxes and corresponding labels to the Ex3-1.vb form and modify the properties for the text boxes and labels as needed. The first text box should allow the user to input the customer's name. The next two text boxes should allow the

user to input the square footage for a lawn and the cost per square foot for a given type of treatment. The fourth text box should display the cost of the treatment, which is equal to the square footage times the cost per square foot. Add a line control above this text box.

Use a button to compute and display the treatment cost. Format the cost per square foot and treatment cost as dollars and cents. Add buttons to print the form, to clear the text boxes and set the focus back to the square footage text box, and to exit the project. Test your project with a customer name of **Caroline Myers** with square footage of **3250** square feet and a treatment cost of **\$.002** per square foot. Click the **Save All** icon to save the resulting project.

3. Review your design for Exercise 3 in Chapter 1 and then use the **Step-by-Step 3-2** instructions to copy the **Ex2-3** files into a new folder named **Ex3-3** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **Ex3-3.vb**, **Ex3-3.vbproj**, and **Ex3-3.sln** and to make **Ex3-3.vb** the startup object.

Add four text boxes and corresponding labels and modify the properties for the text boxes and labels as needed. The first text box should allow the user to input the make of the car being tested. The second and third text boxes should allow the user to input the miles driven and the gallons of gas used. The fourth text box should display the miles per gallon, which is equal to the miles driven divided by the gallons used. Add a line control about this text box.

Use a button to compute and display the miles per gallon (use the Fixed Numeric format). Add buttons to print the form, to clear the text boxes and set the focus back to the automobile name text box, and to exit the project. Another button should provide for the user to exit the project. Test your project with a **Toyonda** make of car that was driven **225** miles on **7.3** gallons of gas. Click the **Save All** icon to save the resulting project.

4. Review your design for Exercise 4 in Chapter 1 and then use the **Step-by-Step 3-2** instructions to copy the **Ex2-4** files into a new folder named **Ex3-4** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **Ex3-4.vb**, **Ex3-4.vbproj**, and **Ex3-4.sln** and to make **Ex3-4.vb** the startup object.

Add five text boxes and corresponding labels and modify the properties for the text boxes and labels as needed. The first text box should allow the user to input the fixed cost of production, while the next two text boxes are for input of the unit cost and unit price values. The fourth text box should display the breakeven volume, which is equal to $\text{Fixed cost}/(\text{Unit price} - \text{Unit cost})$. The fifth text box should display the Breakeven revenue (Cost), which is equal to Breakeven volume times Unit price. Replace the message in the button with the code necessary to make these calculations and display the results. Format the Breakeven volume using the Standard Numeric format. Format all other text boxes to be dollars and cents. Add buttons to print the form, to clear the text boxes and set the focus back to the fixed cost text box, and to exit the project. Test your project with a Unit price of **\$25**, Unit cost of **\$15**, and Fixed cost of **\$1,000**. Click the **Save All** icon to save the resulting project.

5. Review your design for Exercise 5 in Chapter 1 and then use the **Step-by-Step 3-2** instructions to copy the **Ex2-5** files into a new folder named **Ex3-5** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **Ex3-5.vb**, **Ex3-5.vbproj**, and **Ex3-5.sln** and to make **Ex3-5.vb** the startup object.

Add seven text boxes and corresponding labels, and modify the properties for the text boxes and labels as needed. The first text box should allow the user to input the customer's name. The next four text boxes should allow the user to input the length of the room, the width of the room, the window area, and the door area. The sixth text box should display the room area and the seventh text box should display the number of rolls needed. Add a line control above this text box.

Replace the message in the button with the calculations to compute the room area and number of rolls of wallpaper needed. **Note:** Because the number of rolls calculated must be an integer, you should use Integer division to calculate the number of rolls and *then* add one (+1) to the result to account for the fractional remainder of a roll. Use the Fixed Numeric format to format all text boxes *except* the number of rolls. Add buttons to print the form, to clear the text boxes and set the focus back to the customer name text box, and to exit the project. Test your project for a **20' x 15'** room with **2** doors, each of which is **21** square feet, and **4** windows, each of which is **12** square feet. Assume that each roll of wallpaper will cover **45** square feet. Click the **Save All** icon to save the files for this project.

6. Use the **Step-by-Step 3-2** instructions to copy the **SimpleCalc** files that you created in the **Try It Yourself!** exercises into a new folder named **SimpleCalcNew** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **SimpleCalcNew.vb**, **SimpleCalcNew.vbproj**, and **SimpleCalcNew.sln** and to make **SimpleCalcNew.vb** the startup object.

Modify SimpleCalcNew form to replace the single button with four buttons: one for addition, one for subtraction, one for multiplication, and one for division (assume you are dividing the contents of the top text box by the contents of the second text box). Use the arithmetic operators (+, -, *, or /) as the Text property for each button. Also, add a button to exit the project. Try your calculator for various values in the two text boxes. Specifically, try to divide by zero and see what happens. Click the **Save All** icon to save the files in this project.

7. Use the **Step-by-Step 3-2** instructions to copy the **MonthPay** files into a new folder named **MonthPayNew** in the Visual Studio Projects folder. Follow those same instructions to rename the files in the folder to be **MonthPayNew.vb**, **MonthPayNew.vbproj**, and **MonthPayNew.sln** and to make **MonthPay.vb** the startup object. Modify the MonthPayNew.vb form to calculate the future value of a series of fixed value payments into an annuity for some number of months at a fixed interest rate. (Hint: The FV() function works *exactly* like the Pmt() function except that the fixed payment replaces the loan amount in the function and the future value is returned instead of the payment required.) You will need to modify the button code and the labels. Test your annuity calculator with a fixed payment of **\$100** per month for **10** years (120 months) at a **12** percent annual interest rate (1% monthly). Click the **Save All** icon to save the files in this project.

8. The library at Yeehaw Technical Institute needs an application that will compute the overdue fines for books as they are returned. As a first version of this application, assume that the borrower's name, the days overdue, and the fine per day (which differs depending on whether the borrower is a faculty member, a grad student, or an undergraduate) are entered in text boxes and that the Total fine due (Days overdue x Daily fine) is output by clicking a button. Create an interface using text boxes, labels, and buttons. Develop an IPO Table and pseudocode for this problem and then create a project

that will carry out the logic using VB .NET. Give the project a name of **Ex3-8** and change the name of the form to be **Ex3-8.vb**. The form should have Calculate, Clear, Print, and Exit buttons. Try out your project with the following data:

Borrower's name: **Jody Silver**

Days overdue: **10**

Daily fine: **\$.25**

Click the **Save All** icon to save the files in the project.

9. Bob's Bike Factory produces custom-made bikes in four basic models. The price of the basic model plus the price of accessories determine the total price. Some distributors get special discounts, which are subtracted from the price before sales taxes are added. Develop an IPO Table and pseudocode for this project, a sketch of the form, and a VB .NET project that has a form with a title and a logo for Bob's Bike Factory (e.g., a bike icon—use graphics/icons/industry/bicycle.ico) plus Calculate, Clear, Print, and Exit buttons. Give the project a name of **Ex3-9** and rename the form as **Ex3-9.vb**. The final project should have the following features:

1. The price of the model, the discount rate, and the number of bikes are input via text boxes. The tax rate is set in the code.
2. The discount (if any), taxes due, and total amount due are computed by a Calculate button and are output to text boxes with appropriate labels.

Try out your project with the following data:

Bikes ordered: **10 Model Red Racers**

Price: **\$1499.99** each

Tax rate: **7%**

Discount: **15%**

Click the **Save All** icon to save the files in the project.

10. Reckoning that they might amuse themselves on their PDAs while waiting in the snow and rain for dangerous weather phenomena, the meteorologists at The Weather Channel need a simple program to allow them to convert temperatures from Fahrenheit to Celsius. The interface would be composed of two textboxes and a button. They would simply type the Fahrenheit temperature (F) into the first textbox, click the button and the corresponding Celsius temperature (C) would appear in the second textbox. The conversion relationship between the two is: $C = 5/9*(F-32)$. Design and create a VB .NET program for this application. Give your project a name of **Ex3-10** and rename the form as **Ex3-10.vb**. Try out your project with a Fahrenheit temperature of 32 degrees which should result in a Celsius temperature of 0 degrees. Click the **Save All** icon to save the files in the project.

11. A currency trader would like a simple program to calculate the equivalent value of various foreign currencies for a given amount in US\$. Assume that a VB program would utilize two textboxes to input the amount in US\$ and the exchange rate. Also, an appropriately labeled textbox is used to display the equivalent amount of the foreign currency. The amount of foreign currency is calculated by simply multiplying the amount in US\$ by the exchange rate. Assume that the calculation is performed when a button is clicked. Design and create a VB .NET program for this application. Give the project a name of **Ex3-11** and rename the form as **Ex3-11.vb**. Try your project to convert US dollars to UK pounds with an exchange rate of 0.69. Check your newspaper for examples of other exchange rates to use in testing your project. Click the **Save All** icon

to save the files in your project.

12. Financial ratios are often used as a measure of a company's performance. They are important as indicators of the company's health that may be used in management and investment decisions. Several commonly used financial ratios are:

- a. The Current Ratio is used to predict the capability of a company to pay its current liabilities. The Current Ratio is calculated by dividing Current Assets divided by Current Liabilities.
- b. The Quick Ratio provides more liquid test for paying Current Liabilities. The quick ratio is equal to $(\text{Current Assets} - \text{Inventory}) / \text{Current Liabilities}$.
- c. The Receivables Turnover Ratio represents the average time for a firm to convert receivables to cash. This ratio is equal to Net Sales divided by Accounts Receivable.
- d. The Average Collection Period indicates the average number of days to turn over accounts receivable. The Average Collection Period is equal to 365 divided by the Receivables Turnover Ratio.
- e. The extent that a firm depends on loans versus the resources from stakeholders and other owners is measured by the Debt to Equity Ratio. Debt to Equity is equal to the Total Liabilities divided by Owner's Equity.
- f. Return on Investment (ROI) represents the capability of a company to earn profits for its owners. $\text{ROI} = \text{Net Income} / \text{Owner's Equity}$.

Design and create a VB .NET program for calculating these ratios. Give your project a name of **Ex3-12** and rename the form to be **Ex3-12.vb**. Click the **Save All** icon to save the project files.

13. On what day and date will Easter fall this year? How about the next? What about in the year 2053? Since the Easter holiday always falls on a Sunday you could narrow it down, but how could you pinpoint it exactly. The actual date of Easter each year is determined by the Catholic church. In A.D. 325, church leaders decided that Easter should fall on the first Sunday after the first full moon occurring on, or after the spring equinox (the day in March on which day and night have equal length and which marks the beginning of Spring). This sounds difficult enough, but any calculations must also take into account the fact that our calendar is not perfect and must be corrected every so often. Fortunately, an algorithm has been developed that will allow us to calculate the exact date of Easter for any given year.

- a. Choose a year and call it X.
- b. Divide X by 19 to get a quotient (which is ignored) and a remainder A.
- c. Divide X by 100 to get a quotient B and a remainder C.
- d. Divide B by 4 to get a quotient D and a remainder E.
- e. Divide $8B+13$ by 25 to get a quotient G and a remainder (which we ignore).
- f. Divide $19A+B-D-G+15$ by 30 to get a quotient which we ignore and a remainder H.
- g. Divide $A+11H$ by 319 to get a quotient M and a remainder (which is ignored).
- h. Divide C by 4 to get a quotient J and a remainder K.
- i. Divide $2E+2J-K-H+M+32$ by 7 to get a quotient (which is ignored) and a remainder L.

- j. Divide $H-M+L+90$ by 25 to get quotient N and a remainder (which is ignored).
- k. Divide $H-M+L+N+19$ by 32 to get a quotient (which we ignore) and a remainder P .
- l. Easter Sunday will be the P th day of the N th month.
For example, in 2003, Easter fell on April

Design and create a VB .NET program for calculating the date of Easter for any year entered by the user. Give the project a name of **Ex3-13** and rename the form as **Ex3-13.vb**.

This material is protected by copyright and may not be downloaded, reproduced, stored in a retrieval system, modified, made available on a network, used to create derivative works, or transmitted in any form or by any means, except (i) in the United States, as permitted under Section 107 or 108 of the 1976 United States Copyright Act, or internationally, as permitted by other applicable national copyright laws, or (ii) as expressly authorized on this Web site, or (iii) with the prior written permission of Wiley

