

TOPICS

- | | |
|----------------------------------|---|
| 3.1 Gathering Text Input | 3.7 Group Boxes and the Load Event Procedure |
| 3.2 Variables and Data Types | 3.8 Focus on Program Design and Problem Solving: Building the <i>Room Charge Calculator</i> Application |
| 3.3 Performing Calculations | 3.9 More about Debugging: Locating Logic Errors |
| 3.4 Mixing Different Data Types | |
| 3.5 Formatting Numbers and Dates | |
| 3.6 Exception Handling | |

This chapter covers the use of text boxes to gather input from users. It also discusses the use of variables, named constants, type conversion functions, and mathematical calculations. You will be introduced to the `GroupBox` control as a way to organize controls on an application's form. The *Format* menu commands, which allow you to align, size, and center controls are also discussed. You will learn about the form's `Load` procedure, which automatically executes when a form is loaded into memory, and debugging techniques for locating logic errors.

3.1 Gathering Text Input

CONCEPT: In this section, we use the `TextBox` control to gather input the user has typed on the keyboard. We also alter a form's tab order and assign keyboard access keys to controls.

The programs you have written and examined so far perform operations without requiring information from the user. In reality, most programs ask the user to enter values. For example, a program that calculates payroll for a small business might ask the user to enter the name of the employee, the hours worked, and the hourly pay rate. The program then uses this information to print the employee's paycheck.

A **text box** is a rectangular area on a form that accepts keyboard input. As the user types, the characters are stored in the text box. In Visual Basic, you create a text box with a **TextBox** control. Tutorial 3-1 examines an application that uses a TextBox control.



Tutorial 3-1: Using a TextBox control

- Step 1:** Open the *Greetings* project from the student sample programs folder named *Chap3\Greetings*.
- Step 2:** Click the *Start* button (▶) to run the application. The application's form appears, as shown in Figure 3-1. The TextBox control is the white rectangular area beneath the label that reads *Enter Your Name*.
Notice that the TextBox control shows a blinking text cursor, indicating it is ready to receive keyboard input.
- Step 3:** Type your name. As you enter characters on the keyboard, they appear in the TextBox control.
- Step 4:** Click the *Show Greeting* button. The message *Hello* followed by the name you entered, appears in a label below the TextBox control. The form now appears similar to the one shown in Figure 3-2.
- Step 5:** Click inside the TextBox control and use the **Delete** and/or **Backspace** key to erase the name you entered. Enter another name, and then click the *Show Greeting* button. Notice that the greeting message changes accordingly.
- Step 6:** Click the *Exit* button to exit the application. You have returned to Design mode.
- Step 7:** Look at the application's form in the *Design* window. Figure 3-3 shows the form with its controls.
Notice that the name of the TextBox control starts with `txt`, which is the standard prefix for TextBox controls. Like the Label control, the TextBox control has a `Text` property. However, the Label control's `Text` property is only for displaying information—the user cannot directly alter its contents. The TextBox

Figure 3-1 *Greetings* project initial form

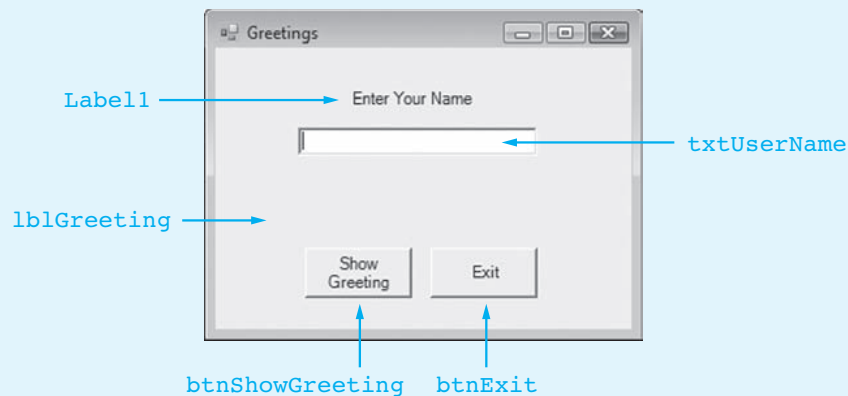


Figure 3-2 *Greetings* project completed form



control's `Text` property is for input purposes. The user can alter it by typing characters into the `TextBox` control. Whatever the user types into the `TextBox` control is stored, as a string, in its `Text` property.

Figure 3-3 *Greetings* project form with controls labeled



Using the Text Property in Code

You access a `TextBox` control's `Text` property in code the same way you access other properties. For example, assume an application has a `Label` control named `lblInfo` and a `TextBox` control named `txtInput`. The following assignment statement copies the contents of the `TextBox` control's `Text` property into the `Label` control's `Text` property.

```
lblInfo.Text = txtInput.Text
```

Clear a Text Box

Recall from the discussion on object-oriented programming in Chapter 1 that an object contains methods, which are actions the object performs. To execute an object's method, write a statement that calls the method. The general format of such a statement is

Object.Method

Object is the name of the object and *Method* is the name of the method that is being called.

A `TextBox` control is an object, and has a variety of methods that perform operations on the text box or its contents. One of these methods is `Clear`, which clears the contents of the text box's `Text` property. The general format of the `Clear` method is

```
TextBoxName.Clear()
```

TextBoxName is the name of the `TextBox` control. Here is an example:

```
txtInput.Clear()
```

When this statement executes, the `Text` property of `txtInput` is cleared and the text box appears empty on the screen.

You can also clear a text box by assigning the predefined constant `String.Empty` to its `Text` property. Here is an example:

```
txtInput.Text = String.Empty
```

Once this statement executes, the `Text` property of `txtInput` is cleared and the text box appears empty on the screen.

String Concatenation

Returning to the *Greetings* application, let's look at the code for the `btnShowGreeting` control's `Click` event:

```
Private Sub btnShowGreeting_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowGreeting.Click

    ' Display a customized greeting to the user
    ' in the lblGreeting control
    lblGreeting.Text = "Hello " & txtUserName.Text
End Sub
```

The assignment statement in this procedure introduces a new operator: the ampersand (&). When the ampersand is used in this way, it performs a **string concatenation**. This means that one string is appended to another.

The & operator creates a string that is a combination of the string on its left and the string on its right. Specifically, it appends the string on its right to the string on its left. For example, assume an application uses a Label control named `lblMessage`. The following statement copies the string “Good morning Charlie” into the control's `Text` property:

```
lblMessage.Text = "Good morning " & "Charlie"
```

In our *Greetings* application, if the user types *Becky* into the `txtUserName` control, the control's `Text` property is set to *Becky*. So the statement

```
lblGreeting.Text = "Hello " & txtUserName.Text
```

assigns the string “Hello Becky” to `lblGreeting`'s `Text` property.

Look again at the assignment statement. Notice there is a space in the string literal after the word *Hello*. This prevents the two strings being concatenated from running together.

In a few moments, it will be your turn to create an application using `TextBox` controls and string concatenation. Tutorial 3-2 leads you through the process.

Aligning Controls in Design Mode

Visual Studio provides a convenient way to align controls on forms. When you drag a control to a position on a form that aligns with another control, guide lines automatically appear. In Figure 3-4, for example, a `TextBox` has been placed below an existing `TextBox`. The blue guide lines tell us the two controls are aligned vertically. If either control is dragged sideways, the guide lines disappear.

In Figure 3-5, two `TextBox` controls are aligned horizontally, indicated by lavender guide lines. In Figure 3-6, guide lines appear for simultaneous vertical and horizontal alignment. All in all, this feature of Visual Studio takes the guesswork out of aligning controls, saving you a lot of time. Tutorial 3-2 shows you how to build the *Date String* application.

Figure 3-4 Vertical guide lines align two TextBox controls

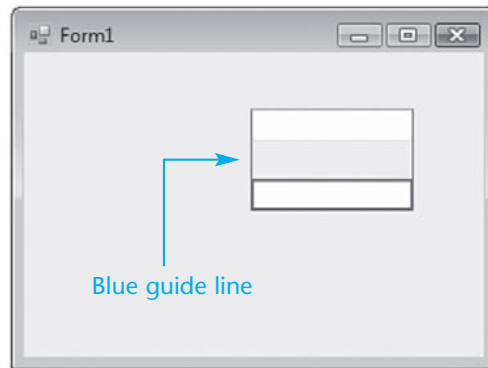


Figure 3-5 Horizontal guide lines align two TextBox controls

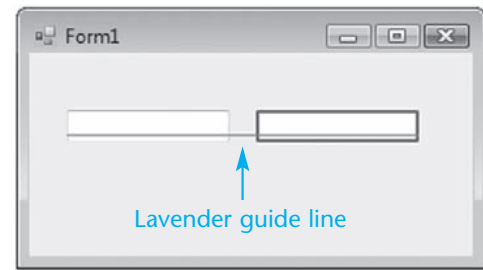
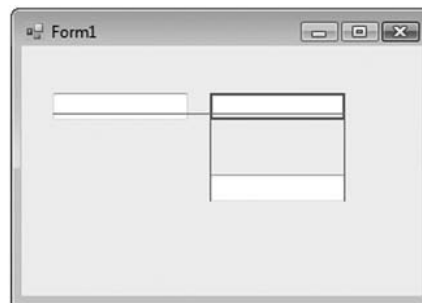


Figure 3-6 Horizontal and vertical guide lines can be used together



Tutorial 3-2: Building the *Date String* application

You will create an application that lets the user enter the following information about today's date:

- The day of the week
- The name of the month
- The numeric day of the month
- The year

When the user enters the information and clicks a button, the application displays a date string such as Friday, December 5, 2008.

Step 1: Start Visual Studio and start a new Windows application named *Date String*.

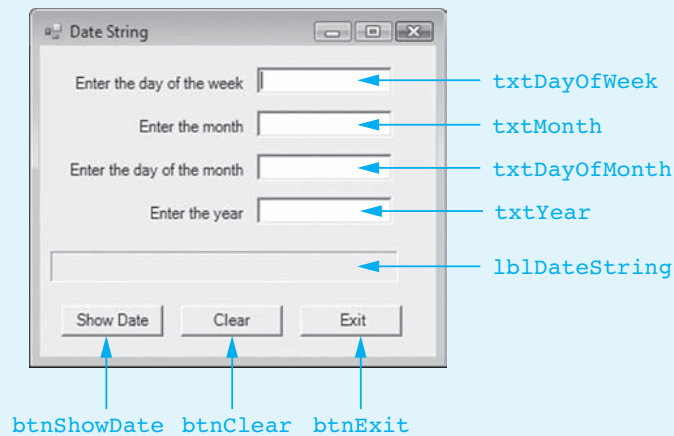
Step 2: Create the form shown in Figure 3-7, using the following instructions:

- You insert TextBox controls by double-clicking the TextBox tool in the ToolBox. When a TextBox control is created, it will be given a default name. As with other controls, you can change a TextBox control's name by modifying its Name property.

- Give each control the name indicated in the figure. The labels that display *Enter the day of the week;* *Enter the month;* *Enter the day of the month;* and *Enter the year;* will not be referred to in code, so they may keep their default names.
- Set the `lblDateString` label's `AutoSize` property to `False`, its `BorderStyle` property to `Fixed3D`, and its `TextAlign` property to `MiddleCenter`. Resize the label as shown in Figure 3-7, and delete the contents of the label's `Text` property.
- Set the form's `Text` property to *Date String*.

As you place `TextBox` control on your form, delete the contents of their `Text` properties. This will cause them to appear empty when the application runs.

Figure 3-7 *Date String form*



Step 3: Next, you will write code for the `btnShowDate` button's `Click` event procedure. Double-click the button to create the code template, and then enter the lines shown in bold:

```
Private Sub btnShowDate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowDate.Click

    ' Concatenate the input and build the date string.
    lblDateString.Text = txtDayOfWeek.Text & ", " _
    & txtMonth.Text & " " _
    & txtDayOfMonth.Text & ", " _
    & txtYear.Text
End Sub
```

This example introduces a new programming technique: breaking up long lines of code with the line-continuation character. The **line-continuation character** is actually two characters: a space, followed by an underscore or underline character.

Quite often, you will find yourself writing statements that are too long to fit entirely inside the *Code* window. Your code will be hard to read if you have to scroll the *Code* window to the right to view long statements. The line-continuation character allows you to break a long statement into several lines.

Here are some rules to remember about the line-continuation character:

- A space must immediately precede the underscore character.
- You cannot break up a word, quoted string, or name using the line-continuation character.
- You cannot separate an object name from its property or method name.
- You cannot put a comment at the end of a line after the line-continuation character. The line-continuation character must be the last thing you type on a line.



TIP: Another way to deal with long code lines is to reduce the font size used in the *Code* window by clicking *Tools* on the menu bar, and then clicking the *Options . . .* command. On the *Options* dialog box, click the arrow next to *Environment* in the left pane, and then click *Fonts and Colors*. Then you may select the desired font and size.

Step 4: The `btnClear` button allows the user to start over with a form that is empty of previous values. The `btnClear_Click` event procedure clears the contents of all the `TextBox` controls and the `lblDateString` label. To accomplish this, the procedure calls each `TextBox` control's `Clear` method, and assigns the special value `String.Empty` to `lblDateString`'s `Text` property. (The value `String.Empty` represents an empty string. Assigning `String.Empty` to a label's `Text` property clears the value displayed by the label.) Enter the following bold code into the `btnClear_Click` event procedure:

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click

    ' Clear the Text Boxes and lblDateString
    txtDayOfWeek.Clear()
    txtMonth.Clear()
    txtDayOfMonth.Clear()
    txtYear.Clear()
    lblDateString.Text = String.Empty
End Sub
```

Step 5: Enter the following code (shown in bold), which terminates the application, into the `btnExit_Click` event procedure:

```
Private Sub btnExit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click

    ' End the application by closing the form.
    Me.Close()
End Sub
```

Step 6: Save the project.


Step 7: Click the *Start* button () to run the application. With the application running, enter the requested information into the `TextBox` controls and click the *Show Date* button. Your form should appear similar to the one shown in Figure 3-8.

Figure 3-8 Running the *Date String* application

Step 8: Click the *Clear* button to test it, and then enter new values into the TextBox controls. Click the *Show Date* button.

Step 9: Click the *Exit* button to exit the application.

The Focus Method

When an application is running and a form is displayed, one of the form's controls always has the **focus**. The control having the focus is the one that receives the user's keyboard input or mouse clicks. For example, when a TextBox control has the focus, it receives the characters that the user enters on the keyboard. When a button has the focus, pressing the **[Enter]** key executes the button's Click event procedure.

You can tell which control has the focus by looking at the form at runtime. When a TextBox control has the focus, a blinking text cursor appears inside it, or the text inside the TextBox control appears highlighted. When a button, radio button, or check box has the focus, a thin dotted line appears around the control.



NOTE: Only controls capable of receiving some sort of input, such as text boxes and buttons, may have the focus.

Often, you want to make sure a particular control has the focus. Consider the *Date String* application, for example. When the *Clear* button is clicked, the focus should return to the `txtDayOfWeek` TextBox control. This would make it unnecessary for the user to click the TextBox control in order to start entering another set of information.

In code, you move the focus to a control by calling the **Focus method**. The method's general syntax is:

```
ControlName.Focus()
```

where *ControlName* is the name of the control. For instance, you move the focus to the `txtDayOfWeek` TextBox control with the statement `txtDayOfWeek.Focus()`. After the statement executes, the `txtDayOfWeek` control will have the focus. In Tutorial 3-3, you add this statement to the *Clear* button's Click event procedure so `txtDayOfWeek` has the focus after the TextBox controls and the `lblDateString` label are cleared.



Tutorial 3-3: Using the `Focus` method

Step 1: Open the *Date String* project that you created in Tutorial 3-2.

Step 2: Open the *Code* window and add the statements shown in bold to the `btnClear_Click` event procedure.

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click

    ' Clear the Text Boxes and lblDateString
    txtDayOfWeek.Clear()
    txtMonth.Clear()
    txtDayOfMonth.Clear()
    txtYear.Clear()
    lblDateString.Text = ""
    ' Return the focus to txtDayOfWeek
    txtDayOfWeek.Focus()
End Sub
```

Step 3: Run the application. Enter some information into the TextBox controls, and then click the *Clear* button. The focus should return to the `txtDayOfWeek` TextBox control.

Step 4: Save the project.

Controlling a Form's Tab Order with the `TabIndex` Property

In Windows applications, pressing the `[Tab]` key changes the focus from one control to another. The order in which controls receive the focus is called the **tab order**. When you place controls on a form in Visual Basic, the tab order will be the same sequence in which you created the controls. In many cases this is the tab order you want, but sometimes you rearrange controls on a form, delete controls, and add new ones. These modifications often lead to a disorganized tab order, which can confuse and irritate the users of your application. Users want to tab smoothly from one control to the next, in a logical sequence.

You can modify the tab order by changing a control's `TabIndex` property. The **`TabIndex` property** contains a numeric value, which indicates the control's position in the tab order. When you create a control, Visual Basic automatically assigns a value to its `TabIndex` property. The first control you create on a form will have a `TabIndex` of 0, the second will have a `TabIndex` of 1, and so on. The control with a `TabIndex` of 0 will be the first control in the tab order. The next control in the tab order will be the one with a `TabIndex` of 1. The tab order continues in this sequence.

You may change the tab order of a form's controls by selecting them, one-by-one, and changing their `TabIndex` property in the *Properties* window. An easier method, however, is to click *View* on the menu bar, and then click *Tab Order*. This causes the form to be displayed in **tab order selection mode**. In this mode, each control's existing `TabIndex` value is displayed on the form. Then you establish a new tab order by clicking the controls in the order you want. When you are finished, exit tab order selection mode by pressing the `[Esc]` key. Tutorial 3-4 shows you how to change the tab order.

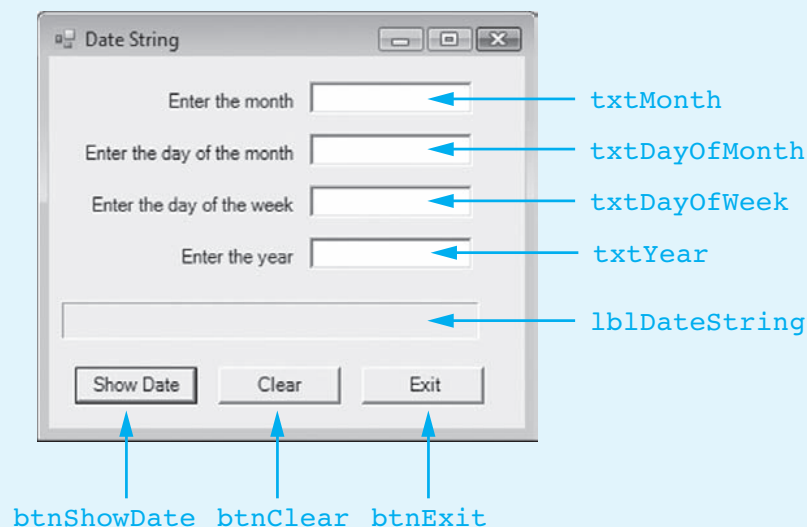


Tutorial 3-4: Changing the tab order

In this tutorial, you rearrange the controls in the *Date String* application, and then change the tab order to accommodate the controls' new positions.

- Step 1:** Open the *Date String* project that you created in Tutorials 3-2 and 3-4.
- Step 2:** Open the application's form in the *Design* window. Rearrange the controls to match Figure 3-9. (You might want to enlarge the form temporarily so you have room to move some of the controls around. Don't forget to move the labels that correspond to the TextBox controls.)

Figure 3-9 *Date String* form

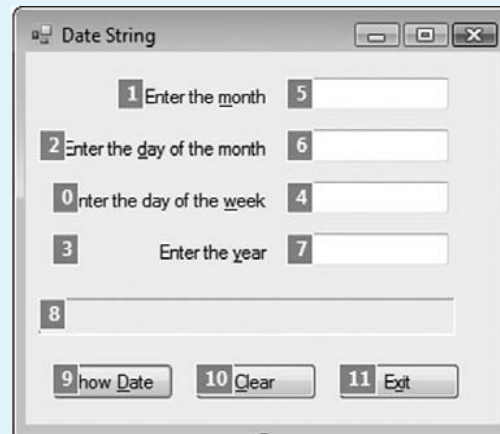


- Step 3:** Run the application and notice which control has the focus. Press the **[Tab]** key several times and observe the tab order.
- Step 4:** Stop the application and return to Design mode.
- Step 5:** Click *View* on the menu bar, and then click *Tab Order*. The form should switch to tab order selection mode, as shown in Figure 3-10. The numbers displayed in the upper left corner of each control are the existing `TabIndex` values.



NOTE: Your existing `TabIndex` values may be different from those shown in Figure 3-10.

- Step 6:** Click the following controls in the order they are listed here: `txtMonth`, `txtDayOfMonth`, `txtDayOfWeek`, `txtYear`, `btnShowDate`, `btnClear`, `btnExit`.

Figure 3-10 Form in tab order selection mode

Step 7: The controls you clicked should now have the following TabIndex values displayed:

```
txtMonth:      0
txtDayOfMonth: 1
txtDayOfWeek:  2
txtYear:       3
btnShowDate:   4
btnClear:      5
btnExit:       6
```



NOTE: The Label controls cannot receive the focus, so do not be concerned with the TabIndex values displayed for them.

Step 8: Press the **[Esc]** key to exit tab order selection mode.

Step 9: Don't forget to change the `btnClear_Click` event procedure so `txtMonth` gets the focus when the form is cleared. The code for the procedure is as follows, with the modified lines in bold:

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click

    ' Clear the Text Boxes and lblDateString
    txtDayOfWeek.Clear()
    txtMonth.Clear()
    txtDayOfMonth.Clear()
    txtYear.Clear()
    lblDateString.Text = ""
    ' Return the focus to txtMonth
    txtMonth.Focus()
End Sub
```

Step 10: Run the application and test the new tab order.

Step 11: End the application and save the project.

Here are a few last notes about the `TabIndex` property:

- If you do not want a control to receive the focus when the user presses the `[Tab]` key, set its `TabStop` property to *False*.
- An error will occur if you assign a negative value to the `TabIndex` property in code.
- A control whose `Visible` property is set to *False* or whose `Enabled` property is set to *False* cannot receive the focus.
- `GroupBox` and `Label` controls have a `TabIndex` property, but they are skipped in the tab order.

Assigning Keyboard Access Keys to Buttons

An **access key**, also known as a **mnemonic**, is a key pressed in combination with the `[Alt]` key to access a control such as a button quickly. When you assign an access key to a button, the user can trigger a `Click` event either by clicking the button with the mouse or by using the access key. Users who are quick with the keyboard prefer to use access keys instead of the mouse.

You assign an access key to a button through its `Text` property. For example, assume an application has a button whose `Text` property is set to *Exit*. You wish to assign the access key `[Alt]+X` to the button, so the user may trigger the button's `Click` event by pressing `[Alt]+X` on the keyboard. To make the assignment, place an ampersand (&) before the letter *x* in the button's `Text` property: *E&xit*. Figure 3-11 shows how the `Text` property appears in the *Property* window.

Although the ampersand is part of the button's `Text` property, it is not displayed on the button. With the ampersand in front of the letter *x*, the letter will appear underlined as shown in Figure 3-12. This indicates that the button may be clicked by pressing `[Alt]+X` on the keyboard.

Figure 3-11 Text property *E&xit*

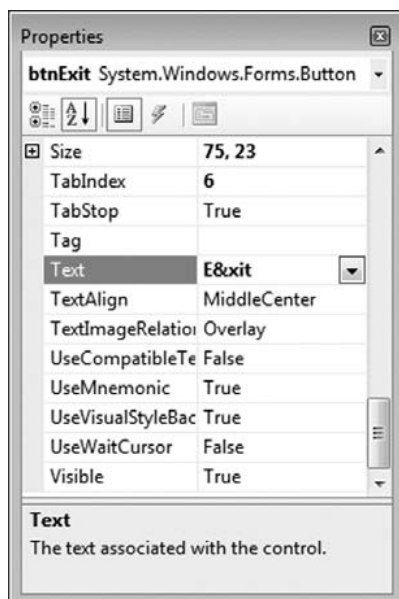
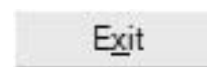


Figure 3-12 Button with *E&xit* text



NOTE: Access keys do not distinguish between uppercase and lowercase characters. There is no difference between `[Alt]+X` and `[Alt]+x`.

Suppose we had stored the value `&Exit` in the button's Text property. The ampersand is in front of the letter *E*, so `[Alt]+[E]` becomes the access key. The button will appear as shown in Figure 3-13.

Assigning the Same Access Key to Multiple Buttons

Be careful not to assign the same access key to two or more buttons on the same form. If two or more buttons share the same access key, a `Click` event is triggered for the first button created when the user presses the access key.

Displaying the & Character on a Button

If you want to display an ampersand character on a button use two ampersands (`&&`) in the Text property. Using two ampersands causes a single ampersand to display and does not define an access key. For example, if a button has the Text property `Beans && Cream` the button will appear as shown in Figure 3-14.

Figure 3-13 Button with `&Exit` text



Figure 3-14 Button with text `Beans && Cream`



Accept Buttons and Cancel Buttons

An **accept button** is a button on a form that is clicked when the user presses the `[Enter]` key. A **cancel button** is a button on a form that is clicked when the user presses the `[Esc]` key. Forms have two properties, `AcceptButton` and `CancelButton`, which allow you to designate an accept button and a cancel button. When you select these properties in the *Properties* window, a down-arrow button (`[v]`) appears, which allows you to display a drop-down list. The list contains the names of all the buttons on the form. You select the button that you want to designate as the accept button or cancel button.

Any button that is frequently clicked should probably be selected as the accept button. This will allow keyboard users to access the button quickly and easily. *Exit* or *Cancel* buttons are likely candidates to become cancel buttons. In Tutorial 3-5, you set access keys, accept, and cancel buttons.





Tutorial 3-5:

Setting access keys, accept, and cancel buttons

In this tutorial, you assign access keys to the buttons in the *Date String* application, and set accept and cancel buttons.

- Step 1:** Open the *Date String* project that you have worked on in Tutorials 3-2 through 3-4.
- Step 2:** Open the application's form in the *Design* window.
- Step 3:** Select the *Show Date* button (`btnShowDate`) and change its text to read *Show & Date*. This assigns `[Alt]+[D]` as the button's access key.

- Step 4:** Select the *Clear* button (`btnClear`) and change its text to read *Clea&r*. This assigns `[Alt]+[R]` as the button's access key.
- Step 5:** Select the *Exit* button (`btnExit`) and change its text to read *E&xit*. This assigns `[Alt]+[X]` as the button's access key.
- Step 6:** Select the form, and then select the `AcceptButton` property in the *Properties* window. Click the down-arrow button () to display the drop-down list of buttons. Select `btnShowDate` from the list.
- Step 7:** With the form still selected, select the `CancelButton` property in the *Properties* window. Click the down-arrow button () to display the drop-down list of buttons. Select `btnExit` from the list.
- Step 8:** Run the application and test the buttons' new settings. Notice that when you press the `[Enter]` key, the *Show Date String* button's `Click` procedure is executed; when you press the `[Esc]` key, the application exits.



NOTE: When the application executes, the access keys you assigned to the buttons are not displayed as underlined characters until you press the `[Alt]` key.

- Step 9:** Save the project.



Checkpoint

- 3.1 What `TextBox` control property holds text entered by the user?
- 3.2 Assume an application has a label named `lblMessage` and a `TextBox` control named `txtInput`. Write the statement that takes text the user entered into the `TextBox` control and assigns it to the label's `Text` property.
- 3.3 If the following statement is executed, what will the `lblGreeting` control display?
`lblGreeting.Text = "Hello " & "Jonathon, " & "how are you?"`
- 3.4 What is the line-continuation character, and what does it do?
- 3.5 What is meant when it is said that a control has the focus?
- 3.6 Write a statement that gives the focus to the `txtLastName` control.
- 3.7 What is meant by tab order?
- 3.8 How does the `TabIndex` property affect the tab order?
- 3.9 How does Visual Basic normally assign the tab order?
- 3.10 What happens when a control's `TabStop` property is set to *False*?
- 3.11 Assume a button's `Text` property is set to the text *Show &Map*. What effect does the `&` character have?
- 3.12 What is an accept button? What is a cancel button? How do you establish these buttons on a form?

3.2 Variables and Data Types

CONCEPT: Variables hold information that may be manipulated, used to manipulate other information, or remembered for later use.



VideoNote

Introduction
to Variables

A **variable** is a storage location in computer memory that holds data while a program is running. It is called a variable because the data it holds can be changed by statements in the program.

In this chapter, you have seen programs that store data in properties belonging to Visual Basic controls. Although control properties are useful, you must store data in variables when performing calculations. Generally speaking, you can do a number of things with variables:

- Copy and store values entered by the user so the values can be manipulated
- Perform arithmetic on numeric values
- Test values to determine that they meet some criterion
- Temporarily hold and manipulate the value of a control property
- Remember information for later use in a program

Think of a variable as a name that represents a location in the computer's random-access memory (RAM). When a value is stored in a variable, it is actually stored in RAM. You use the assignment operator (=) to store a value in a variable, just as you do with a control property. For example, suppose a program uses a variable named `intLength`. The following statement stores the value 112 in that variable:

```
intLength = 112
```

When this statement executes, the value 112 is stored in the memory location the name `intLength` represents. As another example, assume the following statement appears in a program that uses a variable named `strGreeting` and a TextBox control named `txtName`:

```
strGreeting = "Good morning " & txtName.Text
```

Suppose the user has already entered Holly into the `txtName` TextBox control. When the statement executes, the variable `strGreeting` is assigned the string "Good morning Holly".

Declaring Variables

A **variable declaration** is a statement that creates a variable in memory when a program executes. The declaration indicates the name you wish to give the variable and the type of information the variable will hold. Here is the general form of a variable declaration:

```
Dim VariableName As DataType
```

Here is an example of a variable declaration:

```
Dim intLength As Integer
```

Let's look at each part of this statement, and its purpose:

- The `Dim` keyword tells Visual Basic that a variable is being declared.
- `intLength` is the name of the variable.
- `As Integer` indicates the variable's data type. We know it will be used to hold integer numbers.

You can declare multiple variables with one `Dim` statement, as shown in the following statement. It declares three variables, all holding integers:

```
Dim intLength, intWidth, intHeight As Integer
```

Assigning Values to Variables

A value is put into a variable with an assignment statement. For example, the following statement assigns the value 20 to the variable `intUnitsSold`:

```
intUnitsSold = 20
```

The `=` operator is called the assignment operator. A variable name must always appear on the left side of the assignment operator. For example, the following would be incorrect:

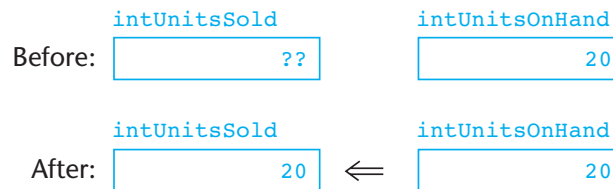
```
20 = intUnitsSold
```

On the right side of the operator, you can put a literal, another variable, or a mathematical expression that matches the variable's type. In the following, the contents of the variable on the right side of the `=` sign is copied into the variable on the left side:

```
intUnitsSold = intUnitsOnHand
```

Suppose `intUnitsOnHand` already equals 20. Then Figure 3-15 shows how the value 20 is copied into the memory location represented by `intUnitsSold`.

Figure 3-15 Assigning `intUnitsOnHand` to `intUnitsSold`



The assignment operator only changes the left operand. The right operand (or expression) does not change value. Sometimes your program will contain a series of statements that pass a value from one variable to the next. When the following statements execute, all three variables will contain the same value, 50:

```
Dim intA, intB, intC As Integer
intA = 50
intB = intA
intC = intB
```

A variable can hold only one value at a time. If you assign a new value to the variable, the new value replaces the variable's previous contents. There is no way to “undo” this operation. For example:

```
Dim intA, intB As Integer
intA = 50
intA = 99
```

After the second assignment statement, `intA` equals 99. The value 50 no longer exists in memory.

Integer Data Types

Integer data types hold integer values such as -5, 26, 12345, and 0. The following code example shows examples of the different integer types available in Visual Basic:


```

Dim bytInches As Byte
Dim shrtFeet as Short
Dim intMiles As Integer
Dim lngNationalDebt As Long

bytInches = 26
shrtFeet = 32767
intMiles = 2100432877
lngNationalDebt = 40000000000001

```

We usually use a three- or four-letter prefix when naming each variable. The prefix is not required, but it helps you to remember a variable's type. Table 3-1 lists the Visual Basic integer data types, showing their naming prefixes and descriptions. Unsigned integers can only hold positive values (zero is considered positive). Signed integers can hold both positive and negative values.

Table 3-1 Integer data types

Type	Naming Prefix	Description
Byte	byt	Holds an unsigned integer value in the range 0 to 255
Short	shrt	Holds a signed integer in the range -32,768 to +32,767
Integer	int	Holds a signed integer in the range -2,147,483,648 to 2,147,483,647
Long	lng	Holds a signed integer in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Each type has a different storage size and range of possible values it can hold. Most of the time, you will use the Integer data type for integer-type values. Its name is easy to remember, and Integer values are efficiently processed by the computer.

Integer Literals

When you write an integer literal in your program code, Visual Basic assumes the literal is type Integer if the value fits within the allowed range for the Integer data type. A value larger than that will be assumed to be type Long. On rare occasions, if you may want to override the literal's default type, append a special character to the end of the number:

```

I Integer literal
L Long integer literal
S Short integer literal

```

In the following code example, an integer literal uses the L character to identify it as type Long:

```

Dim lngCounter As Long
lngCounter = 10000L

```

In the following, an integer literal uses the S character to identify it as type Short:

```

Dim shrtFeet as Short
shrtFeet = 1234S

```



TIP: You cannot embed commas in numeric literals. The following, for example, causes an error: `intMiles = 32,767`

Floating-Point Data Types

Values that have fractional parts and use a decimal point must be stored in one of Visual Basic's floating-point data types. Table 3-2 lists the floating-point data types, showing their naming prefixes and descriptions.

Table 3-2 Floating-point data types

Type	Naming Prefix	Description
Single	sng	Holds a signed single precision real number with 7 significant digits, in the range of approximately plus or minus 1.0×10^{38}
Double	dbl	Holds a signed double precision real number with 15 significant digits, in the range of approximately plus or minus 1.0×10^{308}
Decimal	dec	Decimal real number, 29 significant digits. Its range (with no decimal places) is $\pm 79,228,162,514,264,337,593,543,950,335$

Significant Digits

The significant digits measurement for each floating-point data type is important for certain kinds of calculations. Suppose you were simulating a chemical reaction and needed to calculate the number of calories produced. You might produce a number such as 1.234567824724. If you used a variable of type Single, only the first seven digits would be kept in computer memory, and the remaining digits would be lost. The last digit would be rounded upward, producing 1.234568. This loss of precision happens because the computer uses a limited amount of storage for floating-point numbers. If you did the same chemical reaction calculation using a variable of type Double, the entire result would be safely held in the number, with no loss of precision.

The Decimal data type is used in financial calculations when you need a great deal of precision. This data type helps prevent rounding errors from creeping into repeated calculations.

The following code demonstrates each floating-point data type:

```
Dim sngTemperature As Single
Dim dblWindSpeed As Double
Dim decBankBalance As Decimal

sngTemperature = 98.6
dblWindSpeed = 35.373659262
decBankBalance = 1234567890.1234567890123456789D
```

Notice that the last line requires a D suffix on the number to identify it as a Decimal literal. Otherwise, Visual Basic would assume that the number was type Double.

Floating-Point Literals

Floating-point literals can be written in either fixed-point or scientific notation. The number 47281.97, for example, would be written in scientific notation as 4.728197×10^4 . Visual Basic requires the letter E just before the exponent in scientific notation. So, our sample number would be written in Visual Basic like this:

```
4.728197E+4
```

The + sign after the E is optional. Here is an example of a value having a negative exponent:

```
4.623476E-2
```

Scientific notation is particularly useful for very large numbers. Instead of writing a value such as 12340000000000000000000000000.0, for example, it is easier to write 1.234E+31.

Boolean Data Type

A Boolean type variable can only hold one of two possible values: *True* or *False*. The values True and False are built-in Visual Basic keywords. The word Boolean is named after George Boole, a famous mathematician of the nineteenth century. (His Boolean algebra is the basis for all modern computer arithmetic.)

We use Boolean variables to hold information that is either true or false. The standard naming prefix for Boolean variables is `bln`. Here is an example:

```
Dim blnIsRegistered As Boolean
blnIsRegistered = True
```

We will begin using Boolean variables in Chapter 4.

Char Data Type

Variables of the Char data type can hold a single Unicode character. Unicode characters are the set of values that can represent a large number of international characters in different languages. To assign a character literal to a Char variable, enclose the character in double quotations marks, followed by a lowercase “c”. The standard naming prefix for Char variables is `chr`. The following is an example:

```
Dim chrLetter As Char
chrLetter = "A"c
```

String Data Type

A variable of type String can hold between zero and about 2 billion characters. The characters are stored in sequence. A string literal, as you have seen earlier, is always enclosed in quotation marks. In the following code, a string variable is assigned various string literals:

```
Dim strName As String
strName = "Jose Gonzalez"
```

The standard naming prefix for String variables is `str`.

An empty string literal can be coded as `""` or by the special identifier named `String.Empty`:

```
strName = ""
strName = String.Empty
```

Date Data Type

A variable of type Date can hold date and time information. Date variables are assigned a prefix of `dat` or `dtm`. You can assign a date literal to a Date variable, as shown here:

```
Dim dtmBirth As Date
dtmBirth = #5/1/2009#
```

Notice that the Date literal is enclosed in # symbols. A variety of date and time formats are permitted. All of the following Date literals are valid:

```
#12/10/2009#
#8:45:00 PM#
#10/20/2009 6:30:00 AM#
```

A Date literal can contain a date, a time, or both. When specifying a time, if you omit AM or PM, the hours value is assumed to be based on a 24-hour clock. If you supply a date without the time, the time portion of the variable defaults to 12:00 AM.

In Tutorial 3-6, you will assign text to a variable.



Tutorial 3-6: Assigning text to a variable

In this tutorial, you will modify a program that assigns the contents of text boxes to a string variable.

Step 1: Open the *Variable Demo* project from the student sample programs folder named *Chap3\Variable Demo*.

Step 2: View the *Form1* form in the *Design* window, as shown in Figure 3-16.

Step 3: Double-click the *Show Name* button, which opens the *Code* window. Type the following lines, shown in bold:

```
Private Sub btnShowName_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnShowName.Click

    ' Declare a string variable to hold the full name.
    Dim strFullName As String

    ' Combine the first and last names
    ' and copy the result to lblFullName
    strFullName = txtFirstName.Text & " " & txtLastName.Text
    lblFullName.Text = strFullName
End Sub
```

Figure 3-16 *Variable Demo* application, *Form1*

Step 4: In the *Design* window, double-click the *Clear* button and insert the following lines in the *Code* window (shown in bold):

```
Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click
```

```

    ' Clear TextBox controls and the Label
    txtFirstName.Clear()
    txtLastName.Clear()
    lblFullName.Text = String.Empty

    ' Set focus to first TextBox control
    txtFirstName.Focus()
End Sub

```

Step 5: In the *Design* window, double-click the *Exit* button and insert the following lines in the *Code* window (shown in bold):

```

Private Sub btnExit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click

    ' Close the application window
    Me.Close()
End Sub

```

Step 6: Save the project.

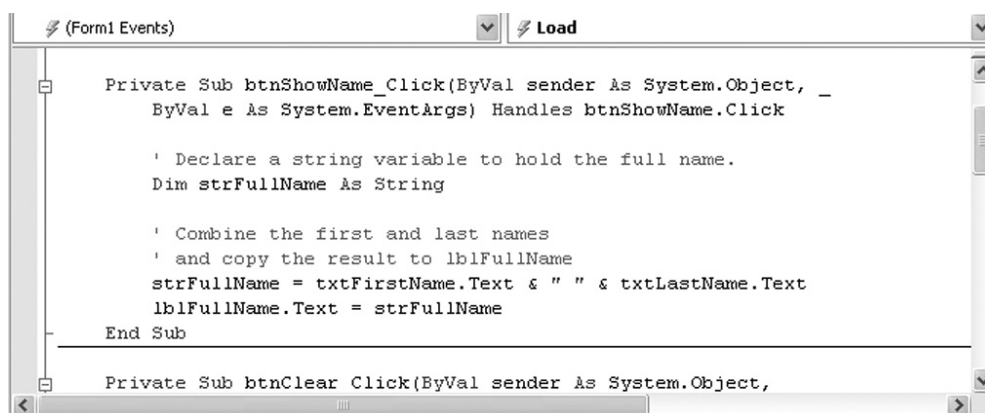
Step 7: Run the program, type in a name, and click the *Show Name* button. The output should look similar to that shown in Figure 3-17.

Figure 3-17 *Variable Demo* application, running



TIP: Code outlining is a Visual Studio tool that lets you expand and collapse sections of code. As your programs get longer, it is helpful to collapse procedures you have already written. Then you can concentrate on new sections of code. For example, Figure 3-18a shows the `btnShowName_Click` procedure from the *Variable Demo* application. A minus sign (–) appears next to its heading, and a horizontal line separates it from the next procedure.

If we click the minus sign next to an expanded procedure, it collapses into a single line showing its name (Figure 3-18b). You can modify outlining options by right-clicking in the *Code* window and selecting *Outlining*.

Figure 3-18a Expanded btnShowName_Click procedure**Figure 3-18b** Collapsed btnShowName_Click procedure

Variable Naming Rules and Conventions

Naming Rules

Just as there are rules and conventions for naming controls, there are rules and conventions for naming variables. Naming rules must be followed because they are part of Visual Basic syntax. The following are Visual Basic's naming rules for variables:

- The first character must be a letter or an underscore character.
- After the first character, you may use letters, numeric digits, and underscore characters.
- Variable names cannot contain spaces or periods.
- Variable names cannot be Visual Basic keywords. For example, words such as `Dim`, `Sub`, and `Private` are predefined in Visual Basic, and their use as variables would confuse the compiler.

Naming Conventions

Naming conventions are guidelines based on recommendations of professional designers, programmers, and educators. If you follow a consistent naming convention, your program source code will be easier to read and understand. In this book, we use a convention in which a three-letter prefix is used in variable names to identify their data type. Table 3-3 lists the variable naming prefixes that we use in this book.

Aside from the variable's prefix, we follow a common capitalization style used in most programming textbooks. After the variable's type prefix, the next letter should be capitalized. Subsequent characters should be lowercase, except the first letter of each word. All variables listed in Table 3-3 follow this rule.

Table 3-3 Recommended prefixes for variable names

Variable Type	Prefix	Examples
Boolean	bln	blnContinue, blnHasRows
Byte	byt	bytInput, bytCharVal
Char	chr	chrSelection, chrMiddleInitial
Date, DateTime	dat or dtm	datWhenPublished, dtmBirthDate
Decimal	dec	decWeeklySalary, decGrossPay
Double	dbl	dblAirVelocity, dblPlanetMass
Integer	int	intCount, intDaysInPayPeriod
Long	lng	lngElapsedSeconds
Object	obj	objStudent, objPayroll
Short	shrt	shrtCount
Single	sng	sngTaxRate, sngGradeAverage
String	str	strLastName, strAddress



TIP: *Be descriptive!* All nontrivial variables should have descriptive names. Avoid short names such as *gp*, *n*, *ldp*, *G5*, and so on. No one reading program code understands short, cryptic variable names. In fact, if you happen to read one of your old programs a few months after writing it, you will depend on descriptive variables to help you remember the meaning of your code.

Variable Declarations and the IntelliSense Feature

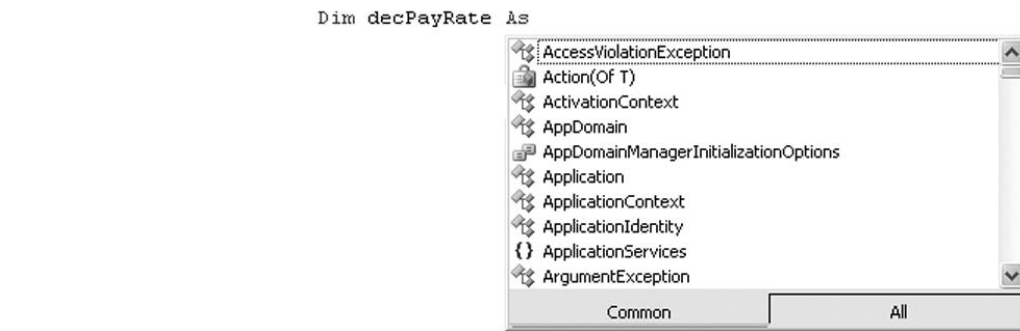
When you are entering a variable declaration, Visual Studio's IntelliSense feature helps you fill in the data type. Suppose you begin to type a variable declaration such as the following:

```
Dim decPayRate As
```

If you press the **[Spacebar]** at this point, a list box appears with all the possible data types in alphabetical order, as shown in Figure 3-19. When the list box appears, type the first few letters of the data type name, and the box will highlight the data type that matches what you have typed. For example, after you type *dec* the Decimal data type will be highlighted. Press the **[Tab]** key to select the highlighted data type.



TIP: You can use the arrow keys or the mouse with the list box's scroll bar to scroll through the list. Once you see the desired data type, double-click it with the mouse.

Figure 3-19 IntelliSense list box

Default Values and Initialization

When a variable is first created it is assigned a default value. Variables with a numeric data type (such as Byte, Decimal, Double, Integer, Long, and Single) are assigned the value 0. Boolean variables are initially assigned the value False, and Date variables are assigned the value 12:00:00 AM, January 1 of year 1. String variables are automatically assigned a special value called *Nothing*.

You may also specify a starting value in the `Dim` statement. This is called **initialization**. Here is an example:

```
Dim intUnitsSold As Integer = 12
```

This statement declares `intUnitsSold` as an integer and assigns it the starting value 12. Here are other examples:

```
Dim strLastName As String = "Johnson"
Dim blnIsFinished As Boolean = True
Dim decGrossPay As Decimal = 2500
Dim chrMiddleInitial As Char = "E"c
```

Forgetting to initialize variables can lead to program errors. Unless you are certain a variable will be assigned a value before being used in an operation, always initialize it. This principle is particularly true with string variables. Performing an operation on an uninitialized string variable often results in a runtime error, causing the program to halt execution because the value *Nothing* is invalid for many operations. To prevent such errors, always initialize string variables or make sure they are assigned a value before being used in other operations. A good practice is to initialize string variables with an empty string, as shown in the following statement:

```
Dim strName As String = String.Empty
```



Checkpoint

- 3.13 What is a variable?
- 3.14 Show an example of a variable declaration.
- 3.15 Which of the following variable names are written with the convention used in this book?
 - a. `decintrestate`
 - b. `InterestRateDecimal`
 - c. `decInterestRate`

- 3.16 Indicate whether each of the following is a legal variable name. If it is not, explain why.
- a. count
 - b. rate*Pay
 - c. deposit.amount
 - d. down_payment
- 3.17 What default value is assigned to each of the following variables?
- a. Integer
 - b. Single
 - c. Boolean
 - d. Byte
 - e. Date
- 3.18 Write a Date literal for the following date and time: 5:35:00 PM on February 20, 2008.
- 3.19 *Bonus question:* Find out which famous Microsoft programmer was launched into space in early 2007. Was this programmer connected in any way to Visual Basic?

3.3 Performing Calculations

CONCEPT: Visual Basic has powerful arithmetic operators that perform calculations with numeric variables and literals.



VideoNote

Problem
Solving with
Variables

There are two basic types of operators in Visual Basic: unary and binary. These reflect the number of operands an operator requires. A **unary operator** requires only a single operand. The negation operator, for example, causes a number to be negative:

-5

It can be applied to a variable. The following line negates the value in `intCount`:

`-intCount`

A **binary operator** works with two operands. The addition operator (+) is binary because it uses two operands. The following mathematical expression adds the values of two numbers:

`5 + 10`

The following adds the values of two variables:

`intA + intB`

Table 3-4 lists the binary arithmetic operators in Visual Basic. Addition, subtraction, multiplication, division, and exponentiation can be performed on both integer and floating-point data types. Only two operations (integer division and modulus) must be performed on integer types.

Table 3-4 Arithmetic operators in Visual Basic

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Floating-point division
\	Integer division
MOD	Modulus (remainder from integer division)
^	Exponentiation ($x^y = x^y$)

Addition

The addition operator (+) adds two values, producing a sum. The values can be literals or variables. The following are examples of valid addition expressions:

```
intA + 10
20 + intB
```

The question is, what happens to the result? Ordinarily, it is assigned to a variable, using the assignment operator. In the following statement, intC is assigned the sum of the values from intA and intB:

```
intC = intA + intB
```

This operation happens in two steps. First, the addition takes place. Second, the sum is assigned to the variable on the left side of the = sign.

The following example adds the contents of two Double variables that hold rainfall measurements for the months of March and April:

```
dblCombined = dblMarchRain + dblAprilRain
```

Subtraction

The subtraction operator (−) subtracts the right-hand operand from the left-hand operand. In the following, the variable intC will contain the difference between intA and intB:

```
intC = intA − intB
```

Alternatively, the difference might be assigned back to the variable intA:

```
intA = intA − intB
```

The following statement uses Decimal variables. It subtracts an employee’s tax amount from his or her gross pay, producing the employee’s net pay:

```
decNetPay = decGrossPay − decTax
```

Addition and Subtraction in Applications

How can addition and subtraction statements be useful in an application? Suppose a college registration program needs to add the credits completed by a student during two semesters (Fall and Spring). First, the variables would be declared:

```
Dim intFallCredits, intSpringCredits, intTotalCredits As Integer
```

Then the application would assign values to `intSpringCredits` and `intFallCredits`, perhaps by asking for their input from the user. Finally, the program would calculate the total credits for the year:

```
intTotalCredits = intFallCredits + intSpringCredits
```

Multiplication

The multiplication operator (`*`) multiplies the right-hand operand by the left-hand operand. In the following statement, the variable `intC` is assigned the product of multiplying `intA` and `intB`:

```
intC = intA * intB
```

The following statement uses `Decimal` variables to multiply an item's price by the sales tax rate, producing a sales tax amount:

```
decTaxAmount = decItemPrice * decTaxRate
```

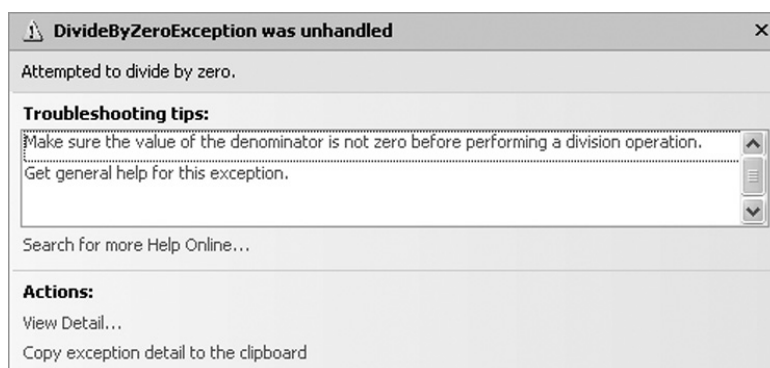
Floating-Point Division

The floating-point division operator (`/`) divides one floating-point value by another. The result, called the quotient, is also a floating-point number. For example, the following statement divides the total points earned by a basketball team by the number of players, producing the average points per player:

```
dblAverage = dblTotalPoints / dblNumPlayers
```

If you try to divide by zero, the program will stop and produce an error message. Suppose `dblNumPlayers` were equal to zero. Then the message dialog shown in Figure 3-20 would appear when the program ran.

Figure 3-20 Error generated because of division by zero



Integer Division

The integer division operator (`\`) divides one integer by another, producing an integer result. For example, suppose we know the number of minutes it will take to finish a job, and we want to calculate the number of hours that are contained in that many minutes. The following statement uses integer division to divide the `intMinutes` variable by 60, giving the number of hours as a result:

```
intHours = intMinutes \ 60
```

Integer division does not save any fractional part of the quotient. The following statement, for example, produces the integer 3:

```
intQuotient = 10 \ 3
```

Modulus

The modulus operator (MOD) performs integer division and returns only the remainder. The following statement assigns 2 to the variable named `intRemainder`:

```
intRemainder = 17 MOD 3
```

Note that 17 divided by 3 equals 5, with a remainder of 2. Suppose a job is completed in 174 minutes, and we want to express this value in both hours and minutes. First, we can use integer division to calculate the hours (2):

```
intTotalMinutes = 174
intHours = intTotalMinutes \ 60
```

Next, we use the MOD operator to calculate the remaining minutes (54):

```
intMinutes = intTotalMinutes Mod 60
```

Now we know that the job was completed in 2 hours, 54 minutes.

Exponentiation

Exponentiation calculates a variable *x* taken to the power of *y* when written in the form *x* ^ *y*. The value it returns is of type Double. For example, the following statement assigns 25.0 to `dblResult`:

```
dblResult = 5.0 ^ 2.0
```

You can use integers as operands, but the result will still be a Double:

```
dblResult = intX ^ intY
```

Negative and fractional exponents are permitted.

Getting the Current Date and Time

Your computer system has an internal clock that calculates the current date and time. Visual Basic provides the functions listed in Table 3-5, which allow you to retrieve the current date, time, or both from your computer. Functions are commands recognized by Visual Basic that return useful information.

Table 3-5 Date and time functions

Function	Description
Now	Returns the current date and time from the system
TimeOfDay	Returns the current time from the system, without the date
Today	Returns the current date from the system, without the time

The following code demonstrates how to use the `Now` function:

```
Dim dtmSystemDate As Date
dtmSystemDate = Now
```

After the code executes, `dtmSystemDate` will contain the current date and time, as reported by the system. The `TimeOfDay` function retrieves only the current time from the system, demonstrated by the following statement.

```
dtmSystemTime = TimeOfDay
```

After the statement executes, `dtmSystemTime` will contain the current time, but not the current date. Instead, it will contain the date January 1 of year 1. The `Today` function retrieves only the current date from the system, demonstrated by the following statement:

```
dtmSystemDate = Today
```

After the statement executes, `dtmSystemDate` will contain the current date, but will not contain the current time. Instead, it will contain the time 00:00:00.



TIP: Later in this chapter you will see how to use the `ToString` method to display only the date value or time value inside a `Date` variable.

Variable Scope

Every variable has a scope and a lifetime. A variable's **scope** refers to the part of a program where the variable is visible and may be accessed by programming statements. There are three types of variable scope:

- A variable declared inside a procedure is called a *local variable*. This type of variable is only visible from its declaring statement to the end of the same procedure. When the procedure ends, the variable is destroyed.
- If a variable is declared inside a class, but outside of any procedure, it is called a *class-level variable*.
- If a variable is declared outside of any class or procedure, it is called a *global variable*.

Declaring Variables Before They Are Used

The first rule of scope is that a variable cannot be used before it is declared. Visual Basic executes statements in a procedure in sequential order, from the first statement to the last. If a programming statement attempts to use a variable before it has been declared, an error occurs. For example, in the following code sequence line 3 generates an error because `dblAverage` has not been declared yet:

```
1: Dim dblTotal As Double = 500.0
2: Dim dblCount As Double = 10.2
3: dblAverage = dblTotal / dblCount           ' ERROR
4: Dim dblAverage As Double
```

If we just reverse the statements in lines 3 and 4, the program compiles correctly:

```
1: Dim dblTotal As Double = 500.0
2: Dim dblCount As Double = 10.2
3: Dim dblAverage As Double
4: dblAverage = dblTotal / dblCount           ' OK
```

Using a Variable That Is Not in the Current Scope

Another rule of scope is that a variable declared inside a procedure is visible only to statements inside the same procedure. For example, suppose an application has the following two event procedures. The variable `intValue` is declared in the `btnButtonOne_Click` procedure. An error results when a statement in the `btnButtonTwo_Click` procedure attempts to access the `intValue` variable:

```
Private Sub btnButtonOne_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ButtonOne.Click

    ' Declare an Integer variable named intValue.
    Dim intValue As Integer
```

```

        ' Assign a value to the variable.
        intValue = 25
    End Sub

    Private Sub btnButtonTwo_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonTwo.Click

        ' Attempt to assign a value to the
        ' intValue variable. This will cause an error!
        intValue = 32
    End Sub

```

What if we declared `intValue` again in the `btnButtonTwo` procedure? Then the program would compile, and we would have created two different variables having the same name. Each has its own scope, separate from the other:

```

    Private Sub btnButtonOne_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonOne.Click

        ' Declare an Integer variable named intValue.
        Dim intValue As Integer
        ' Assign a value to the variable.
        intValue = 25
    End Sub

    Private Sub btnButtonTwo_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonTwo.Click

        ' Declare an Integer variable named intValue.
        Dim intValue As Integer
        ' Assign a value to the variable.
        intValue = 32
    End Sub

```

Variables having the same name are completely separate from each other if they are declared in different procedures.

Using the Same Variable Name Twice in the Same Scope

Another scope rule says that you cannot declare two variables by the same name within the same scope. The following example violates the rule:

```

    Private Sub btnButtonOne_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles ButtonOne.Click

        Dim intValue As Integer
        intValue = 25

        Dim intValue As Integer      ' Error!
        intValue = 32
    End Sub

```

Combined Assignment Operators

Quite often, programs have assignment statements in the following form:

```
intValue = intValue + 1
```

On the right-hand side of the assignment operator, 1 is added to `intValue`. The result is then assigned to `intValue`, replacing the value that was previously stored there. Similarly, the following statement subtracts 5 from `intValue`.

```
intValue = intValue - 5
```

Table 3-6 shows examples of similar statements. Assume that the variable `x` is set to 6 prior to each statement's execution.

Table 3-6 Assignment statements (Assume `x = 6` prior to each statement's execution)

Statement	Operation Performed	Value of <code>x</code> after the Statement Executes
<code>x = x + 4</code>	Adds 4 to <code>x</code>	10
<code>x = x - 3</code>	Subtracts 3 from <code>x</code>	3
<code>x = x * 10</code>	Multiplies <code>x</code> by 10	60
<code>x = x / 2</code>	Divides <code>x</code> by 2	3

Assignment operations are common in programming. For convenience, Visual Basic offers a special set of operators designed specifically for these jobs. Table 3-7 shows the **combined assignment operators**, or **compound operators**.

Table 3-7 Combined assignment operators

Operator	Example Usage	Equivalent To
<code>+=</code>	<code>x += 2</code>	<code>x = x + 2</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>
<code>*=</code>	<code>x *= 10</code>	<code>x = x * 10</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>\=</code>	<code>x \= y</code>	<code>x = x \ y</code>
<code>&=</code>	<code>strName &= lastName</code>	<code>strName = strName & lastName</code>

Operator Precedence

It is possible to build **mathematical expressions** with several operators. The following statement assigns the sum of 17, `x`, 21, and `y` to the variable `intAnswer`.

```
intAnswer = 17 + x + 21 + y
```

Some expressions are not that straightforward, however. Consider the following statement:

```
dblOutcome = 12 + 6 / 3
```

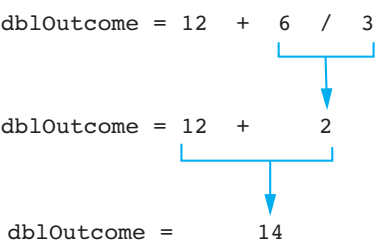
What value will be stored in `dblOutcome`? If the addition takes place before the division, then `dblOutcome` will be assigned 6. If the division takes place first, `dblOutcome` will be assigned 14. The correct answer is 14 because the division operator has higher **precedence** than the addition operator.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence executes first. Multiplication and division have higher precedence than addition and subtraction, so `12 + 6 / 3` works like this:

- 6 is divided by 3, yielding a result of 2.
- 12 is added to 2, yielding a result of 14.

It can be diagrammed as shown in Figure 3-21.

Figure 3-21 `dblOutcome = 12 + 6 / 3`



The precedence of the arithmetic operators, from highest to lowest, is as follows:

- 1. Exponentiation (the `^` operator)
- 2. Multiplication and division (the `*` and `/` operators)
- 3. Integer division (the `\` operator)
- 4. Modulus (the `MOD` operator)
- 5. Addition and subtraction (the `+` and `-` operators)

The multiplication and division operators have the same precedence. This is also true of the addition and subtraction operators. When two operators with the same precedence share an operand, the operator on the left executes before the operator on the right.

Table 3-8 shows some example mathematical expressions with their values.

Table 3-8 Mathematical expressions and their values

Expression	Value
<code>5 + 2 * 4</code>	13
<code>2^3 * 4 + 3</code>	35
<code>10 / 2 - 3</code>	2
<code>8 + 12 * 2 - 4</code>	28
<code>6 - 3 * 2 + 7 - 1</code>	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the sum of `x`, `y`, and `z` is divided by 3. The result is assigned to `dblAverage`.

```
dblAverage = (x + y + z) / 3
```

Without the parentheses, however, `z` would be divided by 3, and the result added to the sum of `x` and `y`. Table 3-9 shows more expressions and their values.

Table 3-9 Additional mathematical expressions and their values

Expression	Value
<code>(5 + 2) * 4</code>	28
<code>10 / (5 - 3)</code>	5
<code>8 + 12 * (6 - 2)</code>	56
<code>(6 - 3) * (2 + 7) / 3</code>	9

More about Mathematical Operations: Converting Mathematical Expressions to Programming Statements

In algebra, the mathematical expression $2xy$ describes the value 2 times x times y . Visual Basic, however, requires an operator for any mathematical operation. Table 3-10 shows some mathematical expressions that perform multiplication and the equivalent Visual Basic expressions.

Table 3-10 Visual Basic equivalents of mathematical expressions

Mathematical Expression	Operation	Visual Basic Equivalent
$6B$	6 times B	<code>6 * B</code>
$(3)(12)$	3 times 12	<code>3 * 12</code>
$4xy$	4 times x times y	<code>4 * x * y</code>



Checkpoint

- 3.20 What value will be stored in `dblResult` after each of the following statements executes?
- `dblResult = 6 + 3 * 5`
 - `dblResult = 12 / 2 - 4`
 - `dblResult = 2 + 7 * 3 - 6`
 - `dblResult = (2 + 4) * 3`
 - `dblResult = 10 \ 3`
 - `dblResult = 6 ^ 2`
- 3.21 What value will be stored in `intResult` after each statement executes?
- `intResult = 10 MOD 3`
 - `intResult = 47 MOD 15`
- 3.22 Write a statement that assigns the current time of day to the variable `dtmThisTime`.
- 3.23 Write a statement that assigns the current date and time to a variable named `dtmCurrent`.
- 3.24 Explain the meaning of the term scope when applied to variables.
- 3.25 What will be the final value of `dblResult` in the following sequence?
- ```
Dim dblResult As Double = 3.5
dblResult += 1.2
```
- 3.26 What will be the final value of `dblResult` in the following sequence?
- ```
Dim dblResult As Double = 3.5
dblResult *= 2.0
```

3.4

Mixing Different Data Types

Implicit Type Conversion

When you assign a value of one data type to a variable of another data type, Visual Basic attempts to convert the value being assigned to the data type of the receiving variable.

This is known as an **implicit type conversion**. Suppose we want to assign the integer 5 to a variable of type Single named `sngNumber`:

```
Dim sngNumber As Single = 5
```

When the statement executes, the integer 5 is automatically converted into a single-precision real number, which is then stored in `sngNumber`. This conversion is a **widening conversion** because no data is lost.

Narrowing Conversions

If you assign a real number to an integer variable, Visual Basic attempts to perform a **narrowing conversion**. Often, some data is lost. For example, the following statement assigns 12.2 to an integer variable:

```
Dim intCount As Integer = 12.2          'intCount = 12
```

Assuming for the moment that Visual Basic is configured to accept this type of conversion, the 12.2 is rounded downward to 12. Similarly, the next statement rounds upward to the nearest integer when the fractional part of the number is .5 or greater:

```
Dim intCount As Integer = 12.5          'intCount = 13
```

Another narrowing conversion occurs when assigning a Double value to a variable of type Single. Both hold floating-point values, but Double permits more significant digits:

```
Dim dblOne As Double = 1.2342376
Dim sngTwo As Single = dblOne          'sngTwo = 1.234238
```

The value stored in `sngTwo` is rounded up to 1.234238 because variables of type Single can only hold seven significant digits.

Converting Strings to Numbers

Under some circumstances, Visual Basic will try to convert string values to numbers. In the following statement, "12.2" is a string containing a numeric expression:

```
Dim strTemp As String = "12.2"
```

The string "12.2" is a string of characters in a numeric-like format that cannot be used for calculations. When the following statements execute, the string "12.2" is converted to the number 12.2:

```
Dim sngTemperature As Single
sngTemperature = "12.2"
```

If we assign the string "12.2" to an Integer variable, the result is rounded downward to 12:

```
Dim intCount As Integer
intCount = "12.2"
```

A similar effect occurs when the user enters a number into a TextBox control. Usually we want to assign the contents of the TextBox's Text property to a numeric variable. The Text property by definition holds strings, so its contents are implicitly converted to a number when we assign it to a numeric variable, as shown here:

```
intCount = txtCount.Text
```

Figure 13-22 shows an example of the steps that occur when the user's input into a TextBox is assigned to a numeric variable.

Figure 3-22 Implicit conversion of TextBox into a numeric variable

1. The user enters this value into txtCount:

26
2. An assignment statement in an event procedure executes:

intCount = txtCount.Text
3. The contents of the Text property is converted to a numeric value:

26

Option Strict

Visual Basic has a configuration option named *Option Strict* that determines whether certain implicit conversions are legal. If you set *Option Strict* to *On*, only widening conversions are permitted (such as Integer to Single). Figure 3-23 shows how implicit conversion between numeric types must be in a left-to-right direction in the diagram. A Decimal value can be assigned to a variable of type Single, an Integer can be assigned to a variable of type Double, and so on. If, on the other hand, *Option Strict* is set to *Off*, all types of numeric conversions are permitted, with possible loss of data.

To set *Option Strict* for a single project, right-click the project name in the *Solution Explorer* window, select *Properties*, and then select the *Compile* tab, as shown in Figure 3-24. From the *Option Strict* drop-down list, you can select *On* or *Off*.

We recommend setting *Option Strict* to *On*, so Visual Basic can catch errors that result when you accidentally assign a value of the wrong type to a variable. When set to *On*, *Option Strict* forces you to use a conversion function, making your intentions clear. This approach helps to avoid runtime errors.

Figure 3-23 Conversions permitted with Option Strict On

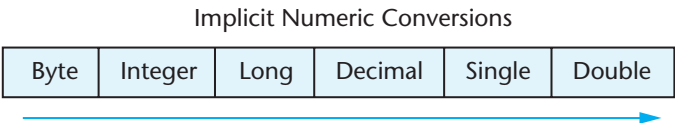
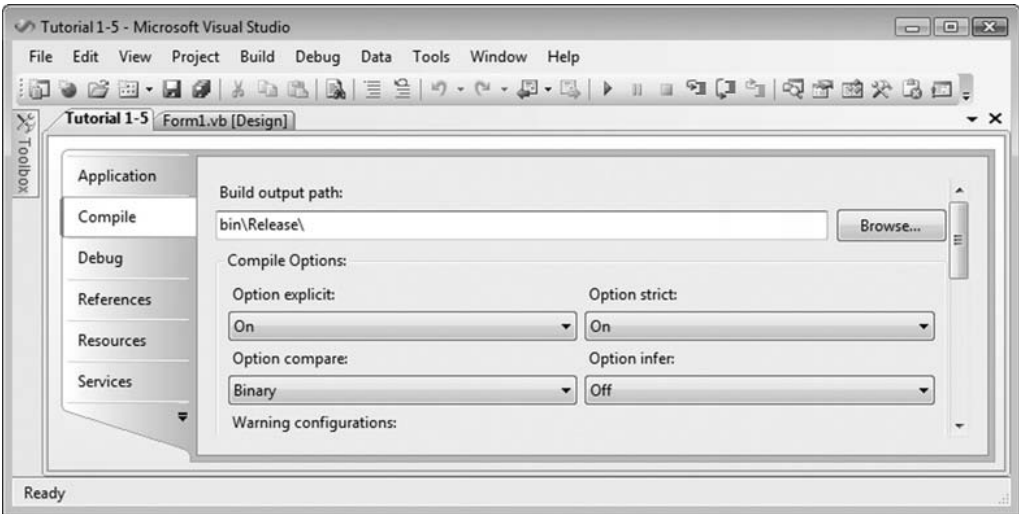


Figure 3-24 Project Properties window



2009933343

Type Conversion Runtime Errors

The following statement does not compile with *Option Strict* set to *On*:

```
Dim intCount As Integer = "abc123"
```

But if *Option Strict* has been set to *Off*, a program containing this statement will compile without errors. Then, when the program executes, it will stop when it reaches this statement because the string “abc123” contains characters that prevent the string from being converted to a number. The program will generate a runtime error message. Recall from Chapter 2 that runtime errors are errors that occur while the application is running. The type of error in this case is also known as a **type conversion error** or **type mismatch error**.

Runtime errors are harder to catch than syntax errors because they occur while a program is running. Particularly in programs where the user has a number of choices (mouse clicks, keyboard input, menus, and so on), it is very difficult for a programmer to predict when runtime errors might occur. Nevertheless, there are certain conversion functions shown in the next section that reduce the chance of this type of mishap.



TIP: Can the best-trained programmers avoid runtime errors? To discover the answer to that question, spend some time using commercial Web sites and notice all the little things that go wrong. Software reliability is particularly hard to guarantee when users have lots of choices.

Literals

Literals (constants) have specific types in Visual Basic. Table 3-11 lists the more common types. Many literals use a suffix character (C, D, @, R, I, L, F, S, !) to identify their type.

Table 3-11 Representing literals (constants) in Visual Basic

Type	Description	Example
Boolean	Keywords <i>True</i> and <i>False</i>	True
Byte	Sequence of decimal digits between 0 and 255	200
Char	Single letter enclosed in double quotes followed by the lowercase letter C	"A"c
Date	Date and/or time representation enclosed in # symbols	#1/20/05 3:15 PM#
Decimal	Optional leading sign, sequence of decimal digits, optional decimal point and trailing digits, followed by the letter D or @	+32.0D 64@
Double	Optional leading sign, sequence of digits with a decimal point and trailing digits, followed by optional letter R	3.5 3.5R
Integer	Optional leading sign, sequence of decimal digits, followed by optional letter I	-3054I +26I
Long	Optional leading sign, sequence of decimal digits, followed by the letter L	40000000L
Short	Optional leading sign, sequence of decimal digits, followed by the letter S	12345S
Single	Optional leading sign, sequence of digits with a decimal point and trailing digits, followed by the letter F or !	26.4F 26.4!
String	Sequence of characters surrounded by double quotes	"ABC" "234"

It's important to know literal types because you often assign literal values to variables. If the receiving variable in an assignment statement has a different type from the literal, an implied conversion takes place. If *Option Strict* is *On*, some conversions are automatic and others generate errors. You need to understand why.

Most of the time, you will be inclined to use no suffix with numeric literals. As long as the receiving variable's data type is compatible with the literal's data type, you have no problem:

```
Dim lngCount As Long = 25           'Integer assigned to Long
Dim dblSalary As Double = 2500.0    'Double assigned to Double
```

In the following example, however, we're trying to assign a Double to a Decimal, which is not permitted when *Option Strict* is *On*:

```
Dim decPayRate As Decimal = 35.5    'Double to Decimal (?)
```

If you know how to create numeric literals, all you have to do is append a *D* to the number. The following is valid:

```
Dim decPayRate As Decimal = 35.5D
```

Named Constants

You have seen several programs and examples where numbers and strings are expressed as literal values. For example, the following statement contains the literal numeric value 0.129:

```
dblPayment = dblPrincipal * 0.129
```

Suppose this statement appears in a banking program that calculates loan information.

In such a program, two potential problems arise. First, it is not clearly evident to anyone other than the original programmer what the number 0.129 is. It appears to be an interest rate, but in some situations there are other fees associated with loan payments. How can you determine the purpose of this statement without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, if the rate changes from 12.9% to 13.2% the programmer will have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A **named constant** is like a variable whose content is read-only, and cannot be changed by a programming statement while the program is running. The following is the general form of a named constant declaration:

```
Const ConstantName As DataType = Value
```

Here is an example of a named constant declaration:

```
Const dblINTEREST_RATE As Double = 0.129
```

It looks like a regular variable declaration except for the following differences:

- The word **Const** is used instead of **Dim**.
- An initialization value is required.
- By convention, all letters after the prefix are capitals.
- Words in the name are separated by the underscore character.

The keyword **Const** indicates that you are declaring a named constant instead of a vari-

able. The value given after the = sign is the value of the constant throughout the program's execution.

A value must be assigned when a named constant is declared or an error will result. An error will also result if any statements in the program attempt to change the contents of a named constant.

One advantage of using named constants is that they help make programs self-documenting. The statement

```
dblPayment = dblPrincipal * 0.129
```

can be changed to read

```
dblPayment = dblPrincipal * dblINTEREST_RATE
```

A new programmer can read the second statement and know what is happening. It is evident that `dblPrincipal` is being multiplied by the interest rate.

Another advantage to using named constants is that consistent changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the value assigned to the named constant in its declaration is the only value that needs to be modified. If the rate increases to 13.2% the declaration is changed to the following:

```
Const dblINTEREST_RATE as Double = 0.132
```

Every statement that uses `dblINTEREST_RATE` will use the new value.

It is also useful to declare named constants for common values that are difficult to remember. For example, any program that calculates the area of a circle must use the value π , which is 3.14159. This value could easily be declared as a named constant, as shown in the following statement:

```
Const dblPI as Double = 3.14159
```



TIP: In this chapter, named constants are declared at the beginnings of procedures. In a later chapter we will demonstrate how named constants can also be declared at the class level.

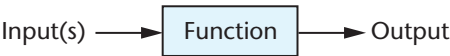
Explicit Type Conversions

Let's assume for the current discussion that *Option Strict* is set to *On*. If you try to perform anything other than a widening conversion, your code will not compile. Visual Basic has a set of conversion functions to solve this problem.

What Is a Function?

A **function** is a named, self-contained body of code, to which you send some input (Figure 3-25). The function performs a predetermined operation and produces a single output. A well-known function in mathematics is the $y = \text{abs}(x)$ function, which finds the absolute value x and stores it in y . The only reason a function might not produce the same output all the time is that it could be affected by the values of variables elsewhere in the program.

Figure 3-25 A function receives input and produces a single output



Visual Basic Conversion Functions

Table 3-12 lists the Visual Basic conversion functions that we will use most often. The input to each conversion function is an expression, which is another name for a constant, a variable, or a mathematical expression (such as 2.0 + 4.2). The following situations require a conversion function:

- When assigning a wider numeric type to a narrower numeric type. In Figure 3-23 (page 131), the arrow pointing from left to right indicates automatic conversions. Any conversion in the opposite direction requires a call to a conversion function. Examples are Long to Integer, Decimal to Long, Double to Single, and Double to Decimal.
- When converting between Boolean, Date, Object, String, and numeric types. These all represent different categories, so they require conversion functions.

Table 3-12 Commonly used type conversion functions

Function	Description
CDate(<i>expr</i>)	Converts a String expression to a Date. Input can also be a Date literal, such as "10/14/2009 1:30 PM".
CDBl(<i>expr</i>)	Converts an expression to a Double. If the input expression is a String, a leading currency symbol (\$) is permitted, as are commas. The decimal point is optional.
CDec(<i>expr</i>)	Converts an expression to a Decimal. If the input expression is a String, a leading currency symbol (\$) is permitted, as are commas. The decimal point is optional.
CInt(<i>expr</i>)	Converts an expression to an Integer. If the input expression is a String, a leading currency symbol (\$) is permitted, as are commas. The decimal point is optional, as are digits after the decimal point.
CStr(<i>expr</i>)	Converts an expression to a String. Input can be a mathematical expression, Boolean value, a date, or any numeric data type.

Details

The CDate function is often used when assigning the contents of a TextBox control to a Date variable. In the next example, the TextBox is named txtBirthDay:

```
Dim datBirthDay As Date = CDate(txtBirthDay.Text)
```

The CDBl function is often used when assigning the contents of a TextBox control to a Double variable.

```
Dim dblSalary As Double = CDBl(txtPayRate.Text)
```

The CDec function is often used when assigning the contents of a TextBox control to a Decimal variable.

```
Dim decPayRate As Decimal = CDec(txtPayRate.Text)
```

Also, CDec is required when assigning a Double value to a Decimal:

```
Dim dblPayRate As Double
Dim decPayRate As Decimal = CDec(dblPayRate)
```

2009933343

You cannot directly assign a numeric literal to a `Decimal` because literals are assumed to be of type `Double`. Instead, you must either use the `CDec` function to convert the literal, or append a `D` suffix character to the literal. The following examples are correct:

```
Dim decPayRate As Decimal = CDec(26.50)
Dim decPayRate2 As Decimal = 26.50D
```

The `CInt` function is required when assigning any floating-point type to an integer:

```
Dim dblWeight As Double = 3.5
Dim intWeight As Integer = CInt(dblWeight)
```

The `CInt` function is also required when assigning a `TextBox` control to an integer:

```
Dim intWeight As Integer = CInt(txtWeight.Text)
```

If You Want to Know More: `CInt` and Rounding

The `CInt` function converts an expression to an `Integer`. If the input value contains digits after the decimal point, a special type of rounding occurs, called *banker's rounding*. Here's how it works:

- If the digit after the decimal point is less than 5, the digits after the decimal point are removed from the number. We say the number is *truncated*.
- If the digit after the decimal point is a 5, the number is rounded toward the nearest even integer.
- If the digit after the decimal point is greater than 5 and the number is positive, it is rounded to the next highest integer.
- If the digit after the decimal point is greater than 5 and the number is negative, it is rounded to the next smallest integer.

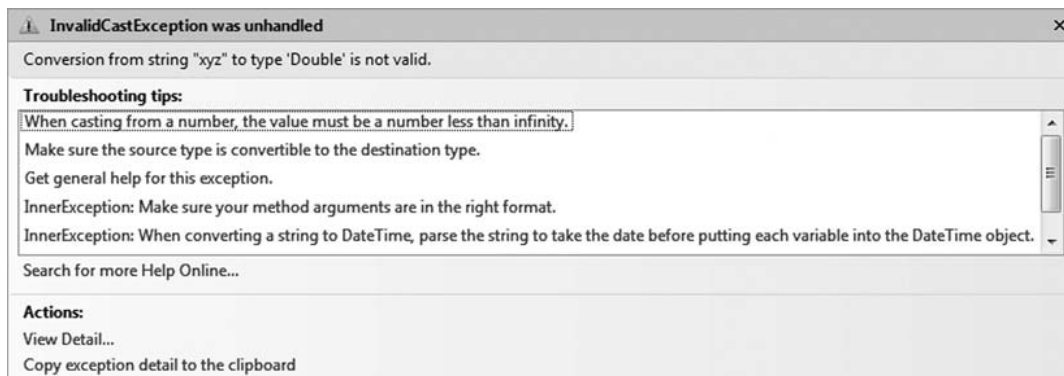
Invalid Conversions

What happens if you call a conversion function, passing a value that cannot be converted? Here's an example:

```
Dim dblSalary As Double
dblSalary = CDb1("xyz")
```

The program stops with a runtime error, displaying the dialog box shown in Figure 3-26. The specific type of error, also known as an **exception**, is called an *InvalidCastException*. The text inside the list box consists of hyperlinks, which take you to specific *Visual Studio Help* pages. Later in this chapter, you will learn how to catch errors like this so the program won't stop.

Figure 3-26 Error displayed when trying to perform an invalid conversion from `String` to `Double`



The Val Function (Optional Topic)

An alternative technique for converting strings to numeric values is to use the `Val` function. The `Val` function converts a string like "34.7" to a number such as 34.7. For example, suppose an application uses a `TextBox` control named `txtInput` and has the following code:

```
Dim intNumber As Integer
intNumber = txtInput.Text
```

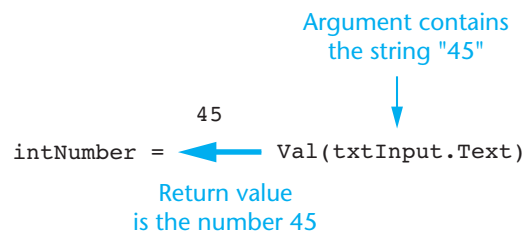
The second statement assigns the contents of the `TextBox` control's `Text` property to an integer variable. To prevent the possibility of a type conversion error we can rewrite the statement to use the `Val` function:

```
intNumber = Val(txtInput.Text)
```

The `Val` function expects its argument to be a string. `Val` converts the string argument to a numeric value and returns the numeric value to the statement that called the function.

In Figure 3-27, the number that `Val` returns is assigned to `intNumber`. If "45" is stored in `txtInput`'s `Text` property, the `Val` function returns the number 45, which is assigned to the `intNumber` variable.

Figure 3-27 The `Val` function



If the string passed to `Val` contains a decimal point, `Val` automatically returns a value of type `Double`:

```
Dim dblOne As Double = Val("10.5")
```

If the string you pass to the `Val` function cannot be converted to a number, it returns zero. In the example, suppose the user enters **\$5000** into the text box. The function assigns a default value of zero to `dblSalary` because the `$` character is not permitted:

```
Dim dblSalary As Double = Val(txtSalary.Text)
```

The `Val` function does not let you know the user made a mistake. For this reason, you usually need to write additional statements that check for a zero value and assume that a user would never purposely enter a salary of zero. Conditional statements are covered in Chapter 4.



TIP: At one time, the `Val` function was the only tool available in Visual Basic for converting strings to numbers. Because other conversion functions do a better job than `Val`, professional programmers prefer not to use it.



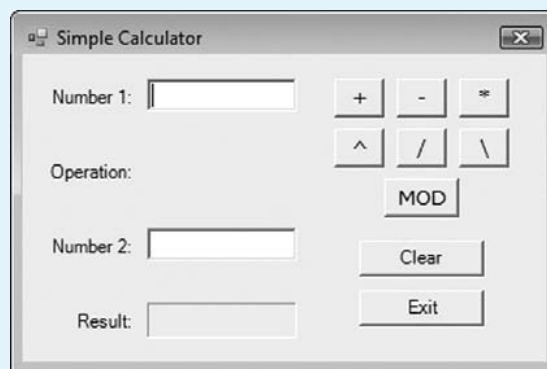
Tutorial 3-7:

Examining a *Simple Calculator* application

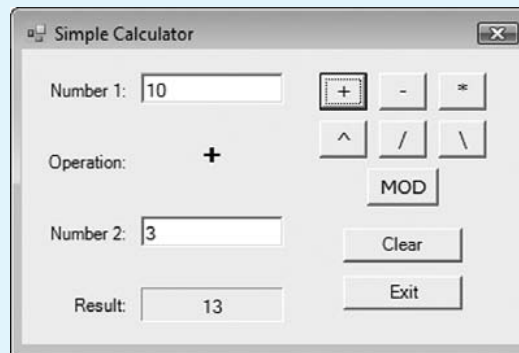
This tutorial examines an application that functions as a simple calculator. The application has two `TextBox` controls, into which you enter numbers. There are buttons for addition, subtraction, multiplication, division, exponentiation, integer division, and modulus. When you click one of these buttons, the application performs a math operation using the two numbers entered into the `TextBox` controls and displays the result in a label.

- Step 1:** Open the *Simple Calculator* project from the Chapter 3 sample programs folder named *Simple Calculator*.
- Step 2:** Click the *Start* button to run the application. The application's form appears, as shown in Figure 3-28.

Figure 3-28 *Simple Calculator* form



- Step 3:** Enter **10** into the *Number 1* `TextBox` control. (In the program source code, this `TextBox` control is named `txtNumber1`.)
- Step 4:** Enter **3** into the *Number 2* `TextBox` control. (In the program source code, this `TextBox` control is named `txtNumber2`.)
- Step 5:** Click the **+** button. Notice that next to the word *Operation* a large plus sign appears on the form. This indicates that an addition operation has taken place. (The plus sign is displayed in a label named `lblOperation`.) The result of $10 + 3$ displays in the *Result* label, which is named `lblResult`. The form appears as shown in Figure 3-29.
- Step 6:** Click the **-** button. Notice that the `lblOperation` label changes to a minus sign and the result of $10 \text{ minus } 3$ displays in the `lblResult` label.
- Step 7:** Click the *****, **^**, **/**, ****, and **MOD** buttons. Each performs a math operation using 10 and 3 as its operands and displays the result in the `lblResult` label.
- Step 8:** If you wish to experiment with other numbers, click the *Clear* button and continue.
- Step 9:** When you are finished, click the *Exit* button.

Figure 3-29 Result of 10 + 3

Step 10: Open *Form1* in the *Design* window, if it is not already open. Double-click the + button on the application form. This opens the *Code* window, with the text cursor positioned in the `btnPlus_Click` event procedure. The code is as follows:

```
Private Sub btnPlus_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnPlus.Click

    ' Perform addition
    Dim dblResult As Double
    lblOperation.Text = "+"
    dblResult = CDbl(txtNumber1.Text) + CDbl(txtNumber2.Text)
    lblResult.Text = CStr(dblResult)
End Sub
```

Step 11: Examine the other event procedures in this application. Most of the arithmetic operator button handlers are similar, except those for the Integer division and Modulus operators. In the Integer division operator handler, for example, the result variable is an integer, and the `CInt` function converts the contents of the `TextBox` controls:

```
Private Sub btnIntegerDivide_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnIntegerDivide.Click

    ' Perform integer division
    Dim intResult As Integer
    lblOperation.Text = "\"
    intResult = CInt(txtNumber1.Text) \ CInt(txtNumber2.Text)
    lblResult.Text = CStr(intResult)
End Sub
```

If You Want to Know More: Full Set of VB Conversion Functions

Table 3-13 contains a more complete list of Visual Basic conversion functions than the one shown earlier. Although you may not use some of these functions all the time, you certainly need to know where to find them.

Table 3-13 Visual Basic type conversion functions

Function	Description
<code>CBool(expr)</code>	Converts an expression to a Boolean value. The expression must be a number, a string that represents a number, or the strings "True" or "False". Otherwise a runtime error is generated. If the expression is nonzero, the function returns <i>True</i> . Otherwise it returns <i>False</i> . For example, <code>CBool(10)</code> and <code>CBool("7")</code> return <i>True</i> , while <code>CBool(0)</code> and <code>CBool("0")</code> return <i>False</i> . If the argument is the string "True", the function returns <i>True</i> , and if the expression is the string "False", the function returns <i>False</i> .
<code>CByte(expr)</code>	Converts an expression to a Byte, which can hold the values 0 through 255. If the argument is a fractional number, it is rounded. If the expression cannot be converted to a value in the range of 0–255, a runtime error is generated.
<code>CChar(expr)</code>	Converts a string expression to a Char. If the string contains more than one character, only the first character is returned. For example, <code>CChar("xyz")</code> returns the character <i>x</i> .
<code>CDate(expr)</code>	Converts an expression to a Date. String expressions must be valid Date literals. For example, <code>CDate("#10/14/2009 1:30 PM#")</code> returns a Date with the value <i>1:30 PM, October 14th, 2009</i> . If the expression cannot be converted to a Date value, a runtime error is generated.
<code>CDBl(expr)</code>	Converts a numeric or string expression to a Double. If the expression converts to a value outside the range of a Double, or is not a numeric value, a runtime error is generated.
<code>CDec(expr)</code>	Converts a numeric or string expression to a Decimal. The <code>CDec</code> function can convert strings starting with a \$ character, such as \$1,200.00. Commas are also permitted. If the expression converts to a value outside the range of a Decimal, or is not a numeric value, a runtime error is generated.
<code>CInt(expr)</code>	Converts a numeric or string expression to an Integer. If the expression converts to a value outside the range of an Integer, or is not a numeric value, a runtime error is generated. Rounds to nearest integer.
<code>CLng(expr)</code>	Converts a numeric or string expression to a Long (long integer). If the expression converts to a value outside the range of a Long, or is not a numeric value, a runtime error is generated.
<code>CObj(expr)</code>	Converts an expression to an Object.
<code>CShort(expr)</code>	Converts a numeric or string expression to a Short (short integer). If the expression converts to a value outside the range of a Short, or is not a numeric value, a runtime error is generated.
<code>CSng(expr)</code>	Converts a numeric or string expression to a Single. If the expression converts to a value outside the range of a Single, or is not a numeric value, a runtime error is generated. The input expression may contain commas, as in "1,234."
<code>CStr(expr)</code>	Converts a numeric, Boolean, Date, or string expression to a String. Input can be an arithmetic expression, a Boolean value, a date, or any numeric data type.



Checkpoint

- 3.27 After the statement `dblResult = 10 \ 3` executes, what value will be stored in `dblResult`?

- 3.28 After each of the following statements executes, what value will be stored in `dblResult`?
- `dblResult = 6 + 3 * 5`
 - `dblResult = 12 / 2 - 4`
 - `dblResult = 2 + 7 * 3 - 6`
 - `dblResult = (2 + 4) * 3`
- 3.29 What value will be stored in `dblResult` after the following statement executes?
- ```
dblResult = CInt("28.5")
```
- 3.30 Will the following statement execute or cause a runtime error?
- ```
dblResult = CDBl("186,478.39")
```
- 3.31 What is a named constant?
- 3.32 Assuming that `intNumber` is an integer variable, what value will each of the following statements assign to it?
- `intNumber = 27`
 - `intNumber = CInt(12.8)`
 - `intNumber = CInt(12.0)`
 - `intNumber = (2 + 4) * 3`
- 3.33 When does a type conversion runtime error occur? (Assume *Option Strict* is *Off*).
- 3.34 Excluding the `Val()` function, which function converts the string "860.2" to value of type `Double`?
- 3.35 How would the following strings be converted by the `CDec` function?
- 48.5000
 - \$34.95
 - 2,300
 - Twelve

3.5 Formatting Numbers and Dates

Users of computer programs generally like to see numbers and dates displayed in an attractive, easy to read format. Numbers greater than 999, for instance, should usually be displayed with commas and decimal points. The value 123456.78 would normally be displayed as "123,456.78".



TIP: Number formatting is dependent on the locale that is used by the computer's Microsoft Windows operating system. That includes Windows XP, Windows Vista, and earlier versions of Windows. *Localization* refers to the technique of adapting your formats for various regions and countries of the world. For example, in North America a currency value is formatted as 123,456.78. In many European countries, the same value is formatted as 123.456,78. In this book, we will only display North American formats, but you can find help on using other types of formats by looking for the topic named *localization* in Visual Studio help.

ToString Method

All numeric and date data types in Visual Basic contain the **ToString** method. This method converts the contents of a variable to a string. The following code segment shows an example of the method's use.

```
Dim intNumber As Integer = 123
lblNumber.Text = intNumber.ToString()
```

In the second statement the number variable's ToString method is called. The method returns the string "123", which is assigned to the Text property of lblNumber.

By passing a formatting string to the ToString method, you can indicate what type of format you want to use when the number or date is formatted. The following statements create a string containing the number 1234.5 in Currency format:

```
Dim dblSample As Double
Dim strResult As String
dblSample = 1234.5
strResult = dblSample.ToString("c")
```

When the last statement executes, the value assigned to strResult is "\$1,234.50". Notice that an extra zero was added at the end because currency values usually have two digits to the right of the decimal point. The value "c" is called a format string. Table 3-14 shows the format strings used for all types of floating-point numbers (Double, Single, and Currency), assuming the user is running Windows in a North American locale. The format strings are not case sensitive, so you can code them as uppercase or lowercase letters. If you call ToString using an integer type (Byte, Integer, or Long), the value is formatted as if it were type Double.

Table 3-14 Standard numeric format strings

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

Number Format

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

```
-2,345.67
```

Fixed-Point Format

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

```
-2345.67
```

Exponential Format

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

```
-2.345670e+003
```

Currency Format

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

(\$2,345.67)

Percent Format

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

–23.45 %

Specifying the Precision

Each numeric format string can optionally be followed by an integer that indicates how many digits to display after the decimal point. For example, the format n3 displays three digits after the decimal point. Table 3-15 shows a variety of numeric formatting examples, based on the North American locale.

Table 3-15 Numeric formatting examples (North American locale)

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

Rounding

Rounding can occur when the number of digits you have specified after the decimal point in the format string is smaller than the precision of the numeric value. Suppose, for example, that the value 1.235 were displayed with a format string of n2. Then the displayed value would be 1.24. If the next digit after the last displayed digit is 5 or higher, the last displayed digit is rounded *away from zero*. Table 3-16 shows examples of rounding using a format string of n2.

Table 3-16 Rounding examples, using the n2 display format string

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

Integer Values with Leading Zeros

Integer type variables (Byte, Integer, or Long) have a special format string, `D` (or `d`), that lets you specify the minimum width for displaying the number. Leading zeros are inserted if necessary. Table 3-17 shows examples.

Table 3-17 Formatting integers, using the `D` (`d`) format string

Integer Value	Format String	Formatted As
23	D	23
23	D4	0023
1	D2	01

Formatting Dates and Times

When you call the `ToString` method using a `Date` or `DateTime` variable, you can format it as a short date, short time, long date, and so on. Table 3-18 lists the most commonly used format strings for dates and times. The following example creates a string containing "8/10/2009", called the short date format.

```
Dim dtmSample As Date = "#8/10/2009#"
Dim strResult As String = dtmSample.ToString("d")
```

The following example gets the current date and formats it with a long date format.

```
Dim strToday As String = Today().ToString("D")
```

Date/time format strings are case sensitive.

Table 3-18 Common date/time formats

Format String	Description
d	Short date format, which shows the month, day, and year. An example is "8/10/2009".
D	Long date format, which contains the day of the week, month, day, and year. An example is "Monday, August 10, 2009".
t	Short time format, which shows the hours and minutes. An example is "3:22 PM".
T	Long time format, which contains the hours, minutes, seconds, and an AM/PM indicator. An example is "3:22:00 PM".
F	Full (long) date and time. An example is "Monday August 10, 2009 3:22:00 PM".

Tutorial 3-8 examines the *Format Demo* application.



Tutorial 3-8:

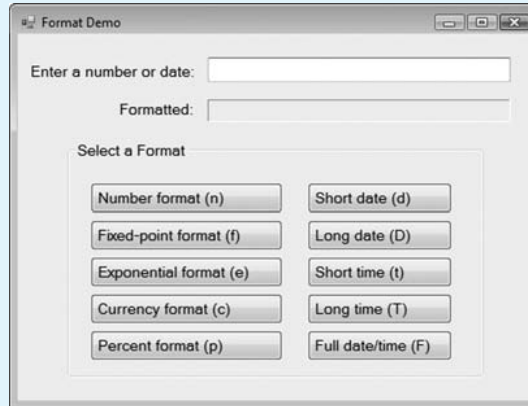
Examining the *Format Demo* application

Step 1: Open the *Format Demo* project from the Chapter 3 sample programs folder named *Format Demo*.

Step 2: Run the application, as shown in Figure 3-30. The five buttons on the left are used for formatting floating-point numeric values. The five buttons on the right

are for formatting dates and times. The text in each button shows which format string is used when the `ToString` method is called.

Figure 3-30 *Format Demo* application




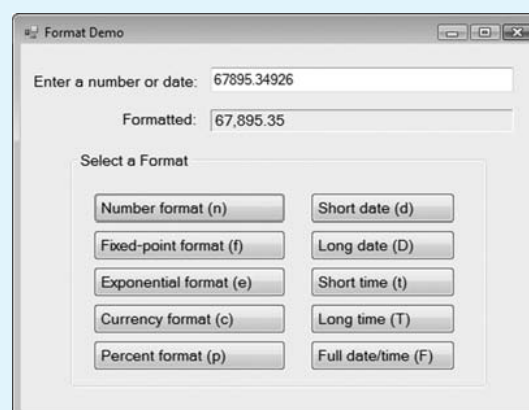
- Step 3:** Enter the value 67895.34926 into the TextBox control at the top of the form. Click the *Number format (n)* button. You should see the output shown in Figure 3-31.
- Step 4:** Click the *Fixed-point format (f)* button. Notice how the number next to the Formatted label changes its appearance.
- Step 5:** Click the remaining buttons on the left side. Change the value of the number in the TextBox control, and experiment with each of the format buttons on the left side.
- Step 6:** Enter the following date into the TextBox: *May 5, 2009 6:35 PM*.
- Step 7:** Click the *Short date (d)* button. You should see the date displayed as “5/5/2009”.
- Step 8:** Click the other date and time buttons in the right-hand column. Notice all the ways the date and time can be displayed.
- Step 9:** Close the application window by clicking the  in the upper right corner. Open the form’s code window and examine the code in each button’s Click event handler.

Figure 3-31 Showing a Number format





Checkpoint

- 3.36 Write a statement that uses the `ToString` method to convert the contents of a variable named `dblSalary` to a Currency format.
- 3.37 For each of the following numeric formats, identify the format string used as the input parameter when calling the `ToString` method.
- Currency
 - Exponential scientific
 - Number
 - Percent
 - Fixed-point
- 3.38 How can you make the `ToString` method display parentheses around a number in Currency format when the number is negative?
- 3.39 In the following table, fill in the expected values returned by the `ToString` function when specific numeric values are used with specific format strings.

Number Value	Format String	<code>ToString()</code> Value
12.3	n4	
12.348	n1	
1234567.1	n3	
123456.0	f1	
123456.0	e3	
.234	p2	
-1234567.8	c3	

- 3.40 Show an example of formatting a `Date` variable in Long Time format when calling the `ToString` method.
- 3.41 Show an example of formatting a `Date` variable in Long Date format when calling the `ToString` method.

3.6 Exception Handling

CONCEPT: A well-engineered program should report errors and try to continue. Or, it should explain why it cannot continue, and then shut down. In this section, you learn how to recover gracefully from errors, using a technique known as exception handling.

We have seen that runtime errors can happen when an attempt to convert a string to a number fails. Often, an uninformed user will enter a value in a `TextBox` that doesn't conform to Visual Basic's rules for converting strings to numbers. Suppose the following code in a button's `Click` event handler assigns the contents of a `TextBox` to a variable:

```
Dim decSalary As Decimal
decSalary = CDec(txtSalary.Text)
```

Then, when we run the program, the user enters a nonnumeric value, as shown in Figure 3-32, resulting in the error message shown on the right. If the user had entered \$4000,

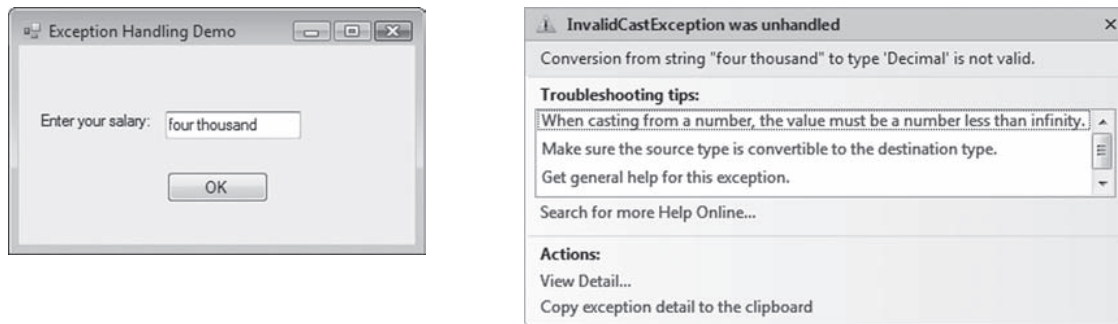
or \$4,000, the `CDec` function would have converted the input into `Decimal`. But this user had a different idea about how to enter numbers.

A common way of describing a runtime error is to say that an *exception was thrown*, and the exception was *not handled*. Or, one can refer to it as an **unhandled exception**.

Failure Can Be Graceful

Exceptions can be thrown because of events outside the programmer's control. A disk file may be unreadable, for example, because of a hardware failure. The user may enter invalid data. The computer may be low on memory. Exception handling is designed to let programs recover from errors when possible. Or, if recovery is not possible, a program should fail gracefully, letting the user know why it failed. Under no circumstances should it just halt without warning. In this section we will show how you can handle exceptions.

Figure 3-32 User input causes a runtime error



MessageBox.Show Function

Before we show how exceptions are handled in Visual Basic, let's look at an easy way to notify the user when an error has occurred. The `MessageBox.Show` function displays a pop-up window containing a message and an `OK` button, as shown in Figure 3-33. The user clicks the button to close the window. Here are two basic formats for `MessageBox.Show`:

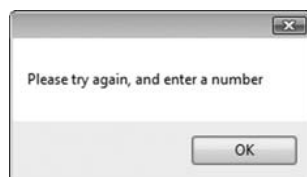
```
MessageBox.Show( message )
MessageBox.Show( message, caption )
```

The following statement displays the message in Figure 3-33:

```
MessageBox.Show("Please try again, and enter a number")
```

We will return to the `MessageBox.Show` function in Chapter 4 and explain its capabilities in more detail.

Figure 3-33 Window displayed by the `MessageBox.Show` function



Handling Exceptions

Visual Basic, along with most modern programming languages, provides a simple mechanism for handling exceptions. It's called an **exception handler**, and uses a **Try-Catch block**. It begins with the `Try` keyword and statements, followed by a `Catch` clause and statements, and concludes with the keywords `End Try`. This is a simplified format of a Try-Catch block, leaving out some options:

```
Try
    try-statements
Catch [exception-type]
    catch-statements
End Try
```

The *try-statements* consists of a list of program statements you would like to execute. They represent the code that would have existed even if you were not using exception handling. The *catch-statements* are one or more statements you would like to execute only if an exception is thrown. The optional *exception-type* argument lets you name the type of exception you want to catch.

Let's use the salary input example we saw earlier. We enclose statements in the `Try` block that input the salary and display a confirmation message by calling the `MessageBox.Show` function. The `Catch` block displays a message telling the user to try again. Lines are numbered for reference:

```
1: Try
2:   Dim decSalary As Decimal
3:   decSalary = CDec(txtSalary.Text)
4:   MessageBox.Show("Your salary is " & decSalary.ToString() & " dollars")
5: Catch
6:   MessageBox.Show("Please try again, and enter a number")
7: End Try
```

If we run the program as before and enter a valid salary, the program confirms the amount, as shown in Figure 3-34. If the user enters an invalid value, line 3 throws an exception, causing the program to jump to line 6.

Figure 3-34 Valid user input, with confirmation message



The `MessageBox.Show` function displays a helpful message, as shown in Figure 3-35. Notice that line 4 never executes when an exception is thrown. Most exception blocks have the same pattern, where some statements in a `Try` block are skipped when an exception occurs.



NOTE: In later chapters, you will see that exception handling is more often used to handle errors that cannot be anticipated by programmers. Examples are database connection errors, file input/output errors, and so on. To keep things simple in this chapter, we use exceptions to handle data conversion errors. In Chapter 4, you will learn to prevent errors by using `If` statements.

Figure 3-35 Invalid user input, causing the exception to be handled

Displaying Exception Object Information

When you catch an exception, you can display the properties of the `Exception` object it returns. For example, in the `Catch` block, identified by the `Catch` keyword, we can assign an arbitrary name (`ex`) to the exception object:

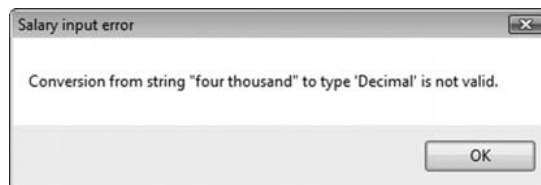
```
Catch ex As Exception
```

On the next line, we can display the `Message` property of the `Exception` in a message box:

```
MessageBox.Show(ex.Message)
```

The user sees a standardized message generated by Visual Basic's runtime system (see Figure 3-36). Here's a more complete example that adds a title to the message box:

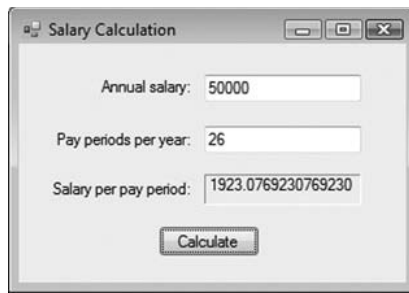
```
Try
    Dim decSalary As Decimal
    decSalary = CDec(txtSalary.Text)
Catch ex As Exception
    MessageBox.Show(ex.Message, "Salary input error")
End Try
```

Figure 3-36 Displaying an exception's `Message` property

If You Want to Know More: Handling Multiple Errors

A sequence of statements in a `Try` block might throw more than one type of exception. Suppose a program inputs someone's yearly salary, the number of pay periods per year, and then divides to get the salary amount per pay period, as shown in Figure 3-37. Someone earning \$50,000 with 26 pay periods per year would earn $(50000 / 26) = 1923.08$ per pay period. Here's the code that would do it, located in a `Try` block. Some of the lines are numbered for reference:

```
Try
1:   Dim decAnnualSalary As Decimal
2:   Dim intPayPeriods As Integer
3:   Dim decSalary As Decimal
4:   decAnnualSalary = CDec(txtAnnualSalary.Text)
5:   intPayPeriods = CInt(txtPayPeriods.Text)
6:   decSalary = decAnnualSalary / intPayPeriods
7:   lblSalary.Text = decSalary.ToString()
Catch
8:   MessageBox.Show("Please try again, and enter a number")
End Try
```

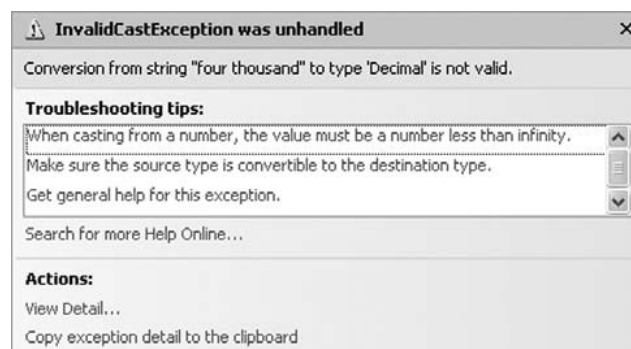
Figure 3-37 The *Salary Calculation* program

Suppose line 4 threw an exception because the user entered an illegal *annual salary*. The program would jump immediately to line 8 in the **Catch** block. Or, suppose the user entered a valid annual salary, but an invalid number for *the pay periods per year*. Then line 5 would throw an exception, and the program would immediately jump to line 8. In both cases, we can see that any remaining statements in the **Try** block are skipped when an exception is thrown.

What if the user entered a value of zero for the pay periods per year? Then a divide by zero exception would be thrown on line 6, which doesn't match the error message we show the user in line 8. That's when we could use two **Catch** blocks, one for bad number formats, and another for dividing by zero (shown in bold):

```
Try
    Dim decAnnualSalary As Decimal
    Dim intPayPeriods As Integer
    Dim decSalary As Decimal
    decAnnualSalary = CDec(txtAnnualSalary.Text)
    intPayPeriods = CInt(txtPayPeriods.Text)
    decSalary = decAnnualSalary / intPayPeriods
    lblSalary.Text = decSalary.ToString()
Catch ex As InvalidCastException
    MessageBox.Show("Please try again, and enter a number")
Catch ex As DivideByZeroException
    MessageBox.Show("Pay periods per year cannot be zero")
End Try
```

Each **Catch** block mentions a specific exception type, so the error message can focus on what the user did wrong. This is a more sophisticated way of handling exceptions, but it requires you to find out the exception type ahead of time. A simple way to achieve this is to run the program without a **Try-Catch** block and copy the name of the exception from the title bar of the runtime error dialog box, as shown in Figure 3-38. Tutorial 3-9 walks you through a salary calculation program with exception handling.

Figure 3-38 *InvalidCastException* example



Tutorial 3-9:

Salary Calculation application with exception handling

In this tutorial, you will create an application that asks the user to input a person's annual salary and number of pay periods per year. The program calculates the amount of salary they should receive per pay period. First, you will implement the program without exception handling, test it, and note how runtime errors occur. Then, you will add exception handling to the program and test it again. Figure 3-39 shows the form layout.

Figure 3-39 The *Salary Calculation* form

- Step 1:** Create a new project named *Salary Calculation*.
- Step 2:** Set the form's Text property to *Salary Calculation*.
- Step 3:** Add two TextBox controls to the form named *txtAnnualSalary* and *txtPayPeriods*.
- Step 4:** Add a Label control to the form just below the two text boxes named *lblSalary*. Set its BorderStyle property to *Fixed3D*.
- Step 5:** Add a Button control to the form named *btnCalculate* and assign *Calculate* to its Text property.
- Step 6:** Add appropriate labels next to the text boxes so your program's form looks like the form shown in Figure 3-39.
- Step 7:** Double-click the *Calculate* button to open the *Code* window. Insert the following code shown in bold into its Click event handler:

```

1: Private Sub btnCalculate_Click(ByVal sender As System.Object, _
2:   ByVal e As System.EventArgs) Handles btnCalculate.Click
3:
4:   Dim decAnnualSalary As Decimal ' annual salary
5:   Dim intPayPeriods As Integer ' number of pay periods per year
6:   Dim decSalary As Decimal ' salary per pay period
7:
8:   decAnnualSalary = CDec(txtAnnualSalary.Text)
9:   intPayPeriods = CInt(txtPayPeriods.Text)
10:  decSalary = decAnnualSalary / intPayPeriods
11:  lblSalary.Text = decSalary.ToString("c")

```

Lines 4 through 6 define the variables used in this procedure. Lines 8 and 9 copy the salary and number of pay periods from text boxes into variables. Line 10 calculates the amount of salary per pay period and line 11 assigns the result to a Label control named *lblSalary*.

Step 8: Save and run the program. Enter **75000** for the annual salary, and **26** for the pay periods per year. When you click *Calculate*, the output should be \$2,884.62.

Step 9: Erase the contents of the pay periods text box and click the *Calculate* button. You should see a runtime error dialog box appear, saying *InvalidCastException* was unhandled. After reading the dialog box, click the *Stop Debugging* button on the Visual Studio toolbar. The program should return to Design mode.

Your next task will be to add exception handling to the program to prevent the type of runtime error you just saw. It is never desirable for a program to halt like this when the user enters bad data.

Step 10: Revise the code in the `btnCalculate_Click` handler method so it looks like the following:

```
Dim decAnnualSalary As Decimal ' annual salary
Dim intPayPeriods As Integer   ' number of pay periods per year
Dim decSalary As Decimal       ' salary per pay period

Try
    decAnnualSalary = CDec(txtAnnualSalary.Text)
    intPayPeriods = CInt(txtPayPeriods.Text)
Catch
    MessageBox.Show("The input fields must contain " _
        & "nonzero numeric values.", "Error")
End Try

decSalary = decAnnualSalary / intPayPeriods
lblSalary.Text = decSalary.ToString("c")
```

Step 11: Save and run the program again. Enter an annual salary of **75000**, and leave the pay periods text box blank. You should see a message box that says *The input fields must contain nonzero numeric values*. After you close the message box, however, a runtime error still appears, with the caption *DivideByZeroException was unhandled*. Click the *Stop Debugging* button on the Visual Studio toolbar to return to Design mode.

Do you see why a runtime error still happened? After the catch clause in the exception handler executed, the program still attempted to divide `decAnnualSalary` by `intPayPeriods`. Because the latter equaled zero, a divide by zero error occurred. The next step will fix this problem.

Step 12: Open the *Code* window and revise the `btnCalculateClick` handler method once again. The way the code is written now, the division calculation only occurs if the pay periods value is converted to an integer without errors:

```
Dim decAnnualSalary As Decimal ' annual salary
Dim intPayPeriods As Integer   ' number of pay periods per year
Dim decSalary As Decimal       ' salary per pay period

Try
    decAnnualSalary = CDec(txtAnnualSalary.Text)
    intPayPeriods = CInt(txtPayPeriods.Text)
    decSalary = decAnnualSalary / intPayPeriods
    lblSalary.Text = decSalary.ToString("c")
Catch
    MessageBox.Show("The input fields must contain " _
        & "nonzero numeric values.", "Error")
End Try
```


Step 13: Save and run the program. Test it with valid data, as before. Test it by leaving the pay periods text box blank. The message box should alert the user, but the program should not halt.

Step 14: Click the *Stop Debugging* button to end the program, or click the *Close* icon in the upper right corner of the program's window.

3.7

Group Boxes and the Load Event Procedure

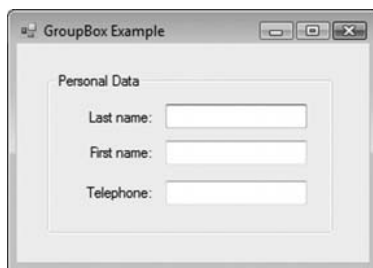
CONCEPT: In this section we discuss the `GroupBox` control, which is used to group other controls. We will also discuss the Load event procedure, which is executed when a form loads into memory.

Group Boxes

A group box is a rectangular border with an optional title that appears in the border's upper left corner. Other controls may be placed inside a group box. You can give forms a more organized look by grouping related controls together inside group boxes.

In Visual Basic, you use the **GroupBox** control to create a group box with an optional title. The title is stored in the `GroupBox` control's `Text` property. Figure 3-40 shows a `GroupBox` control. The control's `Text` property is set to *Personal Data*, and a group of other controls are inside the group box.

Figure 3-40 GroupBox containing other controls



Creating a Group Box and Adding Controls to It

To create a group box, select the `GroupBox` control from the *Containers* section of the *Toolbox* window and then draw the group box at the desired size on the form. To add another control to the group box, select the `GroupBox` control that you placed on the form, and then double-click the desired tool in the *Toolbox* to place another control inside the group box.

The controls you place inside a group box become part of a group. When you move a group box, the objects inside it move as well. When you delete a group box, the objects inside it are also deleted.

Moving an Existing Control to a Group Box

If an existing control is not inside a group box, but you want to move it to the group box, follow these steps:

1. Select the control you wish to add to the group box.
2. Cut the control to the clipboard.
3. Select the group box.
4. Paste the control.

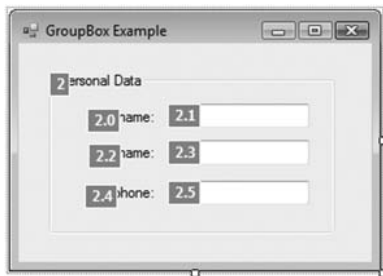
Group Box Tab Order

The value of a control's `TabIndex` property is handled differently when the control is placed inside a `GroupBox` control. `GroupBox` controls have their own `TabIndex` property and the `TabIndex` value of the controls inside the group box are relative to the `GroupBox` control's `TabIndex` property. For example, Figure 3-41 shows a `GroupBox` control displayed in tab order selection mode. As you can see, the `GroupBox` control's `TabIndex` is set to 2. The `TabIndex` of the controls inside the group box are displayed as 2.0, 2.1, 2.2, and so on.



NOTE: The `TabIndex` properties of the controls inside the group box will not appear this way in the *Properties* window. They will appear as 0, 1, 2, and so on.

Figure 3-41 Group box `TabIndex` values



Assigning an Access Key to a GroupBox Control

Although `GroupBox` controls cannot receive the focus, you can assign a keyboard access key to them by preceding a character in their `Text` property with an ampersand (&). When the user enters the access key, the focus moves to the control with the lowest `TabIndex` value inside the group box.

Selecting and Moving Multiple Controls

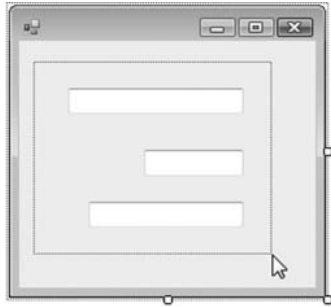
It is possible to select multiple controls and work with them all at once. For example, you can select a group of controls and move them all to a different location on the form. You can also select a group of controls and change some of their properties.

Select multiple controls by using one of the following techniques:

- Position the cursor over an empty part of the form near the controls you wish to select. Click and drag a selection box around the controls. This is shown in Figure 3-42. When you release the mouse button, all the controls that are partially or completely enclosed in the selection box will be selected.
- Hold down the **[Ctrl]** key while clicking each control you wish to select.

After using either of these techniques, all the controls you have selected will appear with sizing handles. You may now move them, delete them, or use the *Properties* window to set many of their properties to the same value.

Figure 3-42 Selecting multiple controls by clicking and dragging the mouse



TIP: In a group of selected controls, it is easy to deselect a control that you have accidentally selected. Simply hold down the **Ctrl** key and click the control you wish to deselect.

The Load Event Procedure

When a form loads into memory, a Load event takes place. If you need to execute code automatically when a form is displayed, you can place the code in the form's **Load event procedure**. For example, in the next section you will develop an application that displays the current date and time on the application's form. You will accomplish this by writing code in the form's Load event procedure that retrieves the date and time from the system.

To write code in a form's Load event procedure, double-click any area of the form where there is no other control. The *Code* window will appear with an event procedure similar to the following:

```
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
  
End Sub
```

Inside the procedure, simply write the statements you wish the procedure to execute. Be sure to leave a blank line before your first line of code.



Checkpoint

- 3.42 How is a group box helpful when designing a form with a large number of controls?
- 3.43 When placing a new control inside an existing *GroupBox* control, what must you do before you double-click the new control in the *ToolBox*?
- 3.44 How is the clipboard useful when you want to move an existing control into a group box?

- 3.45 How does the tab order of controls inside a group box correspond to the tab order of other controls outside the group box?
- 3.46 Which event procedure executes when the program's startup form displays?

3.8

Focus on Program Design and Problem Solving: Building the *Room Charge Calculator* Application

A guest staying at the Highlander Hotel may incur the following types of charges:

- Room charges, based on a per-night rate
- Room service charges
- Telephone charges
- Miscellaneous charges

The manager of the Highlander Hotel has asked you to create an application that calculates the guest's total charges. Figure 3-43 shows the application's form after the user has entered values into the text boxes and clicked the *Calculate Charges* button.

Figure 3-43 Sample output from the *Room Charge Calculator* application

Room Charge Calculator

Highlander Hotel

Today's Date: **Tuesday, November 27.**

Time: **11:53:08 PM**

Room Information

Nights: 5

Nightly Charge: 80.5

Additional Charges

Room Service: 10.50

Telephone: 5.25

Misc: 3.00

Total Charges

Room Charges:

Additional Charges:

Subtotal:

Tax:

Total Charges:

Calculate Charges Clear Exit

Figure 3-44 shows how the controls are arranged on the form with names. You can use this diagram when designing the form and assigning values to the Name property of each control. Notice that all Label controls appearing to the left of TextBox controls have their TextAlign property set to *MiddleRight*. This causes the labels to be right-aligned. The same is true of the labels on the lower half of the form, including *Room Charges*, *Additional Charges*, and so on. All those labels have their TextAlign property set to *MiddleRight*. Table 3-19, located at the end of this section, can be used as a reference.

Figure 3-44 Named controls

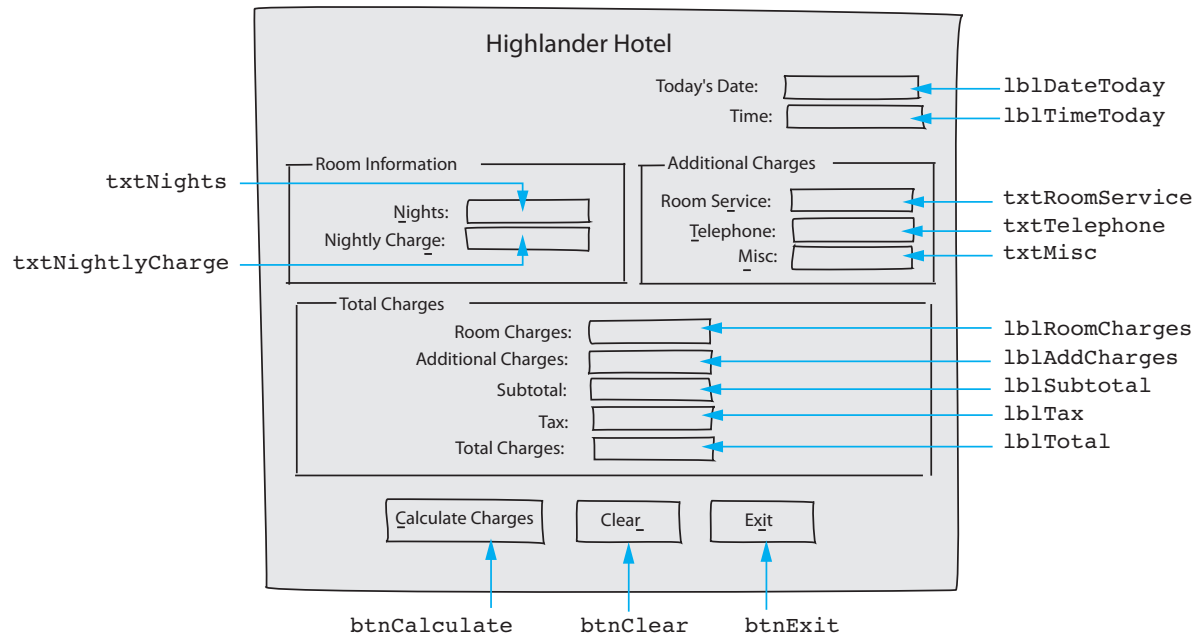


Table 3-19 lists and describes the methods (event procedures) needed for this application. Notice that a Load event procedure is needed for the form.

Table 3-19 Methods in the *Room Charge Calculator* application

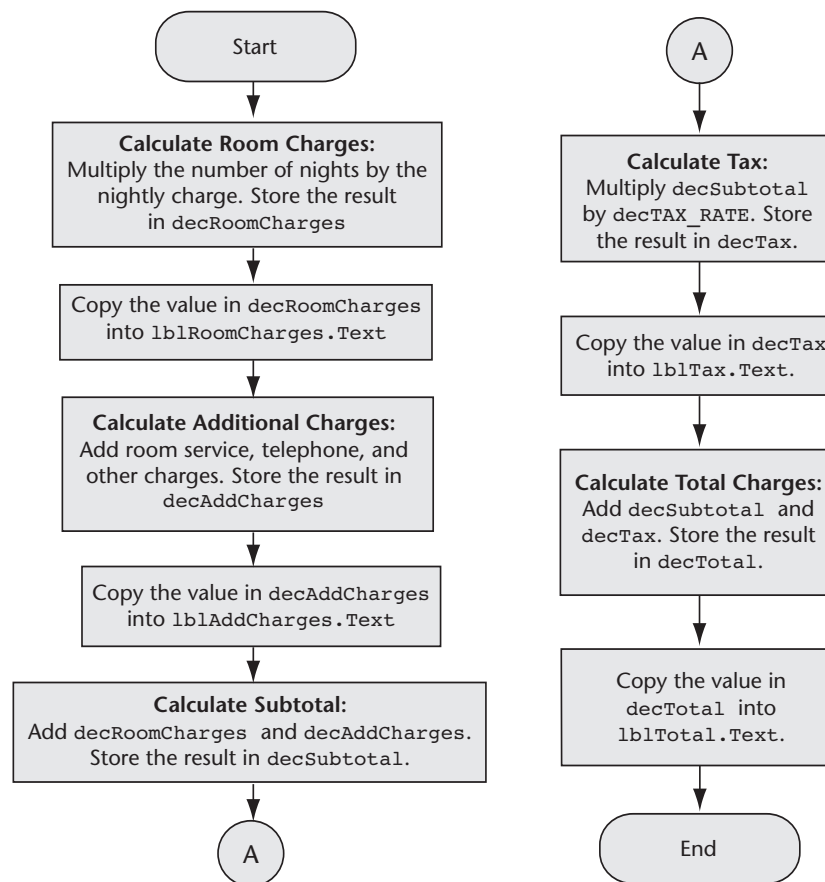
Method	Description
<code>btnCalculate_Click</code>	Calculates the room charges, additional charges, subtotal (room charges plus additional charges), 8% tax, and the total charges. These values are copied to the Text properties of the appropriate labels.
<code>btnClear_Click</code>	Clears the TextBox controls, and the labels used to display summary charge information. This procedure also resets the values displayed in the <code>lblDateToday</code> and <code>lblTimeToday</code> labels.
<code>btnExit_Click</code>	Ends the application
<code>Form1_Load</code>	Initializes the <code>lblDateToday</code> and <code>lblTimeToday</code> labels to the current system date and time


Figure 3-45 shows the flowchart for the `btnCalculate_Click` procedure. The procedure uses the following Decimal variables:


```
decRoomCharges
decAddCharges
decSubtotal
decTax
decTotal
```

The procedure also uses a named constant, `decTAX_RATE`, to hold the tax rate.

Figure 3-45 Flowchart for `btnCalculate_Click`

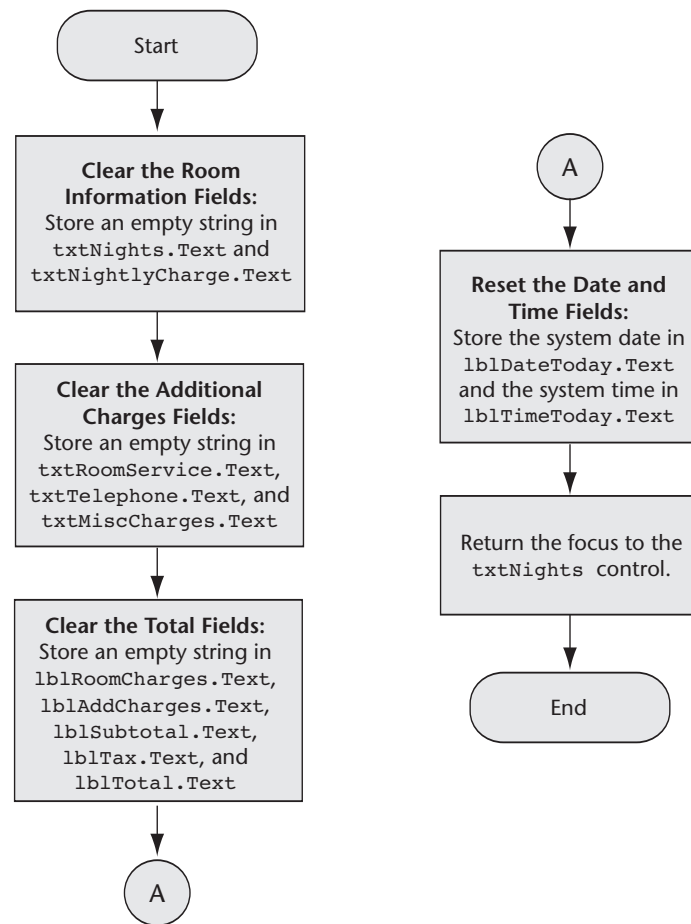
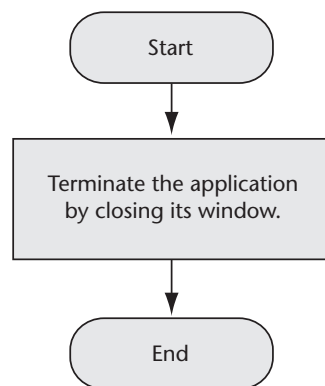
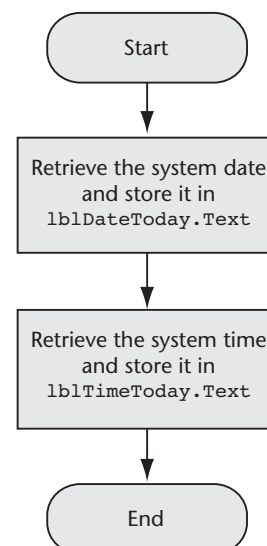


The flowchart in Figure 3-45 uses a new symbol: 

This is called the **connector symbol** and is used when a flowchart is broken into two or more smaller flowcharts. This is necessary when a flowchart does not fit on a single page or must be divided into sections. A connector symbol, which is a small circle with a letter or number inside it, allows you to connect two flowcharts. In the flowchart shown in Figure 3-45, the  connector indicates that the second flowchart segment begins where the first flowchart segment ends.

The flowcharts for the `btnClear_Click`, `btnExit_Click`, and `Form1_Load` procedures are shown in Figures 3-46, 3-47, and 3-48, respectively.

Recall that the form's Load procedure executes each time the form loads into memory. Tutorial 3-10 shows you how to create the *Room Charge Calculator* application.

Figure 3-46 Flowchart for `btnClear_Click`**Figure 3-47** Flowchart for `btnExit_Click`**Figure 3-48** Flowchart for `Form1_Load` procedure



Tutorial 3-10:

Beginning the *Room Charge Calculator* application

Step 1: Create a new Windows application project named *Room Charge Calculator*.

Step 2: Figure 3-44 shows a sketch of the application's form and shows the names of the named controls. Refer to this figure as you set up the form and create the controls. Once you have completed the form, it should appear as shown in Figure 3-49.

Figure 3-49 The *Room Charge Calculator* form



TIP: Most of the controls are contained inside group boxes. Refer to Section 3.7 for instructions on creating controls inside a group box.



TIP: The TextBox controls are all the same size, and all the Label controls that display output are the same size. If you are creating many similar instances of a control, it is easier to create the first one, set its size and properties as needed, copy it to the clipboard, and then paste it onto the form to create another one.

Step 3: Table 3-20 (on page 164) lists the relevant property settings of all the controls on the form. Refer to this table and make the necessary property settings.

Step 4: Now you will write the application's event procedures, beginning with the form's Load procedure. Double-click any area of the form not occupied by another control. The Code window should open with a code template for the

Form1_Load procedure. Complete the procedure by typing the following code shown in bold:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    ' Get today's date from the system and display it.
    lblDateToday.Text = Now.ToString("D")
    ' Get the current time from the system and display it.
    lblTimeToday.Text = Now.ToString("T")
End Sub
```

Step 5: Double-click the *Calculate Charges* button. The *Code* window should open with a code template for the btnCalculate_Click procedure. Complete the procedure by typing the following code shown in bold:

```
Private Sub btnCalculate_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles btnCalculate.Click

    ' Declare variables for the calculations.
    Dim decRoomCharges As Decimal ' Room charges total
    Dim decAddCharges As Decimal ' Additional charges
    Dim decSubtotal As Decimal ' Subtotal
    Dim decTax As Decimal ' Tax
    Dim decTotal As Decimal ' Total of all charges
    Const decTAX_RATE As Decimal = 0.08D ' Tax rate

    Try
        ' Calculate and display the room charges. Handle
        ' error if the fields are blank.
        decRoomCharges = CDec(txtNights.Text) * _
            CDec(txtNightlyCharge.Text)
        lblRoomCharges.Text = decRoomCharges.ToString("c")
    Catch
        MessageBox.Show("Nights and Nightly Charge must be numbers", _
            "Error")
    End Try

    Try
        ' Calculate and display the additional charges. Handle
        ' error if fields are blank.
        decAddCharges = CDec(txtRoomService.Text) + _
            CDec(txtTelephone.Text) + _
            CDec(txtMisc.Text)
        lblAddCharges.Text = decAddCharges.ToString("c")
    Catch
        MessageBox.Show("Room service, Telephone, and Misc. " _
            & "must be numbers", "Error")
    End Try

    ' Calculate and display the subtotal.
    decSubtotal = decRoomCharges + decAddCharges
    lblSubtotal.Text = decSubtotal.ToString("c")

    ' Calculate and display the tax.
    decTax = decSubtotal * decTAX_RATE
    lblTax.Text = decTax.ToString("c")
```

```

' Calculate and display the total charges.
decTotal = decSubtotal + decTax
lblTotal.Text = decTotal.ToString("c")
End Sub

```

Exception handling was used in this procedure to check for missing user input. If the user forgets to enter the number of nights and the nightly charge, for example, the first `Catch` block displays a message box, as shown in Figure 3-50. Imagine how much better this is than letting the program halt unexpectedly. If the user forgets to enter amounts for room service, telephone, and miscellaneous charges, a second `Catch` block displays the message box shown in Figure 3-51.

Figure 3-50 Message box displayed by first exception handler

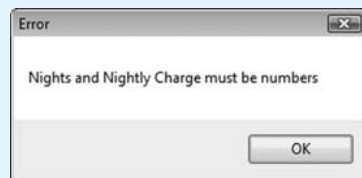
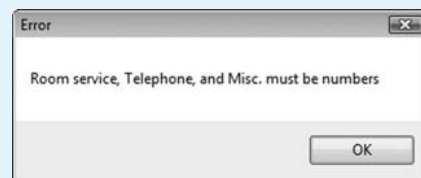


Figure 3-51 Message box displayed by second exception handler



TIP: When you type the name of a built-in Visual Basic function or a method in the *Code* window, an IntelliSense box appears showing help on the function or method's arguments. If you do not want to see the IntelliSense box, press the `[Esc]` key when it appears.

Step 6: Open the *Design* window and double-click the *Clear* button. The *Code* window should open with a code template for the `btnClear_Click` procedure. Complete the procedure by typing the following code shown in bold.

```

Private Sub btnClear_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnClear.Click

    ' Clear the room info fields.
    txtNights.Clear()
    txtNightlyCharge.Clear()
    ' Clear the additional charges fields.
    txtRoomService.Clear()
    txtTelephone.Clear()
    txtMisc.Clear()

    ' Clear the total fields.
    lblRoomCharges.Text = String.Empty
    lblAddCharges.Text = String.Empty
    lblSubtotal.Text = String.Empty
    lblTax.Text = String.Empty
    lblTotal.Text = String.Empty

    ' Get today's date from the operating system and display it.
    lblDateToday.Text = Now.ToString("D")

```

```
' Get the current time from the operating system and display it.
lblTimeToday.Text = Now.ToString("T")

' Reset the focus to the first field.
txtNights.Focus()
End Sub
```

Step 7: Open the *Design* window and double-click the *Exit* button. The *Code* window should open with a code template for the `btnExit_Click` procedure. Complete the procedure by typing the following code shown in bold.

```
Private Sub btnExit_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnExit.Click

    ' End the application, by closing the window.
    Me.Close()
End Sub
```

Step 8: Use the *Save All* command on the *File* menu (or the *Save All* button) to save the project.

Step 9: Run the application. If there are errors, compare your code with that shown, and correct them. Once the application runs, enter test values, as shown in Figure 3-52 for the charges and confirm that it displays the correct output.

Figure 3-52 Sample output from the *Room Charge Calculator* application



Table 3-20 lists each control, along with any relevant property values.

Table 3-20 Named controls in *Room Charge Calculator* form

Control Type	Control Name	Property	Property Value
Label	(Default)	Text: Font: TextAlign:	<i>Highlander Hotel</i> <i>MS sans serif, bold, 18 point</i> <i>MiddleCenter</i>
Label	(Default)	Text: TextAlign:	<i>Today's Date:</i> <i>MiddleRight</i>
Label	lblDateToday	Text: AutoSize: Font.Bold: BorderStyle: TextAlign:	(Initially cleared) <i>False</i> <i>True</i> <i>None</i> <i>MiddleLeft</i>
Label	(Default)	Text: TextAlign:	<i>Time:</i> <i>MiddleRight</i>
Label	lblTimeToday	Text: AutoSize: Font.Bold: BorderStyle: TextAlign:	(Initially cleared) <i>False</i> <i>True</i> <i>None</i> <i>MiddleLeft</i>
Group box	(Default)	Text:	<i>Room Information</i>
Label	(Default)	Text: TextAlign:	<i>&Nights:</i> <i>MiddleRight</i>
TextBox	txtNights	Text:	(Initially cleared)
Label	(Default)	Text: TextAlign:	<i>Nightly Char&ge:</i> <i>MiddleRight</i>
TextBox	txtNightlyCharge	Text:	(Initially cleared)
Group box	(Default)	Text:	<i>Additional Charges</i>
Label	(Default)	Text: TextAlign:	<i>Room Se&rvice:</i> <i>MiddleRight</i>
TextBox	txtRoomService	Text:	(Initially cleared)
Label	(Default)	Text: TextAlign:	<i>&Telephone:</i> <i>MiddleRight</i>
TextBox	txtTelephone	Text:	(Initially cleared)
Label	(Default)	Text: TextAlign:	<i>&Misc:</i> <i>MiddleRight</i>
TextBox	txtMisc	Text:	(Initially cleared)
Group box	(Default)	Text:	<i>Total Charges</i>
Label	(Default)	Text: TextAlign:	<i>Room Charges:</i> <i>MiddleRight</i>
Label	lblRoomCharges	Text: AutoSize: BorderStyle:	(Initially cleared) <i>False</i> <i>Fixed3D</i>
Label	(Default)	Text: TextAlign:	<i>Additional Charges:</i> <i>MiddleRight</i>
Label	lblAddCharges	Text: AutoSize: BorderStyle:	(Initially cleared) <i>False</i> <i>Fixed3D</i>

(continues)

Table 3-20 Named controls in *Room Charge Calculator* form (continued)

Control Type	Control Name	Property	Property Value
Label	(Default)	Text:	<i>Subtotal:</i>
		TextAlign:	<i>MiddleRight</i>
Label	lblSubtotal	Text:	(Initially cleared)
		AutoSize:	<i>False</i>
		BorderStyle:	<i>Fixed3D</i>
Label	(Default)	Text:	<i>Tax:</i>
		TextAlign:	<i>MiddleRight</i>
Label	lblTax	Text:	(Initially cleared)
		AutoSize:	<i>False</i>
		BorderStyle:	<i>Fixed3D</i>
Label	(Default)	Text:	<i>Total Charges:</i>
		TextAlign:	<i>MiddleRight</i>
Label	lblTotal	Text:	(Initially cleared)
		AutoSize:	<i>False</i>
		BorderStyle:	<i>Fixed3D</i>
Button	btnCalculate	Text:	<i>C&alculate Charges</i>
		TabIndex:	2
Button	btnClear	Text:	<i>Clea&r</i>
		TabIndex:	3
Button	btnExit	Text:	<i>E&xit</i>
		TabIndex:	4

Changing Colors with Code (Optional Topic)

Chapter 2 showed how to change the foreground and background colors of a control's text by setting the `ForeColor` and `BackColor` properties in the *Design* view. In addition to using the *Properties* window, you can also store values in these properties with code. Visual Basic provides numerous values that represent colors, and can be assigned to the `ForeColor` and `BackColor` properties in code. The following are a few of the values:

```
Color.Black
Color.Blue
Color.Cyan
Color.Green
Color.Magenta
Color.Red
Color.White
Color.Yellow
```

For example, assume an application has a Label control named `lblMessage`. The following code sets the label's background color to black and foreground color to yellow:

```
lblMessage.BackColor = Color.Black
lblMessage.ForeColor = Color.Yellow
```

Visual Basic also provides values that represent default colors on your system. For example, the value `SystemColors.Control` represents the default control background color and `SystemColors.ControlText` represents the default control text color. The following statements set the `lblMessage` control's background and foreground to the default colors.

```
lblMessage.BackColor = SystemColors.Control
lblMessage.ForeColor = SystemColors.ControlText
```

In Tutorial 3-11, you will modify the *Room Charge Calculator* application so that the total charges are displayed in white characters on a blue background. This will make the total charges stand out visually from the rest of the information on the form.



Tutorial 3-11: Changing a label's colors

In this tutorial, you will modify two of the application's event procedures: `btnCalculate_Click` and `btnClear_Click`. In the `btnCalculate_Click` procedure, you will add code that changes the `lblTotal` control's color settings just before the total charges are displayed. In the `btnClear_Click` procedure, you will add code that reverts `lblTotal`'s colors back to their normal state.

Step 1: With the *Room Charge Calculator* project open, open the *Code* window and scroll to the `btnCalculate_Click` event procedure.

Step 2: Add the following lines to the end of the `btnCalculate_Click` procedure:

```
' Change the background and foreground colors
' for the total charges.
lblTotal.BackColor = Color.Blue
lblTotal.ForeColor = Color.White
```

Step 3: Add the following bold lines to the end of the `btnClear_Click` procedure, just before the line at the end that calls the `Focus` method. The existing line is shown to help you find the right location:

```
' Reset the lblTotal control's colors.
lblTotal.BackColor = SystemColors.Control
lblTotal.ForeColor = SystemColors.ControlText

' Reset the focus to the first field.
txtNights.Focus()
```

Step 4: Save the project.

Step 5: Run and test the application. When you click the *Calculate Charges* button, the value displayed in the `lblTotal` label should appear in white text on a blue background. When you click the *Clear* button, the color of the `lblTotal` label should return to normal.

3.9

More about Debugging: Locating Logic Errors

CONCEPT: Visual Studio allows you to pause a program, and then execute statements one at a time. After each statement executes, you may examine variable contents and property values.

A **logic error** is a mistake that does not prevent an application from running, but causes the application to produce incorrect results. Mathematical mistakes, copying a value to the wrong variable, or copying the wrong value to a variable are examples of logic errors. Logic errors can be difficult to find. Fortunately, Visual Studio provides you with debugging tools that make locating logic errors easier.

Visual Studio allows you to set breakpoints in your program code. A **breakpoint** is a line you select in your source code. When the application is running and it reaches a breakpoint, the application pauses and enters Break mode. While the application is paused, you may examine variable contents and the values stored in certain control properties.

Visual Studio allows you to **single-step** through an application's code once its execution has been paused by a breakpoint. This means that the application's statements execute one at a time, under your control. After each statement executes, you can examine variable and property values. This process allows you to identify the line or lines of code causing the error. In Tutorial 3-12, you single-step through an application's code.



Tutorial 3-12:

Single-stepping through an application's code at runtime

In this tutorial, you set a breakpoint in an application's code, run it in debugging mode, and single-step through the application's code to find a logic error.

Step 1: Open the *Average Race Times* project from the student sample programs folder named *Chap3\Average Race Times*.

Step 2: Run the application. The application's form appears, as shown in Figure 3-53.

Figure 3-53 *Average Race Times* form

Step 3: This application allows you to enter the finishing times of three runners in a race, and then see their average time. Enter **25** as the time for all three runners.

Step 4: Click the *Calculate Average* button. The application displays the incorrect value 58.3 as the average time. (The correct value should be 25.)

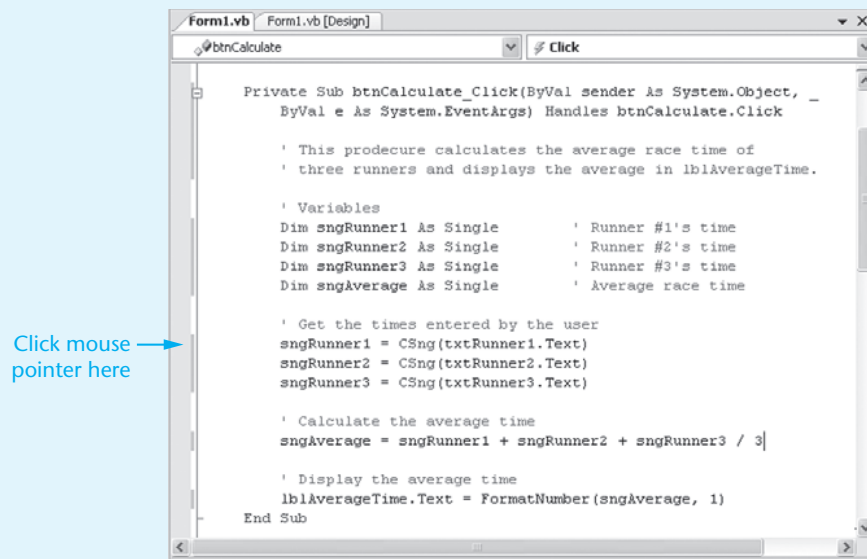
Step 5: Click the *Exit* button to stop the application.

Step 6: Open the *Code* window and locate the following line of code, which appears in the `btnCalculate_Click` event procedure:

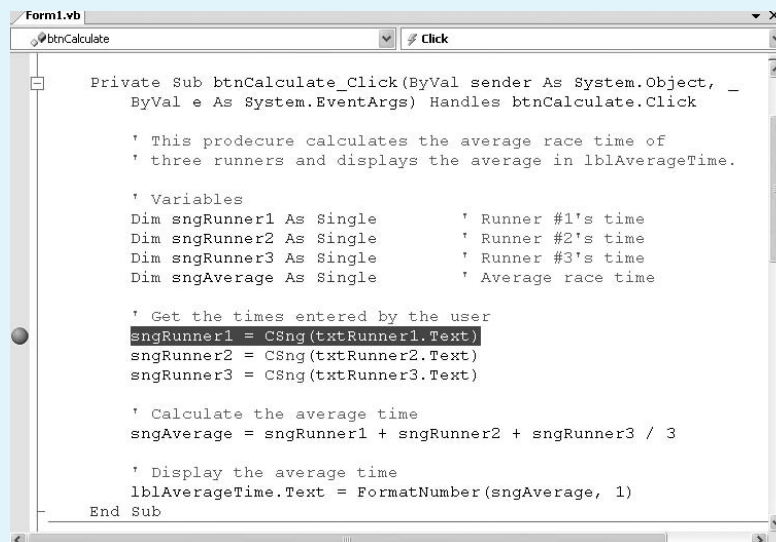
```
sngRunner1 = CSng(txtRunner1.Text)
```

This line of code is where we want to pause the execution of the application. We must make this line a breakpoint.

Step 7: Click the mouse in the left margin of the *Code* window, next to the line of code, as shown in Figure 3-54.

Figure 3-54 Click the mouse in the left margin of the *Code* window

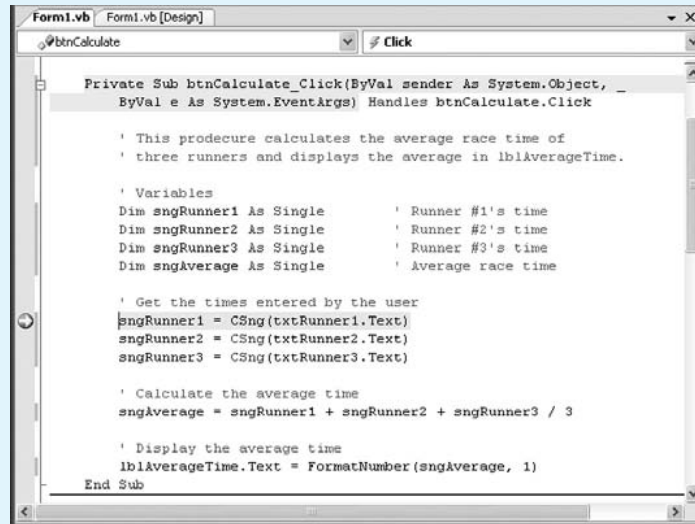
Step 8: Notice that a red dot appears next to the line in the left margin. This is shown in Figure 3-55.

Figure 3-55 Breakpoint code highlighted

The dot indicates that a breakpoint has been set on this line. Another way to set a breakpoint is to move the text cursor to the line you wish to set as a breakpoint, and then press **[F9]**.

Step 9: Now that you have set the breakpoint, run the application. When the form appears, enter **25** as the time for each runner.

Step 10: Click the *Calculate Average* button. When program execution reaches the breakpoint, it goes into Break mode and the *Code* window reappears. The breakpoint line is shown with yellow highlighting and a small yellow arrow appears in the left margin, as shown in Figure 3-56.

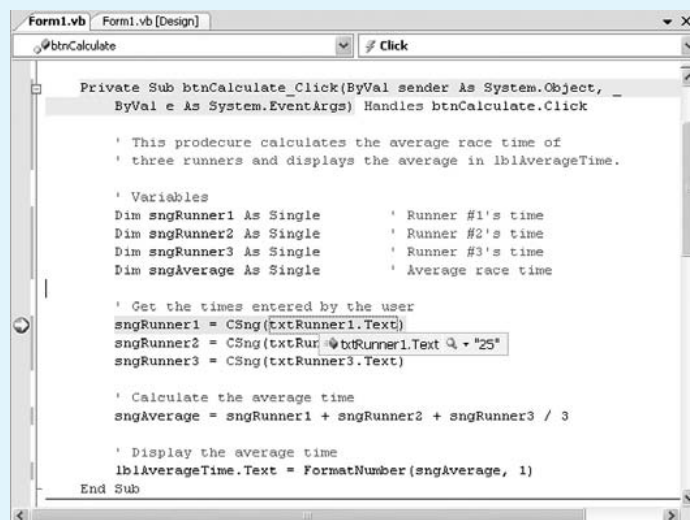
Figure 3-56 Breakpoint during Break mode

The yellow highlighting and small arrow indicate the application's current execution point. The **execution point** is the next line of code that will execute. (The line has not yet executed.)



NOTE: If the highlighting and arrow appear in a color other than yellow, the color options on your system may have been changed.

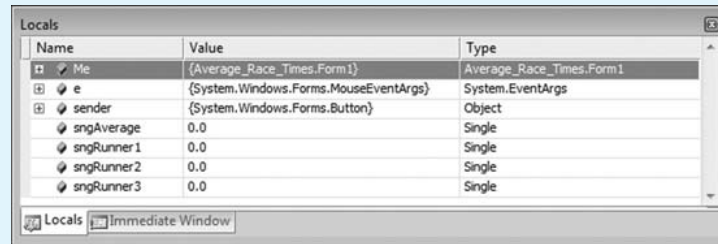
Step 11: To examine the contents of a variable or control property, hover the cursor over the variable or the property's name in the *Code* window. A small box will appear showing the variable or property's contents. For example, Figure 3-57 shows the result of hovering the mouse pointer over the expression `txtRunner1.Text` in the highlighted line. The box indicates that the property is currently set to 25.

Figure 3-57 `txtRunner1.Text` property contents revealed

Step 12: Now hover the mouse pointer over the variable name `sngRunner1`. A box appears indicating that the variable is set to 0.0. Because the highlighted statement has not yet executed, no value has been assigned to this variable.

Step 13: You may also examine the contents of variables with the *Autos*, *Locals*, and *Watch* windows. Figure 3-58 shows the *Locals* window, which normally appears near the bottom of your screen.

Figure 3-58 The *Locals* window

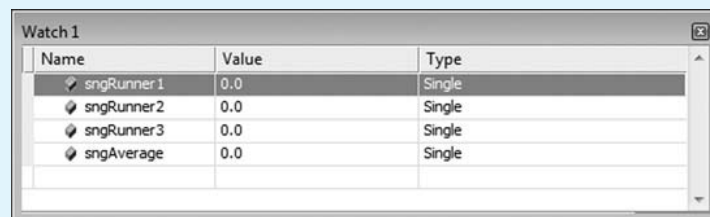


A description of each window follows:

- The *Autos* window (Visual Studio only) displays a list of the variables appearing in the current statement, the three statements before, and the three statements after the current statement. The current value and the data type of each variable are also displayed.
- The *Immediate* window allows you to type debugging commands using the keyboard. This window is generally used by advanced programmers.
- The *Locals* window displays a list of all the variables in the current procedure. The current value and the data type of each variable are also displayed.
- The *Watch* window allows you to add the names of variables you want to watch. This window displays only the variables you have added. Visual Studio lets you open multiple *Watch* windows, whereas Visual Basic Express offers only one *Watch* window.

Step 14: From the menu, select *Debug*, select *Windows*, select *Watch*, and select *Watch*. (If you're using Visual Studio, you must select one of several *Watch* windows.) A *Watch* window should appear, similar to the one shown in Figure 3-59.

Figure 3-59 *Watch 1* window displayed



Step 15: If you do not already see the variables `sngRunner1`, `sngRunner2`, `sngRunner3`, and `sngAverage` displayed in the *Watch* window, you can add them by performing the following:

- Click the first blank line in the window.
- Type `sngRunner1` and press `[Enter]`.
- Type `sngRunner2` and press `[Enter]`.
- Type `sngRunner3` and press `[Enter]`.
- Type `sngAverage` and press `[Enter]`.

You have added the variables `sngRunner1`, `sngRunner2`, `sngRunner3`, and `sngAverage` to the *Watch* window. The variables are all equal to zero.

Step 16: Now you are ready to single-step through each statement in the event procedure. To do this, use the *Step Into* command. (The *Step Over* command, which is similar to *Step Into*, is covered in Chapter 6.) You activate the *Step Into* command by one of the following methods:

- Press the `[F8]` key.
- Select *Debug* from the menu bar, and then select *Step Into* from the *Debug* menu.



NOTE: Visual Studio users: We assume your profile is set to Visual Basic Developer. If your profile is set to *Visual Studio Developer*, the *Step Into* command is executed by the `[F11]` key instead of the `[F8]` key.

When you activate the *Step Into* command, the highlighted statement is executed. Press the `[F8]` key now. Look at the *Watch* window and notice that the `sngRunner1` variable is now set to 25.0. Also notice that the next line of code is now highlighted.

Step 17: Press the `[F8]` key two more times. The variables `sngRunner1`, `sngRunner2`, and `sngRunner3` should display values of 25 in the *Watch* window.

Step 18: The following statement, which is supposed to calculate the average of the three scores, is now highlighted:

```
sngAverage = sngRunner1 + sngRunner2 + sngRunner3 / 3.0
```

After this statement executes, the average of the three numbers should display next to `sngAverage`. Press `[F8]` to execute the statement.

Step 19: Notice that the *Watch* window now reports that `sngAverage` holds the value 58.3333321. This is not the correct value, so there must be a problem with the math statement that just executed. Can you find it? The math statement does not calculate the correct value because the division operation takes place before any of the addition operations. You must correct the statement by inserting a set of parentheses.

From the menu, select *Debug*, and then click *Stop Debugging* to halt the application. In the *Code* window, insert a set of parentheses into the math statement so it appears as follows:

```
sngAverage = (sngRunner1 + sngRunner2 + sngRunner3) / 3
```

Step 20: Next, you will clear the breakpoint so the application will not pause again when it reaches that line of code. To clear the breakpoint, use one of the following methods:

- Click the mouse on the breakpoint dot in the left margin of the *Code* window.
- Press `[Ctrl]+[Shift]+[F9]`.
- Select *Debug* from the menu bar, and then select *Delete All Breakpoints* from the *Debug* menu.

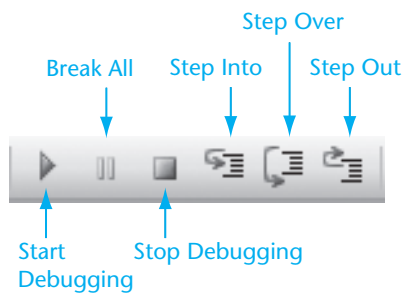
Step 21: Run the application again. Enter **25** as each runner's time, and then click the *Calculate Average* button. This time the correct average, 25.0, is displayed.

Step 22: Click the *Exit* button to stop the application.

If You Want to Know More: Debugging Commands in the Toolbar

Visual Studio provides a toolbar for debugging commands, shown in Figure 3-60.

Figure 3-60 *Debug toolbar commands*



Checkpoint

- 3.47 What is the difference between a syntax error and a logic error?
- 3.48 What is a breakpoint?
- 3.49 What is the purpose of single-stepping through an application?

Summary

3.1 Gathering Text Input

- Words and characters typed into a `TextBox` control is stored in the control's `Text` property. The standard prefix for `TextBox` control names is `txt`.
- The `&` operator is used to perform string concatenation.
- The control that has the focus receives the user's keyboard input or mouse clicks. The focus is moved by calling the `Focus` method.
- The order in which controls receive the focus when the `[Tab]` key is pressed at run-time is called the tab order. When you place controls on a form, the tab order will be the same sequence in which you created the controls. You can modify the tab order by changing a control's `TabIndex` property.
- Tab order selection mode allows you to easily view the `TabIndex` property values of all the controls on a form.
- If you do not want a control to receive the focus when the user presses the `[Tab]` key, set its `TabStop` property to *False*.
- You assign an access key to a button by placing an ampersand (`&`) in its `Text` property. The letter that immediately follows the ampersand becomes the access key. That letter appears underlined on the button.
- Forms have two properties named `AcceptButton` and `CancelButton`. `AcceptButton` refers to the control that will receive a `Click` event when the user presses the `[Enter]` key. `CancelButton` refers to the control that will receive a `Click` event when the user presses the `[Esc]` key.

3.2 Variables and Data Types

- The assignment operator (`=`) is used to store a value in a variable, just as with a control property. A variable's data type determines the type of information that the variable can hold.
- Rules for naming variables are enforced by the Visual Basic compiler. Naming conventions, on the other hand, are not rigid—they are based on a standard style of programming.
- When a variable is first created in memory, Visual Basic assigns it an initial value, which depends on its data type. You may also initialize a variable, which means that you specify the variable's starting value.
- Variables of the `Date` (`DateTime`) data type can hold a date and time. You may store values in a `Date` variable with date literals, strings, or user input.
- Each Visual Basic data type has a method named `ToString` that returns a string representation of the variable calling the method.

3.3 Performing Calculations

- A unary operator has only one operand. An example is the negation operator (minus sign).
- A binary operator has two operands. An example is the addition operator (plus sign).
- The `\` symbol identifies the integer division operator.

- The `*` symbol identifies the multiplication operator.
- The `^` symbol identifies the exponentiation operator.
- The `MOD` operator returns the remainder after performing integer division.
- When two operators share an operand, the operator with the highest precedence executes first. Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others.
- A combined assignment operator combines the assignment operator with another operator.
- The `Now` function retrieves the current date and time from the computer system. The `TimeOfDay` function retrieves the current time. The `Today` function retrieves the current date.
- A variable's scope determines where a variable is visible and where it can be accessed by programming statements.
- A variable declared inside a procedure is called a local variable. This type of variable is only visible from its declaring statement to the end of the same procedure. If a variable is declared inside a class, but outside of any procedure, it is called a class-level variable. If a variable is declared outside of any class or procedure, it is called a global variable.

3.4 Mixing Different Data Types

- Implicit type conversion occurs when you assign a value of one data type to a variable of another data type. Visual Basic attempts to convert the value being assigned to the data type of the destination variable.
- A narrowing conversion occurs when a real number is assigned to one of the integer type variables. It also occurs when a larger type is assigned to a smaller type.
- A widening conversion occurs when data of a smaller type is assigned to a variable of a larger type. An example is when assigning any type of integer to a `Double`.
- Visual Basic attempts to convert strings to numbers, particularly when the strings contain digits. Other permitted characters are a single `$`, a single decimal point, a leading sign, and commas.
- The *Option Strict* statement determines whether certain implicit conversions are legal. When *Option Strict* is *On*, only widening conversions are permitted. When *Option Strict* is *Off*, both narrowing and widening conversions are permitted.
- A type conversion or type mismatch error is generated when an automatic conversion is not possible.
- An explicit type conversion is performed by one of Visual Basic's conversion functions. The conversion functions discussed in this chapter are `CDate` (convert to date), `CDB1` (convert to `Double`), `CDec` (convert to `Decimal`), `CInt` (convert to `Integer`), and `CStr` (convert to `String`).
- The `CInt` function performs a special type of rounding called bankers rounding.
- The `Val` function converts a string argument to a number.
- Visual Basic provides several type conversion functions, such as `CInt` and `CDB1`, which convert expressions to other data types.
- Tutorial 3-7 showed a simple calculator program that uses the `CInt` and `CDB1` functions.

3.5 Formatting Numbers and Dates

- Ordinarily, numeric values should be formatted when they are displayed. Formatting gives your programs a more professional appearance.
- The `ToString` method converts the contents of a variable into a string.
- You can pass a format string as an input argument to the `ToString` method. The format string can be used to configure the way a number or date is displayed.
- Number format (`n` or `N`) displays numeric values with thousands separators and a decimal point.
- Fixed-point format (`f` or `F`) displays numeric values with no thousands separator and a decimal point.
- Exponential format (`e` or `E`) displays numeric values in scientific notation. The number is normalized with a single digit to the left of the decimal point.
- Currency format (`c` or `C`) displays a leading currency symbol (such as \$), digits, thousands separators, and a decimal point.
- Percent format (`p` or `P`) causes the number to be multiplied by 100 and displayed with a trailing space and % sign.
- You can use the `ToString` method to format dates and times. Several standard formats were shown in this chapter: short date, long date, short time, long time, and full date and time.

3.6 Exception Handling

- Exception handling is a structured mechanism for handling errors in Visual Basic programs.
- Exception handling begins with the `Try` keyword, followed by one or more `Catch` blocks, followed by `End Try`.
- Some types of errors are preventable by the programmer, such as dividing by zero. Other errors may be caused by user input, which is beyond the control of the programmer.
- When a program throws an exception, it generates a runtime error. An unhandled exception causes a program to terminate and display an error message.
- You can write exception handlers that catch exceptions and find ways for the program to recover. Your exception handler can also display a message to the user.
- Exception handlers can handle multiple exceptions by specifically identifying different types of exceptions with different catch blocks.

3.7 Group Boxes, Form Formatting, and the Load Event Procedure

- A `GroupBox` control, which is used as a container for other controls, appears as a rectangular border with an optional title. You can create the `GroupBox` first and then create other controls inside it. Alternatively, you can drag existing controls inside the `GroupBox`.
- In the *Design* window, grid lines can be used to align controls. You can select and work with multiple controls simultaneously.
- Every form has a `Load` event procedure, executed when the form loads into memory. If you need to execute code before a form is displayed, place it in the form's `Load` event handler.

3.8 Focus on Program Design and Problem Solving: Building the *Room Charge Calculator* Application

- The *Room Charge Calculator* application calculates charges for guests at an imaginary hotel. It combines many of the techniques introduced in this chapter, such as type conversion functions, formatting numbers, and formatting dates.
- Visual Basic provides numerous values that represent colors. These values may be used in code to change a control's foreground and background colors.

3.9 More about Debugging: Locating Logic Errors

- A logic error is a programming mistake that does not prevent an application from compiling, but causes the application to produce incorrect results.
- A runtime error occurs during a program's execution—it halts the program unexpectedly.
- A breakpoint is a line of code that causes a running application to pause execution and enter Break mode. While the application is paused, you may perform debugging operations such as examining variable contents and the values stored in control properties.
- Single-stepping is the debugging technique of executing an application's programming statements one at a time. After each statement executes, you can examine variable and property contents.

Key Terms

accept button
access key
Autos window
binary operator
breakpoint
cancel button
code outlining
combined assignment operators
compound operators
connector symbol
exception
execution point
exception handler
expression
focus
Focus method
function
GroupBox control
Immediate window
implicit type conversion
initialization
line-continuation character
Load event procedure
Locals window
logic error
mathematical expression

mnemonic
named constant
naming conventions
narrowing conversion
Option Strict
precedence
scope (of a variable)
single-step
Step Into command
string concatenation
tab order
tab order selection mode
TabIndex property
TabStop property
text box
TextBox control
ToString method
Try-Catch block
type conversion error
type mismatch error
variable
variable declaration
unary operator
unhandled exception
Watch window
widening conversion

Review Questions and Exercises

Fill-in-the-Blank

1. The _____ control allows you to capture input the user has typed on the keyboard.
2. _____ is the standard prefix for TextBox control names.
3. _____ means that one string is appended to another.
4. The _____ character allows you to break a long statement into two or more lines of code.
5. The _____ character is actually two characters: a space followed by an underscore.
6. The control that has the _____ is the one that receives the user's keyboard input or mouse clicks.
7. The order in which controls receive the focus is called the _____.
8. You can modify the tab order by changing a control's _____ property.
9. If you do not want a control to receive the focus when the user presses the `[Tab]` key, set its _____ property to *False*.
10. An access key is a key that you press in combination with the _____ key to access a control such as a button quickly.
11. You define a button's access key through its _____ property.
12. A(n) _____ is a storage location in the computer's memory, used for holding information while the program is running.
13. A(n) _____ is a statement that causes Visual Basic to create a variable in memory.
14. A variable's _____ determines the type of information the variable can hold.
15. A(n) _____ variable is declared inside a procedure.
16. A(n) _____ error is generated anytime a nonnumeric value that cannot be automatically converted to a numeric value is assigned to a numeric variable or property.
17. A(n) _____ is a specialized routine that performs a specific operation, and then returns a value.
18. The _____ function converts an expression to an integer.
19. The _____ format string, when passed to the `ToString` method, produces a number in Currency format.
20. A(n) _____ is information that is being passed to a function.
21. When two operators share an operand, the operator with the highest _____ executes first.
22. A(n) _____ is like a variable whose content is read-only; it cannot be changed while the program is running.
23. A(n) _____ appears as a rectangular border with an optional title.

24. A form's _____ procedure executes each time a form loads into memory.
25. A(n) _____ is a line of code that causes a running application to pause execution and enter Break mode.

True or False

Indicate whether the following statements are true or false.

1. T F: The TextBox control's Text property holds the text entered by the user into the TextBox control at runtime.
2. T F: You can access a TextBox control's Text property in code.
3. T F: The string concatenation operator automatically inserts a space between the joined strings.
4. T F: You cannot break up a word with the line-continuation character.
5. T F: You can put a comment at the end of a line, after the line-continuation character.
6. T F: Only controls capable of receiving input, such as TextBox controls and buttons, may have the focus.
7. T F: You can cause a control to be skipped in the tab order by setting its TabPosition property to *False*.
8. T F: An error will occur if you assign a negative value to the TabIndex property in code.
9. T F: A control whose Visible property is set to *False* still receives the focus.
10. T F: GroupBox and Label controls have a TabIndex property, but they are skipped in the tab order.
11. T F: When you assign an access key to a button, the user can trigger a Click event by typing [Alt]+ the access key character.
12. T F: A local variable may be accessed by any other procedure in the same Form file.
13. T F: When a string variable is created in memory, Visual Basic assigns it the initial value 0.
14. T F: A variable's scope is the time during which the variable exists in memory.
15. T F: A variable declared inside a procedure is only visible to statements inside the same procedure.
16. T F: The CDb1 function converts a number to a string.
17. T F: If the CInt function cannot convert its argument, it causes a runtime error.
18. T F: The multiplication operator has higher precedence than the addition operator.
19. T F: A named constant's value can be changed by a programming statement, while the program is running.
20. T F: The statement `lblMessage.BackColor = Color.Green` will set lblMessage control's background color to green.
21. T F: You can select multiple controls simultaneously with the mouse.
22. T F: You can change the same property for multiple controls simultaneously.

23. T F: To group controls in a group box, draw the controls first, then draw the group box around them.
24. T F: While single-stepping through an application's code in Break mode, the highlighted execution point is the line of code that has already executed.

Multiple Choice

1. When the user types input into a TextBox control, in which property is it stored?
 - a. Input
 - b. Text
 - c. Value
 - d. Keyboard
2. Which character is the string concatenation operator?
 - a. &
 - b. *
 - c. %
 - d. @
3. In code, you move the focus to a control with which method?
 - a. MoveFocus
 - b. SetFocus
 - c. ResetFocus
 - d. Focus
4. Which form property allows you to specify a button to be clicked when the user presses the Enter key?
 - a. DefaultButton
 - b. AcceptButton
 - c. CancelButton
 - d. EnterButton
5. Which form property allows you to specify a button that is to be clicked when the user presses the Esc key?
 - a. DefaultButton
 - b. AcceptButton
 - c. CancelButton
 - d. EnterButton
6. You can modify a control's position in the tab order by changing which property?
 - a. TabIndex
 - b. TabOrder
 - c. TabPosition
 - d. TabStop
7. You assign an access key to a button through which property?
 - a. AccessKey
 - b. AccessButton
 - c. Mnemonic
 - d. Text
8. A group box's title is stored in which property?
 - a. Title
 - b. Caption
 - c. Text
 - d. Heading

9. You declare a named constant with which keyword?
 - a. `Constant`
 - b. `Const`
 - c. `NamedConstant`
 - d. `Dim`
10. Which of the following is the part of a program in which a variable is visible and may be accessed by programming statement?
 - a. segment
 - b. lifetime
 - c. scope
 - d. module
11. If a variable named `dblTest` contains the value 1.23456, then which of the following values will be returned by the expression `dblTest.ToString("N3")`?
 - a. 1.23456
 - b. 1.235
 - c. 1.234
 - d. +1.234
12. If the following code executes, which value is assigned to `strA`?


```
Dim dblTest As Double = 0.25
Dim strA = dblTest.ToString("p")
```

 - a. "0.25"
 - b. "2.50"
 - c. "25.00"
 - d. "0.25"

Short Answer

1. Describe the difference between the Label control's Text property and the TextBox control's Text property.
2. How do you clear the contents of a text box?
3. What is the focus when referring to a running application?
4. Write a statement that sets the focus to the `txtPassword` control.
5. How does Visual Basic automatically assign the tab order to controls?
6. How does a control's `TabIndex` property affect the tab order?
7. How do you assign an access key to a button?
8. How does assigning an access key to a button change the button's appearance?
9. What is the difference between the Single and Integer data types?
10. Create variable names that would be appropriate for holding each of the following information items:
 - a. The number of backpacks sold this week
 - b. The number of pounds of dog food in storage
 - c. Today's date
 - d. An item's wholesale price
 - e. A customer's name
 - f. The distance between two galaxies, in kilometers
 - g. The number of the month (1 = January, 2 = February, and so on)

11. Why should you always make sure that a string variable is initialized or assigned a value before it is used in an operation?
12. When is a local variable destroyed?
13. How would the following strings be converted by the `CDec` function?
 - a. "22.9000"
 - b. "1xfc47uvy"
 - c. "\$19.99"
 - d. "0.05%"
 - e. `String.Empty`
14. Briefly describe how the `CDec` function converts a string argument to a number.
15. Complete the following table by providing the value of each mathematical expression:

Expression	Value
$5 + 2 * 8$	_____
$20 / 5 - 2$	_____
$4 + 10 * 3 - 2$	_____
$(4 + 10) * 3 - 2$	_____
16. Assuming that the variable `dblTest` contains the value 67521.584, complete the following table, showing the value returned by each function call:

Function Call	Return Value
<code>dblTest.ToString("d2")</code>	_____
<code>dblTest.ToString("c2")</code>	_____
<code>dblTest.ToString("e1")</code>	_____
<code>dblTest.ToString("f2")</code>	_____
17. Describe one way to select multiple controls in Design mode.
18. Describe three ways to set a breakpoint in an application's code.

What Do You Think?

1. Why doesn't Visual Basic automatically insert a space between strings concatenated with the `&` operator?
2. Why would you want to use the line-continuation character to cause a statement to span multiple lines?
3. Why are Label controls not capable of receiving the focus?
4. Why should the tab order of controls in your application be logical?
5. Why assign access keys to buttons?
6. What is the significance of showing an underlined character on a button?
7. Generally speaking, which button should be set as a form's default button?
8. Why can't you perform arithmetic operations on a string, such as "28.9"?
9. Suppose a number is used in calculations throughout a program and must be changed every few months. What benefit is there to using a named constant to represent the number?
10. How can you get your application to execute a group of statements each time a form is loaded into memory?
11. How can you place an existing control in a group box?

- Visual Basic automatically reports syntax errors. Why doesn't it automatically report logic errors?

Find the Error

- Load the *Chap3\Error1\Error1* project from the student sample programs folder. Run the application. Type **2**, **4**, and **6** into the three TextBox controls, and then click the *Show Sum* button. The application reports the sum as 246. Fix the application so it correctly displays the sum of the numbers.
- Load the *Chap3\Error2\Error2* project from the student sample programs folder. The application has an error. Find the error and fix it.
- Load the *Chap3\Error3\Error3* project from the student sample programs folder. The *btnCalculate_Click* procedure contains an error. Find the error and fix it.

Algorithm Workbench

- Create a flowchart that shows the necessary steps for making the cookies in the following recipe:

Ingredients:

1/2 cup butter	1/2 teaspoon vanilla
1 egg	1/2 teaspoon salt
1 cup sifted all-purpose flour	1/2 teaspoon baking soda
1/2 cup brown sugar	1/2 cup chopped nuts
1/2 cup sugar	1/2 cup semisweet chocolate chips

Steps:

Preheat oven to 375°.
 Cream the butter.
 Add the sugar and the brown sugar to the butter and beat until creamy.
 Beat the egg and vanilla into the mixture.
 Sift and stir the flour, salt, and baking soda into the mixture.
 Stir the nuts and chocolate chips into the mixture.
 Shape the mixture into 1/2-inch balls.
 Place the balls about one inch apart on a greased cookie sheet.
 Bake for 10 minutes.

- A hot dog, still in its package, should be heated for 40 seconds in a microwave. Draw a flowchart showing the necessary steps to cook the hot dog.
- The following pseudocode algorithm for the event procedure *btnCalcArea_Click* has an error. The event procedure is supposed to calculate the area of a room's floor. The area is calculated as the room's width (entered by the user into *txtWidth*), multiplied by the room's length (entered by the user into *txtLength*). The result is displayed with the label *lblArea*. Find the error and correct the algorithm.
 - Multiply the *intWidth* variable by the *intLength* variable and store the result in the *intArea* variable.
 - Copy the value in *txtWidth.Text* into the *intWidth* variable.
 - Copy the value in *txtLength.Text* into the *intLength* variable.
 - Copy the value in the *intArea* variable into *lblArea.Text*.
- The following steps should be followed in the event procedure *btnCalcAvailCredit_Click*, which calculates a customer's available credit. Construct a flowchart that shows these steps.

- a. Copy the value in the TextBox control `txtMaxCredit` into the variable `decMaxCredit`.
 - b. Copy the value in the TextBox control `txtUsedCredit` into the variable `decUsedCredit`.
 - c. Subtract the value in `decUsedCredit` from `decMaxCredit`. Store the result in `decAvailableCredit`.
 - d. Copy the value in `decAvailableCredit` into the label `lblAvailableCredit`.
5. Convert the flowchart you constructed in Exercise 4 into Visual Basic code.
 6. Design a flowchart or pseudocode for the event procedure `btnCalcSale_Click`, which calculates the total of a retail sale. Assume the program uses `txtRetailPrice`, a TextBox control that holds the retail price of the item being purchased, and `decTAX_RATE`, a constant that holds the sales tax rate. The event procedure uses the items above to calculate the sales tax for the purchase and the total of the sale. Display the total of the sale in a label named `lblTotal`.
 7. Convert the flowchart or pseudocode you constructed in Exercise 6 into Visual Basic code.

Programming Challenges



VideoNote

The Miles
per Gallon
Calculator
Problem

1. Miles per Gallon Calculator

Create an application that calculates a car's gas mileage. The formula for calculating the miles that a car can travel per gallon of gas is:

$$MPG = \frac{\text{miles}}{\text{gallons}}$$

In the formula *MPG* is miles-per-gallon, *miles* is the number of miles that can be driven on a full tank of gas, and *gallons* is the number of gallons that the tank holds.

The application's form should have TextBox controls that let the user enter the number of gallons of gas the tank holds, and the number of miles the car can be driven on a full tank. When the *Calculate MPG* button is clicked, the application should display the number of miles that the car can be driven per gallon of gas. The form should also have a *Clear* button that clears the input and results, and an *Exit* button that ends the application. The application's form should appear as shown in Figure 3-61.

Figure 3-61 Miles per Gallon Calculator

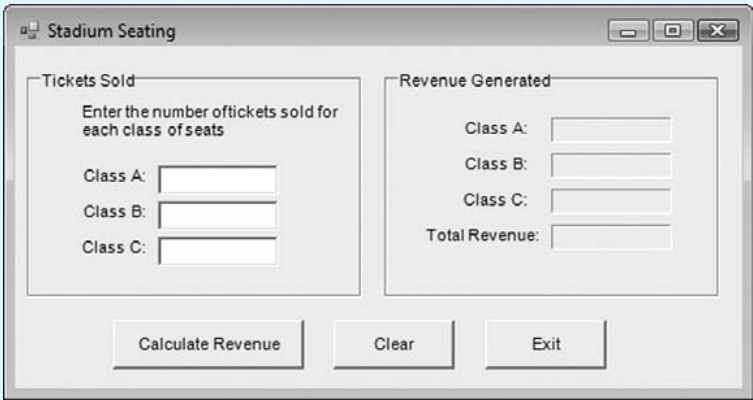
Use the following set of test data to determine if the application is calculating properly:

Gallons	Miles	Miles per Gallon
10	375	37.50
12	289	24.08
15	190	12.67

2. Stadium Seating

There are three seating categories at an athletic stadium. For a baseball game, Class A seats cost \$15 each, Class B seats cost \$12 each, and Class C seats cost \$9 each. Create an application that allows the user to enter the number of tickets sold for each class. The application should be able to display the amount of income generated from each class of ticket sales and the total revenue generated. The application’s form should resemble the one shown in Figure 3-62.

Figure 3-62 Stadium Seating form



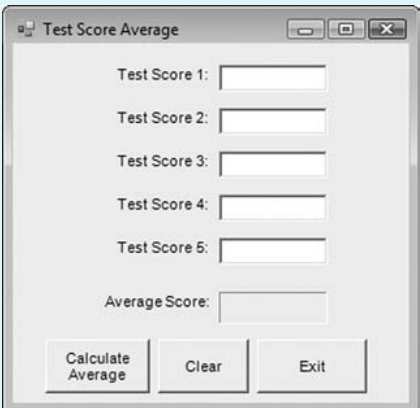
Use the following test data to determine if the application is calculating properly:

Ticket Sales	Revenue
Class A: 320	Class A: \$4,800.00
Class B: 570	Class B: \$6,840.00
Class C: 890	Class C: \$8,010.00
	Total Revenue: \$19,650.00
Class A: 500	Class A: \$7,500.00
Class B: 750	Class B: \$9,000.00
Class C: 1,200	Class C: \$10,800.00
	Total Revenue: \$27,300.00
Class A: 100	Class A: \$1,500.00
Class B: 300	Class B: \$3,600.00
Class C: 500	Class C: \$4,500.00
	Total Revenue: \$9,600.00

3. Test Score Average

Create an application that allows the user to enter five test scores. It should be able to calculate and display the average score. The application’s form should resemble the one shown in Figure 3-63. Notice that the labels next to each TextBox control have been assigned an access key. As described in this chapter, use a label to assign an access key indirectly to a TextBox control.

Figure 3-63 Test Score Average form



Use the following test data to determine if the application is calculating properly:

Test Scores	Averages
Test Score 1: 85	Average: 86.60
Test Score 2: 90	
Test Score 3: 78	
Test Score 4: 88	
Test Score 5: 92	
Test Score 1: 90	Average: 70.00
Test Score 2: 80	
Test Score 3: 70	
Test Score 4: 60	
Test Score 5: 50	
Test Score 1: 100	Average: 82.2
Test Score 2: 92	
Test Score 3: 56	
Test Score 4: 89	
Test Score 5: 74	

4. Theater Revenue

A movie theater only keeps a percentage of the revenue earned from ticket sales. The remainder goes to the movie company. Create an application that calculates and displays the following figures for one night’s box office business at a theater:

- a. *Gross revenue for adult tickets sold.* This is the amount of money taken in for all adult tickets sold.
- b. *Net revenue for adult tickets sold.* This is the amount of money from adult ticket sales left over after the payment to the movie company has been deducted.
- c. *Gross revenue for child tickets sold.* This is the amount of money taken in for all child tickets sold.
- d. *Net revenue for child tickets sold.* This is the amount of money from child ticket sales left over after the payment to the movie company has been deducted.
- e. *Total gross revenue.* This is the sum of gross revenue for adult and child tickets sold.
- f. *Total net revenue.* This is the sum of net revenue for adult and child tickets sold.

The application’s form should resemble the one shown in Figure 3-64.

Figure 3-64 Theater Revenue form

Assume the theater keeps 20% of its box office receipts. Use a named constant in your code to represent this percentage. Use the following test data to determine if the application is calculating properly:

Ticket Sales

Price per Adult Ticket: \$6.00
 Adult Tickets Sold: 120
 Price per Child Ticket: \$4.00

Ticket Sales (continued)

Child Tickets Sold: 72

Revenue

Gross Adult Ticket Sales: \$720.00
 Gross Child Ticket Sales: \$288.00
 Total Gross Revenue: \$1,008.00

Revenue (continued)

Net Adult Ticket Sales: \$144.00
 Net Child Ticket Sales: \$57.60
 Total Net Revenue: \$201.60

Design Your Own Forms**5. How Many Widgets?**

The Yukon Widget Company manufactures widgets that weigh 9.2 pounds each. Create an application that calculates how many widgets are stacked on a pallet, based on the total weight of the pallet. The user should be able to enter how much the pallet weighs alone and how much it weighs with the widgets stacked on it. The user should click a button to calculate and display the number of widgets stacked on the pallet. Use the following test data to determine if the application is calculating properly:

Pallet	Pallet and Widgets	Number of Widgets
100	5,620	600
75	1,915	200
200	9,400	1,000

6. Celsius to Fahrenheit

Create an application that converts Celsius to Fahrenheit. The formula is $F = 1.8 * C + 32$ where F is the Fahrenheit temperature and C is the Celsius temperature. Use the following test data to determine if the application is calculating properly:

Celsius	Fahrenheit
100	212
0	32
56	132.8

7. Currency

Create an application that converts U.S. dollar amounts to pounds, euros, and yen. The following conversion factors are not accurate, but you can use them in your application:

1 dollar = 0.68 pound

1 dollar = 0.83 euro

1 dollar = 108.36 yen

In your code, declare named constants to represent the conversion factors for the different types of currency. For example, you might declare the conversion factor for yen as follows:

```
Const dblYEN_FACTOR As Double = 108.36
```

Use the named constants in the mathematical conversion statements. Use the following test data to determine whether the application is calculating properly:

Dollars	Conversion Values
\$100.00	Pounds: 68
	Euros: 83
	Yen: 10,836
\$ 25.00	Pounds: 17
	Euros: 20.75
	Yen: 2,709
\$ 1.00	Pounds: 0.68
	Euros: 0.83
	Yen: 108.36

8. Monthly Sales Tax

A retail company must file a monthly sales tax report listing the total sales for the month, and the amount of state and county sales tax collected. The state sales tax rate is 4% and the county sales tax rate is 2%. Create an application that allows the user to enter the total sales for the month. From this figure, the application should calculate and display the following:

- The amount of county sales tax
- The amount of state sales tax
- The total sales tax (county plus state)

In the application's code, represent the county tax rate (0.02) and the state tax rate (0.04) as named constants. Use the named constants in the mathematical statements. Use the following test data to determine whether the application is calculating properly:

Total Sales	Tax Amounts
9,500	County sales tax: \$190.00
	State sales tax: \$380.00
	Total sales tax: \$570.00
5,000	County sales tax: \$100.00
	State sales tax: \$200.00
	Total sales tax: \$300.00
15,000	County sales tax: \$300.00
	State sales tax: \$600.00
	Total sales tax: \$900.00

9. Property Tax

A county collects property taxes on the assessment value of property, which is 60% of the property's actual value. If an acre of land is valued at \$10,000, its assessment value is \$6,000. The property tax is then \$0.64 for each \$100 of the assessment

value. The tax for the acre assessed at \$6,000 will be \$38.40. Create an application that displays the assessment value and property tax when a user enters the actual value of a property. Use the following test data to determine if the application is calculating properly:

Actual Property Value	Assessment and Tax
100,000	Assessment value: \$ 60,000.00 Property tax: 384.00
75,000	Assessment value: 45,000.00 Property tax: 288.00
250,000	Assessment value: 150,000.00 Property tax: 960.00

10. Pizza Pi

Joe's Pizza Palace needs an application to calculate the number of slices a pizza of any size can be divided into. The application should do the following:

- Allow the user to enter the diameter of the pizza, in inches.
- Calculate the number of slices that can be cut from a pizza that size.
- Display a message that indicates the number of slices.

To calculate the number of slices that can be cut from the pizza, you must know the following facts:

- Each slice should have an area of 14.125 inches.
- To calculate the number of slices, divide the area of the pizza by 14.125.

The area of the pizza is calculated with the following formula:

$$\text{Area} = \pi r^2$$



NOTE: π is the Greek letter pi. 3.14159 can be used as its value. The variable r is the radius of the pizza. Divide the diameter by 2 to get the radius.

Use the following test data to determine if the application is calculating properly:

Diameter of Pizza	Number of Slices
22 inches	27
15 inches	13
12 inches	8

11. Distance Traveled

Assuming there are no accidents or delays, the distance that a car travels down the interstate can be calculated with the following formula:

$$\text{Distance} = \text{Speed} \times \text{Time}$$

Create a VB application that allows the user to enter a car's speed in miles-per-hour. When a button is clicked, the application should display the following:

- The distance the car will travel in 5 hours
- The distance the car will travel in 8 hours
- The distance the car will travel in 12 hours

12. Tip, Tax, and Total

Create a VB application that lets the user enter the food charge for a meal at a restaurant. When a button is clicked, it should calculate and display the amount of a 15 percent tip, 7 percent sales tax, and the total of all three amounts.

13. Body Mass Index

Create a VB application that lets the user enter his or her weight (in pounds) and height (in inches). The application should calculate the user's body mass index (BMI). The BMI is often used to determine whether a person with a sedentary lifestyle is overweight or underweight for their height. A person's BMI is calculated with the following formula:

$$BMI = weight \times 703 / height^2$$

14. How Much Insurance?

Many financial experts advise that property owners should insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Create a VB application that lets the user enter the replacement cost of a building and then displays the minimum amount of insurance he or she should buy for the property.

15. How Many Calories?

A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 "servings" in the bag and that a serving equals 300 calories. Create a VB application that lets the user enter the number of cookies they actually ate and then reports the number of total calories consumed.

16. Automobile Costs

Create a VB application that lets the user enter the monthly costs for the following expenses incurred from operating his or her automobile: loan payment, insurance, gas, oil, tires, and maintenance. The program should then display the total monthly cost of these expenses and the total annual cost of these expenses.

