CHAPTER 8

Sub and Function Procedures

After studying Chapter 8, you should be able to:

- Explain the difference between Sub and Function procedures
- Create a Sub procedure
- Pass information to a procedure
- Explain the difference between passing data *by value* and passing data *by reference*
- Explain the purpose of the `sender` and `e` parameters
- Associate a procedure with more than one object and event
- Create a Function procedure
- Convert an Object variable to a different type using the TryCast operator
- Utilize a timer control

Procedures

As you already know, a procedure is a block of program code that performs a specific task. Most procedures in Visual Basic are either Sub procedures or Function procedures. The difference between both types of procedures is that a **Function procedure** returns a value after performing its assigned task, whereas a **Sub procedure** does not return a value. All of the applications coded in previous chapters contain Sub procedures that are predefined in Visual Basic, such as the Click, TextChanged, and KeyPress event procedures. In this chapter, you will learn how to create and use your own Sub procedures. You also will learn how to create and use Function procedures.

You also can create Property procedures in Visual Basic. Property procedures are covered in Chapter 12.

Sub Procedures

There are two types of Sub procedures in Visual Basic: event procedures and independent Sub procedures. The procedures coded in the previous chapters were event procedures. An event procedure is a Sub procedure that is associated with a specific object and event, such as a button's Click event or a text box's TextChanged event. The computer automatically processes an event procedure when the event occurs. An **independent Sub procedure**, on the other hand, is a procedure that is independent of any object and event. An independent Sub procedure is processed only when called (invoked) from code. Programmers use independent Sub procedures for several reasons. First, they allow the programmer to avoid duplicating code when different sections of a program need to perform the same task. Rather than enter the same code in each of those sections, the programmer can enter the code in a procedure and then have each section call the procedure to perform its task when needed. Second, consider an event procedure that must perform many tasks. To keep the event procedure's code from getting unwieldy and difficult to understand, the programmer can assign some of the tasks to one or more independent Sub procedures. Doing this makes the event procedure easier to code, because it allows the programmer to concentrate on one small piece of the code at a time. And finally, independent Sub procedures are used extensively in large and complex programs, which typically are written by a team of programmers. The programming team will break up the program into small and manageable tasks, and then assign some of the tasks to different team members to be coded as independent Sub procedures. Doing this allows more than one programmer to work on the program at the same time, decreasing the time it takes to write the program.

When you enter a procedure below the last event procedure in the Code Editor window, be sure to enter it above the End Class clause.

Figure 8-1 shows the syntax for creating an independent Sub procedure in Visual Basic. It also includes an example of an independent Sub procedure, as well as the steps for entering an independent Sub procedure in the Code Editor window. Some programmers enter independent Sub procedures above the first event procedure, while others enter them below the last event procedure. Still others enter them either immediately above or immediately below the procedure from which they are invoked. In this book, the independent Sub procedures will usually be entered above the first event procedure in the Code Editor window.

As the syntax in Figure 8-1 shows, independent Sub procedures have both a procedure header and a procedure footer. In most cases, the procedure header begins with the `Private` keyword, which indicates that

the procedure can be used only within the current Code Editor window. Following the `Private` keyword is the **Sub** keyword, which identifies the procedure as a Sub procedure. After the **Sub** keyword is the procedure name. The rules for naming an independent Sub procedure are the same as those for naming variables; however, procedure names are usually entered using Pascal case. The Sub procedure's name should indicate the task the procedure performs. It is a common practice to begin the name with a verb. For example, a good name for a Sub procedure that clears the contents of the label controls in an interface is ClearLabels.

Using Pascal case, you capitalize the first letter in the procedure name and the first letter of each subsequent word in the name.

Following the procedure name in the procedure header is a set of parentheses that contains an optional *parameterList*. The *parameterList* lists the data type and name of one or more memory locations, called parameters. The **parameters** store the information passed to the procedure when it is invoked. If the procedure does not require any information to be passed to it, as is the case with the ClearLabels procedure in Figure 8-1, an empty set of parentheses follows the procedure name in the procedure header. You will learn more about parameters later in this chapter. A Sub procedure ends with its procedure footer, which is always **End Sub**. Between the procedure header and procedure footer, you enter the instructions to be processed when the procedure is invoked.

HOW TO Create an Independent Sub Procedure

Syntax

Private Sub *procedureName*(**[***parameterList***]**) — procedure header
    statements
End Sub — procedure footer

Example

```
Private Sub ClearLabels()
    regularLabel.Text = String.Empty
    overtimeLabel.Text = String.Empty
    grossLabel.Text = String.Empty
End Sub
```

clears the contents of the labels

Steps

1. Click a blank line in the Code Editor window. The blank line can be anywhere between the Public Class and End Class clauses. However, it must be outside any other Sub or Function procedure.
2. Type the Sub procedure header and then press Enter. The Code Editor automatically enters the End Sub clause for you.

Figure 8-1 How to create an independent Sub procedure

You can invoke an independent Sub procedure using the **Call statement**. Figure 8-2 shows the statement's syntax and includes an example of using the statement to invoke the ClearLabels procedure from Figure 8-1. In the syntax, *procedureName* is the name of the procedure you are calling (invoking), and *argumentList* (which is optional) is a comma-separated list of arguments you want passed to the procedure. If you have no information to pass to the procedure that you are calling, as is the case with the ClearLabels

The Call keyword is optional when invoking a Sub procedure. Therefore, you also can call the ClearLabels procedure using the statement `ClearLabels()`.

procedure, you include an empty set of parentheses after the *procedureName* in the Call statement. The ClearLabels procedure is used in the Gadis Antiques application, which you view in the next section.

HOW TO Call an Independent Sub Procedure

Syntax
Call *procedureName*([*argumentList*])

Example
```
Call ClearLabels()
```

Figure 8-2 How to call an independent Sub procedure

The Gadis Antiques Application

The manager at Gadis Antiques wants an application that calculates an employee's regular pay, overtime pay, and gross pay. Employees are paid on an hourly basis and receive time and one-half for the hours worked over 40. Figure 8-3 shows a sample run of the Gadis Antiques application, and Figure 8-4 shows a portion of the application's code. When the user clicks the Clear button in the interface, the button's Click event procedure should clear the contents of the regularLabel, overtimeLabel, and grossLabel. The labels also should be cleared in the TextChanged event procedures for the two combo boxes. Recall that the TextChanged event occurs when the text portion of a combo box changes. You can clear the labels by entering the appropriate assignment statements in the Click event procedure and in both TextChanged event procedures. Or, you can enter the assignment statements in an independent Sub procedure and then call the Sub procedure from each of the three event procedures, as shown in Figure 8-4. Entering the code in an independent Sub procedure saves you from having to enter the same statements more than once. In addition, if you subsequently need to assign the string "0.00" rather than the empty string to the labels, you will need to make the change in only one place in the code.

If you want to experiment with the Gadis Antiques application, open the solution contained in the Try It 1! folder.

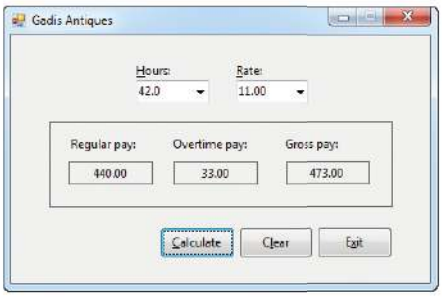

Figure 8-3 Sample run of the Gadis Antiques application

```
Public Class MainForm

    Private Sub ClearLabels()
        regularLabel.Text = String.Empty
        overtimeLabel.Text = String.Empty
        grossLabel.Text = String.Empty

    End Sub

    Private Sub MainForm_Load...

    Private Sub exitButton_Click...

    Private Sub calcButton_Click...

    Private Sub clearButton_Click(ByVal sender As Object,
    ByVal e As System.EventArgs) Handles clearButton.Click
        Call ClearLabels()
    End Sub

    Private Sub hoursComboBox_TextChanged(
    ByVal sender As Object, ByVal e As System.EventArgs
    ) Handles hoursComboBox.TextChanged
        Call ClearLabels()
    End Sub

    Private Sub rateComboBox_TextChanged(
    ByVal sender As Object, ByVal e As System.EventArgs
    ) Handles rateComboBox.TextChanged
        Call ClearLabels()
    End Sub
End Class
```

ClearLabels procedure

invokes the ClearLabels procedure

invokes the ClearLabels procedure

invokes the ClearLabels procedure

Figure 8-4 Partial code for the Gadis Antiques application

Notice that the ClearLabels procedure is entered above the event procedures in the code shown in Figure 8-4.

When the computer processes the `Call ClearLabels()` statement in the clearButton's Click event procedure, it temporarily leaves the procedure to process the code in the ClearLabels procedure. The assignment statements in the ClearLabels procedure remove the contents of the regularLabel, overtimeLabel, and grossLabel. After processing the assignment statements, the computer processes the ClearLabels procedure's End Sub clause, which ends the procedure. The computer then returns to the clearButton's Click event procedure and processes the line of code located immediately below the Call statement—in this case, the event procedure's End Sub clause. A similar process is followed when either combo box's TextChanged event occurs. The computer temporarily leaves the event procedure to process the code contained in the ClearLabels procedure. When the ClearLabels procedure ends, the computer returns to the event procedure and processes the code immediately below the Call statement.

Mini-Quiz 8-1

The answers to Mini-Quiz questions are located in Appendix A.

1. An event procedure is a Sub procedure that is associated with a specific object and event.

   a. True

   b. False

2. If the DisplayMessage procedure does not require any information to be passed to it, its procedure header should be _____.

   a. `Private Sub DisplayMessage`

   b. `Private Sub DisplayMessage()`

   c. `Private Sub DisplayMessage(none)`

   d. `Private Sub DisplayMessage[]`

3. Which of the following invokes the DisplayMessage procedure from Question 2?

   a. `Call DisplayMessage`

   b. `Call Sub DisplayMessage()`

   c. `Call DisplayMessage(none)`

   d. `Call DisplayMessage()`

Including Parameters in an Independent Sub Procedure

As mentioned earlier, an independent Sub procedure can contain one or more parameters in its procedure header; each parameter stores an item of data. The data is passed to the procedure through the *argumentList* in the Call statement. The number of arguments in the Call statement's argumentList should agree with the number of parameters in the procedure's parameterList. If the parameterList contains one parameter, then the argumentList should have one argument. Similarly, a procedure that contains three parameters requires three arguments in the Call statement. (Refer to the first Tip on this page for an exception to this general rule.) In addition to having the same number of arguments as parameters, the data type and position of each argument should agree with the data type and position of its corresponding parameter. For example, if the first parameter has a data type of String and the second a data type of Double, then the first argument in the Call statement should have the String data type and the second should have the Double data type. This is because, when the procedure is called, the computer stores the value of the first argument in the procedure's first parameter, the value of the second argument in the second parameter, and so on. An argument can be a literal constant, named constant, keyword, or variable; however, in most cases, it will be a variable.

Visual Basic allows you to specify that an argument in the Call statement is optional. To learn more about optional arguments, complete Computer Exercise 29 at the end of this chapter.

The internal memory of a computer is similar to a large post office. Like each post office box, each memory cell has a unique address.

Passing Variables

Every variable has both a value and a unique address that represents its location in the computer's internal memory. Visual Basic allows you to pass either a copy of the variable's value or the variable's address to the receiving procedure. Passing a copy of the variable's value is referred to as **passing**

by value. Passing a variable's address is referred to as **passing by reference**. The method you choose—*by value* or *by reference*—depends on whether you want the receiving procedure to have access to the variable in memory. In other words, it depends on whether you want to allow the receiving procedure to change the variable's contents.

Although the idea of passing information *by value* and *by reference* may sound confusing at first, it is a concept with which you already are familiar. To illustrate, assume you have a savings account at a local bank. During a conversation with your friend Jake, you mention the amount of money you have in the account. Sharing this information with Jake is similar to passing a variable *by value*. Knowing your account balance does not give Jake access to your bank account. It merely provides information that he can use to compare to the balance in his savings account. The savings account example also provides an illustration of passing information *by reference*. To deposit money to or withdraw money from your account, you must provide the bank teller with your account number. The account number represents the location of your account at the bank and allows the teller to change the account balance. Giving the teller your bank account number is similar to passing a variable *by reference*. The account number allows the teller to change the contents of your bank account, similar to the way the variable's address allows the receiving procedure to change the contents of the variable.

Passing Variables by Value

To pass a variable *by value*, you include the keyword `ByVal` before the name of its corresponding parameter in the receiving procedure's parameterList. When you pass a variable *by value*, the computer passes a copy of the variable's contents to the receiving procedure. When only a copy of the contents is passed, the receiving procedure is not given access to the variable in memory. Therefore, it cannot change the value stored inside the variable. It is appropriate to pass a variable *by value* when the receiving procedure needs to *know* the variable's contents, but it does not need to *change* the contents. Unless you specify otherwise, variables in Visual Basic are automatically passed *by value*.

The Pet Information application provides an example of passing variables *by value*. Figure 8-5 shows a sample run of the application, and Figure 8-6 shows the code entered in two of the application's procedures: an independent Sub procedure named ShowMsg and the displayButton's Click event procedure. The Call statement in the Click event procedure passes two variables, *by value*, to the ShowMsg procedure: a String variable and an Integer variable. Notice that the number, data type, and sequence of the arguments in the Call statement match the number, data type, and sequence of the corresponding parameters in the ShowMsg procedure header. Also notice that the names of the arguments do not need to be identical to the names of the corresponding parameters. In fact, to avoid confusion, it usually is better to use different names for the arguments and parameters. The parameters in a procedure header have procedure scope, which means they can be used only by the procedure.

If you want to experiment with the Pet Information application, open the solution contained in the Try It 2! folder.

Figure 8-5 Sample run of the Pet Information application

The Call statement does not indicate whether a variable is being passed to a procedure *by value* or *by reference*. To make that determination, you need to look at the receiving procedure's header.

parameterList

```
Private Sub ShowMsg(ByVal pet As String, ByVal age As Integer)
      ' calculates age on next birthday, and
      ' displays name and next age in a message

      age = age + 1
      msgLabel.Text = "On his or her next birthday," &
            " your pet " & pet & " will be " &
            age & " years old."
End Sub


Private Sub displayButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles displayButton.Click
      ' gets the pet information, then calls
      ' a procedure to display the information

      Dim inputName As String
      Dim inputAge As String
      Dim currentAge As Integer

      inputName = InputBox("Pet's name:", "Name")
      inputAge = InputBox("Pet's age (years):", "Age")
      Integer.TryParse(inputAge, currentAge)

      Call ShowMsg(inputName, currentAge)
End Sub
```

you also can use age += 1

argumentList

Figure 8-6 ShowMsg procedure and displayButton Click event procedure

The InputBox functions in the displayButton's Click event procedure prompt the user to enter the pet's name and age. If the user enters Micky as the name and 8 as the age, the functions store the strings "Micky" and "8" in the `inputName` and `inputAge` variables, respectively. The TryParse method then converts the contents of the String `inputAge` variable to Integer and stores the result in the Integer `currentAge` variable. Next, the Call statement calls the ShowMsg procedure, passing it the `inputName` and `currentAge` variables *by value*. You can tell that the variables are passed *by value* because the keyword `ByVal` appears before each variable's corresponding parameter in the ShowMsg procedure header. Passing both variables *by value* means that only a copy of the contents of both variables—in this case, the string "Micky" and the integer 8—are passed to the procedure. At this point, the computer temporarily leaves the displayButton's Click event procedure to process the ShowMsg procedure.

The parameterList in the ShowMsg procedure header tells the computer to create two procedure-level variables: a String variable named **pet** and an Integer variable named **age**. The computer stores a copy of the information passed to the procedure in those variables. In this case, it stores the string "Micky" in the **pet** variable and the integer 8 in the **age** variable. Next, the computer processes the statements contained within the procedure. The first statement adds the number 1 to the contents of the **age** variable and then assigns the sum (9) to the **age** variable. The last statement in the procedure displays a message that contains the values stored in the **pet** and **age** variables. As shown earlier in Figure 8-5, the statement displays the message "On his or her next birthday, your pet Micky will be 9 years old." The ShowMsg procedure footer is processed next and ends the procedure. At this point, the **pet** and **age** variables are removed from the computer's internal memory. (Recall that a procedure-level variable is removed from the computer's memory when the procedure ends.) The computer then returns to the displayButton's Click event procedure to process the line of code immediately following the Call statement. In this case, the line of code is the End Sub clause, which ends the event procedure. The computer then removes the **inputName**, **inputAge**, and **currentAge** procedure-level variables from its internal memory.

Passing Variables by Reference

Instead of passing a copy of a variable's value to a procedure, you can pass the variable's address. In other words, you can pass the variable's location in the computer's internal memory. As you learned earlier, passing a variable's address is referred to as passing *by reference*, and it gives the receiving procedure access to the variable being passed. You pass a variable *by reference* when you want the receiving procedure to change the contents of the variable. To pass a variable *by reference* in Visual Basic, you include the keyword **ByRef** before the name of its corresponding parameter in the receiving procedure's parameterList. The **ByRef** keyword tells the computer to pass the variable's address rather than a copy of its contents.

The Gross Pay application provides an example of passing a variable *by reference*. Figure 8-7 shows a sample run of the application, and Figure 8-8 shows the code entered in two of the application's procedures: an independent Sub procedure named CalcGrossPay and the calcButton's Click event procedure. Here again, notice that the number, data type, and sequence of the arguments in the Call statement match the number, data type, and sequence of the corresponding parameters in the procedure header. Also notice that the names of the arguments are not identical to the names of their corresponding parameters. The parameterList indicates that the first two variables in the argumentList are passed *by value*, and the third variable is passed *by reference*.

Figure 8-7 Sample run of the Gross Pay application

If you want to experiment with the Gross Pay application, open the solution contained in the Try It 3! folder.

```
Private Sub CalcGrossPay(ByVal hours As Double,
                         ByVal rate As Double,
                         ByRef gross As Double)
    ' calculates the gross pay

    gross = hours * rate
    ' add overtime, if necessary
    If hours > 40 Then
        gross = gross + (hours - 40) * rate / 2
    End If
End Sub


Private Sub calcButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles calcButton.Click
    ' displays gross pay

    Dim hoursWkd As Double
    Dim rateOfPay As Double
    Dim grossPay As Double

    hoursWkd =
        Convert.ToDouble(hoursListBox.SelectedItem)
    rateOfPay =
        Convert.ToDouble(rateListBox.SelectedItem)

    ' use a Sub procedure to calculate the gross pay
    Call CalcGrossPay(hoursWkd, rateOfPay, grossPay)

    grossLabel.Text = grossPay.ToString("C2")
End Sub
```

parameterList

passed by value

passed by reference

Figure 8-8 CalcGrossPay procedure and calcButton Click event procedure

Desk-checking the procedures shown in Figure 8-8 will help clarify the difference between passing *by value* and passing *by reference*. **Desk-checking** refers to the process of reviewing the program instructions while seated at your desk rather than in front of the computer. Desk-checking is also called **hand-tracing**, because you use a pencil and paper to follow each of the instructions by hand. Before you begin the desk-check, you first choose a set of sample data for the input values, which you then use to manually compute the expected output values. You will desk-check Figure 8-8's procedures using 43 and $7.50 as the hours worked and rate of pay, respectively. The expected gross pay amount is $333.75, as shown in Figure 8-9.

43	hours worked
* 7.50	hourly pay rate
322.50	regular pay for 43 hours
+ 11.25	overtime pay for 3 hours at $3.75 per hour
333.75	gross pay

Figure 8-9 Gross pay calculation using sample input values

When the user clicks the Calculate button after selecting 43.0 and 7.50 in the hoursListBox and rateListBox, respectively, the Dim statements in the

button's Click event procedure create and initialize three Double variables. Next, the two Convert.ToDouble methods convert the items selected in the list boxes to Double, storing the results in the `hoursWkd` and `rateOfPay` variables. Figure 8-10 shows the contents of the variables before the Call statement is processed.

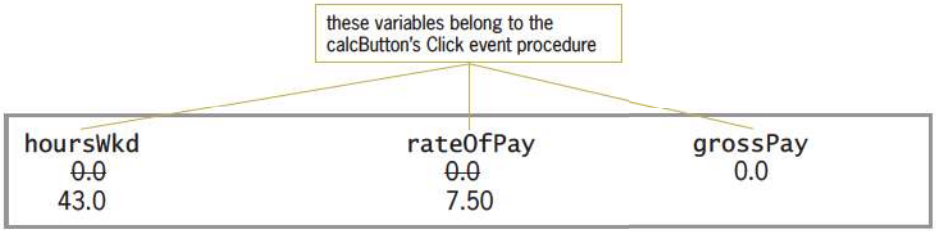

Figure 8-10 Desk-check table before the Call statement is processed

The computer processes the Call statement next. The Call statement invokes the CalcGrossPay procedure, passing it three arguments. At this point, the computer temporarily leaves the Click event procedure to process the code contained in the CalcGrossPay procedure; the procedure header is processed first. The `ByVal` keyword indicates that the first two parameters are receiving values from the Call statement—in this case, copies of the numbers stored in the `hoursWkd` and `rateOfPay` variables. As a result, the computer creates the `hours` and `rate` variables listed in the parameterList, and stores the numbers 43.0 and 7.50, respectively, in the variables. The `ByRef` keyword indicates that the third parameter is receiving the address of a variable. When you pass a variable's address to a procedure, the computer uses the address to locate the variable in its internal memory. It then assigns the parameter name to the memory location. In this case, the computer locates the `grossPay` variable in memory and assigns the name `gross` to it. At this point, the memory location has two names: one assigned by the calcButton's Click event procedure and the other assigned by the CalcGrossPay procedure, as indicated in Figure 8-11. Notice that two of the variables in the figure belong strictly to the Click event procedure, and two belong strictly to the CalcGrossPay procedure. One memory location, however, belongs to both procedures. Although both procedures can access the memory location, each procedure uses a different name to do so. The Click event procedure uses the name `grossPay`, whereas the CalcGrossPay procedure uses the name `gross`.

Although the `grossPay` and `gross` variables refer to the same location in memory, the `grossPay` variable is recognized only within the calcButton's Click event procedure. Similarly, the `gross` variable is recognized only within the CalcGrossPay procedure.

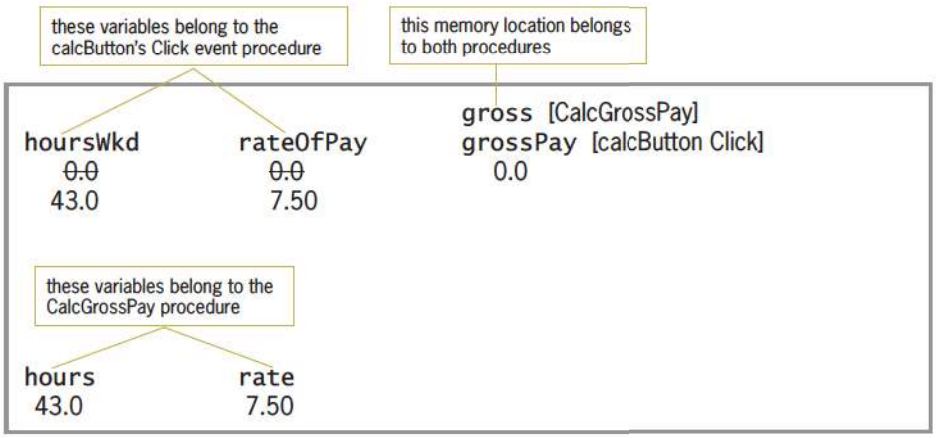

Figure 8-11 Desk-check table after the Call statement and CalcGrossPay procedure header are processed

After processing the CalcGrossPay procedure header, the computer processes the code contained in the procedure. The first statement calculates the gross pay by multiplying the contents of the **hours** variable (43.0) by the contents of the **rate** variable (7.50), and then assigns the result (322.50) to the **gross** variable. Figure 8-12 shows the desk-check table after the statement is processed. Notice that when the value in the **gross** variable changes, the value in the **grossPay** variable also changes. This happens because the names **gross** and **grossPay** refer to the same location in the computer's internal memory.

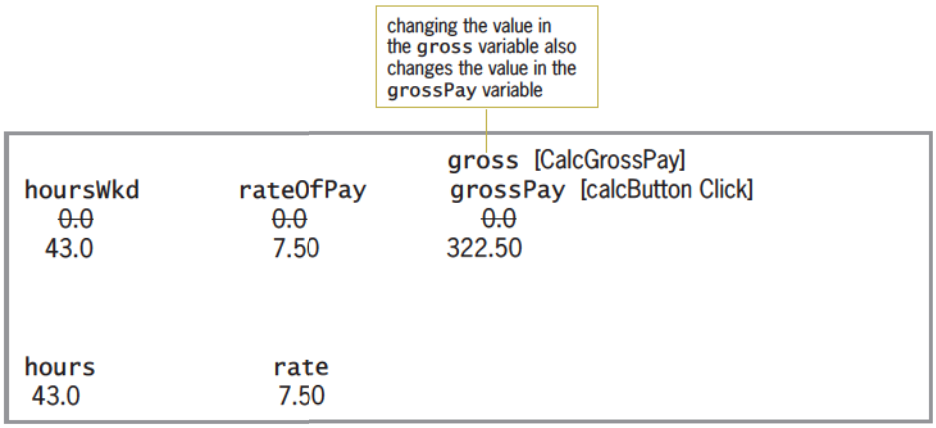

Figure 8-12 Desk-check table after the first statement in the CalcGrossPay procedure is processed

The **hours** variable contains a value that is greater than 40, so the statement in the selection structure's true path calculates the overtime pay (11.25) and adds it to the regular pay (322.50); it assigns the result (333.75) to the **gross** variable. The result agrees with the manual calculation performed earlier. Figure 8-13 shows the desk-check table after the statement is processed.

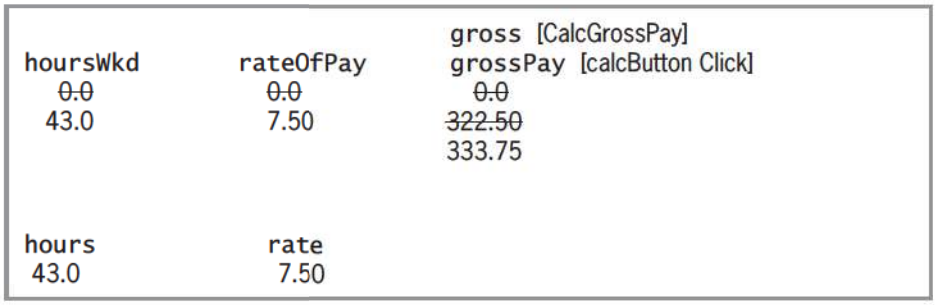

Figure 8-13 Desk-check table after the statement in the selection structure's true path is processed

The CalcGrossPay procedure's End Sub clause is processed next and ends the procedure. At this point, the computer removes the **hours** and **rate** variables from memory. It also removes the **gross** name from the appropriate location in memory, as indicated in Figure 8-14. Notice that the **grossPay** memory location now has only one name: the name assigned to it by the calcButton's Click event procedure.

hoursWkd	rateOfPay	~~gross~~ [CalcGrossPay] grossPay [calcButton Click]
~~0.0~~	~~0.0~~	~~0.0~~
43.0	7.50	~~322.50~~
		333.75
~~hours~~	~~rate~~	
~~43.0~~	~~7.50~~	

Figure 8-14 Desk-check table after the CalcGrossPay procedure ends

After the CalcGrossPay procedure ends, the computer returns to the line of code below the Call statement in the calcButton's Click event procedure. In this case, it returns to the `grossLabel.Text = grossPay.ToString("C2")` statement, which formats the contents of the `grossPay` variable and displays the result ($333.75) in the grossLabel, as shown earlier in Figure 8-7. Finally, the computer processes the Click event procedure's End Sub clause. When the Click event procedure ends, the computer removes the procedure's variables (`hoursWkd`, `rateOfpay`, and `grossPay`) from memory.

Mini-Quiz 8-2

The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following is a valid procedure header for a procedure that receives a copy of the values stored in two String variables?
   a. `Private Sub Display(ByRef x As String, ByRef y As String)`
   b. `Private Sub Display(ByVal x As String, ByVal y As String)`
   c. `Private Sub Display(ByValue x As String, ByValue y As String)`
   d. `Private Sub Display(ByCopy x As String, ByCopy y As String)`

2. Which of the following indicates that the procedure will receive two items of data: an integer and the address of a Double variable?
   a. `Private Sub Calc(ByVal x As Integer, ByRef y As Double)`
   b. `Private Sub Calc(Value x As Integer, Address y As Double)`
   c. `Private Sub Calc(ByInt x As Integer, ByAdd y As Double)`
   d. `Private Sub Calc(ByCopy x As Integer, ByAdd y As Double)`

3. Which of the following invokes the Calc procedure from Question 2, passing it an Integer variable named `sales` and a Double variable named `bonus`?

   a. `Call Calc(ByVal sales, ByRef bonus)`

   b. `Call Calc(sales, bonus)`

   c. `Call Calc(bonus, sales)`

   d. both b and c

Associating a Procedure with Different Objects and Events

As you learned in Chapter 1, the Handles clause in an event procedure's header indicates the object and event associated with the procedure. The `Handles clearButton.Click` clause in Figure 8-15, for example, indicates that the clearButton_Click procedure is associated with the Click event of the clearButton. As a result, the procedure will be processed when the clearButton's Click event occurs. Although an event procedure's name—in this case, clearButton_Click—contains the names of its associated object and event, that is not a requirement. You can change the name of an event procedure to any name that follows the naming rules for procedures. For example, you can change the name clearButton_Click to Clear and the procedure will still work correctly. This is because the Handles clause, rather than the event procedure's name, determines when the procedure is processed.

```
Private Sub ClearLabels()
      regularLabel.Text = String.Empty
      overtimeLabel.Text = String.Empty
      grossLabel.Text = String.Empty
End Sub

Private Sub clearButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles clearButton.Click
      Call ClearLabels()
End Sub

Private Sub hoursComboBox_TextChanged(
ByVal sender As Object, ByVal e As System.EventArgs
) Handles hoursComboBox.TextChanged
      Call ClearLabels()
End Sub

Private Sub rateComboBox_TextChanged(
ByVal sender As Object, ByVal e As System.EventArgs
) Handles rateComboBox.TextChanged
      Call ClearLabels()
End Sub
```

procedure name

the Handles clause determines when the procedure is processed

Figure 8-15 Some of the Gadis Antiques application's code from Figure 8-4

You can associate a procedure with more than one object and event, as long as each event contains the same parameters in its procedure header. In the code shown in Figure 8-15, for instance, you can associate the ClearLabels procedure with the clearButton's Click event and the TextChanged events for the two combo boxes. This is because the three event procedures have the same parameters in their procedure header: `sender` and `e`. You may have noticed that all event procedures contain the `sender` and `e` parameters in their procedure header. The **sender parameter** contains the memory address of the object that raised the event (in other words, caused the event to occur). For example, when the clearButton's Click event occurs, the button's address in memory is stored in the `sender` parameter. Similarly, when the TextChanged event occurs for either of the combo boxes, the corresponding combo box's memory address is stored in the `sender` parameter. The **e parameter** in an event procedure's header contains additional information provided by the object that raised the event. The `e` parameter in the KeyPress event procedure's header, for instance, contains a character that corresponds to the key pressed by the user. You can determine the items of information contained in an event procedure's `e` parameter by viewing its properties. You do this by displaying the event procedure's code template in the Code Editor window, and then typing the letter e followed by a period. The Code Editor displays a list that includes the `e` parameter's properties.

You learned about a text box's KeyPress event in Chapter 5.

To associate a procedure with more than one object and event, you enter the appropriate parameters in the procedure's parameterList, and then enter each object and event in the procedure's Handles clause. You use commas to separate each parameter and also to separate each object and event. In the ClearLabels procedure in Figure 8-16, the Handles clause tells the computer to process the procedure's code when any of the following occurs: the clearButton's Click event, the hoursComboBox's TextChanged event, or the rateComboBox's TextChanged event. The ClearLabels procedure in Figure 8-16 can be used in place of the ClearLabels, clearButton_Click, hoursComboBox_TextChanged, and rateComboBox_TextChanged procedures shown in Figure 8-15.

If you want to experiment with the code shown in Figure 8-16, open the solution contained in the Try It 4! folder.

```
Private Sub ClearLabels(ByVal sender As Object,
                        ByVal e As System.EventArgs
                        ) Handles clearButton.Click,
                        hoursComboBox.TextChanged,
                        rateComboBox.TextChanged
     regularLabel.Text = String.Empty
     overtimeLabel.Text = String.Empty
     grossLabel.Text = String.Empty
End Sub
```

parameterList

the Handles clause associates the procedure with three objects and events

Figure 8-16 ClearLabels procedure

Function Procedures

In addition to creating Sub procedures in Visual Basic, you also can create Function procedures. Recall that the difference between both types of procedures is that a Function procedure returns a value after performing its assigned task, whereas a Sub procedure does not return a value. Function procedures are

referred to more simply as **functions**. Figure 8-17 shows the syntax for creating a function in Visual Basic. The header and footer in a function are almost identical to the header and footer in a Sub procedure, except the function's header and footer contain the `Function` keyword rather than the `Sub` keyword. Also different from a Sub procedure header, a function's header includes the `As` *dataType* section, which specifies the data type of the value returned by the function. As is true with a Sub procedure, a function can receive information either *by value* or *by reference*. The information it receives is listed in the parameterList in the header. Between the function's header and footer, you enter the instructions to process when the function is invoked. In most cases, the **Return statement** is the last statement within a function. The Return statement's syntax is `Return` *expression*, where *expression* represents the one and only value that will be returned to the statement invoking the function. The data type of the *expression* must agree with the data type specified in the `As` *dataType* section.

In addition to the syntax, Figure 8-17 also includes two examples of a function, as well as the steps you follow to enter a function in the Code Editor window. As with Sub procedures, you can enter your functions above the first event procedure, below the last event procedure, or immediately above or below the procedure from which they are invoked. In this book, you usually will enter the functions above the first event procedure. Like Sub procedure names, function names are entered using Pascal case and typically begin with a verb. The name should indicate the task the function performs. For example, a good name for a function that returns the area of a circle is GetCircleArea.

HOW TO Create a Function Procedure

specifies the data type of the return value

Syntax

function header

```
Private Function procedureName([parameterList]) As dataType
        statements
        Return expression
End Function
```

function footer

Example 1

returns the `area` variable's value to the statement that called the function

```
Private Function GetCircleArea(ByVal radius As Double) As Double
    Dim area As Double
    area = 3.141593 * radius ^ 2
    Return area
End Function
```

Example 2

calculates and returns the area to the statement that called the function

```
Private Function GetCircleArea(ByVal radius As Double) As Double
    Return 3.141593 * radius ^ 2
End Function
```

Steps

1. Click a blank line in the Code Editor window. The blank line can be anywhere between the Public Class and End Class clauses. However, it must be outside any other Sub or Function procedure.
2. Type the Function procedure header and then press Enter. The Code Editor automatically enters the End Function clause for you.

Figure 8-17 How to create a Function procedure

You can invoke a function from one or more places in an application's code. You invoke a function that you create in exactly the same way as you invoke one of Visual Basic's built-in functions, such as the InputBox function. You do this by including the function's name and arguments (if any) in a statement. The number, data type, and position of the arguments should agree with the number, data type, and position of the function's parameters. In most cases, the statement that invokes a function assigns the function's return value to a variable. However, it also may use the return value in a calculation or simply display the return value. Figure 8-18 shows examples of invoking the GetCircleArea function from Figure 8-17. The `GetCircleArea(circleRadius)` entry in each example invokes the function, passing it the value stored in the `circleRadius` variable.

HOW TO Invoke a Function Procedure

Example 1 – assign the return value to a variable

```
circleArea = GetCircleArea(circleRadius)
```

Example 2 – use the return value in a calculation

```
halveArea = GetCircleArea(circleRadius) / 2
```

the assignment statement divides the function's return value by 2 and assigns the result to the `halveArea` variable

Example 3 – display the return value

```
areaLabel.Text = GetCircleArea(circleRadius).ToString
```

Figure 8-18 How to invoke a Function procedure

The Circle Area Calculator Application

The Circle Area Calculator application uses the GetCircleArea function from the previous section to calculate the area of a circle. It then displays the area in the areaLabel. Figure 8-19 shows a sample run of the application, and Figure 8-20 shows a portion of the application's code.

If you want to experiment with the Circle Area Calculator application, open the solution contained in the Try It 5! folder.

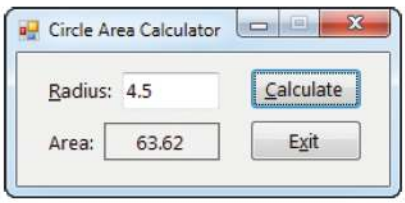

Figure 8-19 Sample run of the Circle Area Calculator application

GetCircleArea function

```
Private Function GetCircleArea(ByVal radius As Double) As Double
    Dim area As Double
    area = 3.141593 * radius ^ 2
    Return area
End Function

Private Sub calcButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles calcButton.Click
    Dim circleRadius As Double
    Dim circleArea As Double

    Double.TryParse(radiusTextBox.Text, circleRadius)
    circleArea = GetCircleArea(circleRadius)
    areaLabel.Text = circleArea.ToString("N2")
End Sub
```

invokes the GetCircleArea function and assigns the return value to the `circleArea` variable

Figure 8-20 Partial code for the Circle Area Calculator application

The `circleArea = GetCircleArea(circleRadius)` statement in the calcButton's Click event procedure calls the GetCircleArea function, passing it the value stored in the `circleRadius` variable. The computer stores the value in the `radius` variable listed in the function's header. After processing the header, the computer processes the statements contained in the function. The Dim statement in the function creates and initializes the Double `area` variable. Next, the assignment statement calculates the circle's area, using the radius value passed to the function, and assigns the result to the `area` variable. The `Return area` statement then returns the contents of the `area` variable to the statement that invoked the function. That statement is the `circleArea = GetCircleArea(circleRadius)` statement in the calcButton's Click event procedure. The End Function clause is processed next and ends the function. At this point, the computer removes the `radius` and `area` variables from its internal memory.

To review what you learned about Sub and Function procedures, view the Ch08SubAndFunction video.

The `circleArea = GetCircleArea(circleRadius)` statement in the calcButton's Click event procedure assigns the function's return value to the `circleArea` variable. The next statement in the event procedure displays the contents of the `circleArea` variable in the areaLabel, as shown earlier in Figure 8-19. The End Sub clause is processed next and ends the procedure. The computer then removes the `circleRadius` and `circleArea` variables from its internal memory.

The last concepts covered in this chapter are how to convert Object variables to a different data type and how to use a timer control. You will use the concepts in this chapter's programming tutorials.

Converting Object Variables

Every event procedure contains the `ByVal sender As Object` code in its procedure header. The code creates a variable named `sender` and assigns the Object data type to it. As you learned in Chapter 3, an Object variable can store any type of data. In this case, the `sender` variable contains the address of the object that raised the event. Unlike variables declared using the String and numeric data types, variables declared using the Object data type do not have a set of properties. This is because there are no common attributes for all of the different types of data that can be stored in an Object variable. If you need

to access the properties of the object whose address is stored in the `sender` variable, you must convert the variable to the appropriate data type. The process of converting a variable from one data type to another is sometimes referred to as **type casting** or, more simply, as **casting**. You can cast a variable from the Object data type to a different data type using the **TryCast operator**. The operator's syntax is shown in Figure 8-21, along with examples of using the operator.

HOW TO Use the TryCast Operator

Syntax
TryCast(*object, dataType***)**

Example 1
```
Dim thisTextBox As TextBox
thisTextBox = TryCast(sender, TextBox)
thisTextBox.SelectAll()
```
The assignment statement casts (converts) the `sender` variable to the TextBox data type and assigns the result to a TextBox variable named `thisTextBox`. The SelectAll method selects (highlights) the contents of the text box whose address is stored in the `thisTextBox` variable.

Example 2
```
Dim clickedButton As Button
clickedButton = TryCast(sender, Button)
MessageBox.Show(clickedButton.Text)
```
The assignment statement casts (converts) the `sender` variable to the Button data type and assigns the result to a Button variable named `clickedButton`. The MessageBox.Show method displays the Text property of the button whose address is stored in the `clickedButton` variable.

Figure 8-21 How to use the TryCast operator

The Full Name Application

The Full Name application allows the user to enter a first name and a last name. It then concatenates both names and displays the result in the interface. When a text box in the interface receives the focus, the application's code selects the text box's existing text. Figure 8-22 shows a sample run of the application, and Figure 8-23 shows two ways of writing the code for each text box's Enter event. As shown in Version 1, you can enter the appropriate SelectAll method in each text box's Enter event procedure. Or, as Version 2 indicates, you can associate each text box's Enter event with the SelectText procedure and then enter the TryCast operator and SelectAll method in that procedure.

If you want to experiment with the Full Name application, open the solution contained in the Try It 6! folder.

Figure 8-22 Sample run of the Full Name application

selects the firstTextBox's text

selects the lastTextBox's text

assigns the text box that raised the Enter event to the thisTextBox variable, and then selects the text box's text

```
Version 1
Private Sub firstTextBox_Enter(ByVal sender As Object,
ByVal e As System.EventArgs) Handles firstTextBox.Enter
      ' selects the existing text

      firstTextBox.SelectAll()
End Sub

Private Sub lastTextBox_Enter(ByVal sender As Object,
ByVal e As System.EventArgs) Handles lastTextBox.Enter
      ' selects the existing text

      lastTextBox.SelectAll()
End Sub

Version 2
Private Sub SelectText(ByVal sender As Object,
                       ByVal e As System.EventArgs
                       ) Handles firstTextBox.Enter,
                       lastTextBox.Enter
      ' selects the existing text

      Dim thisTextBox As TextBox
      thisTextBox = TryCast(sender, TextBox)
      thisTextBox.SelectAll()
End Sub
```

Figure 8-23 Two ways of writing the Enter event procedures for both text boxes

Using a Timer Control

The game application in Programming Tutorial 2 requires you to use a timer control. You instantiate a timer control using the Timer tool, which is located in the Components section of the toolbox. Timer controls are not placed on the form; instead, they are placed in the component tray. Recall that the component tray stores controls that do not appear in the user interface during run time. The purpose of a **timer control** is to process code at one or more regular intervals. The length of each interval is specified in milliseconds and entered in the timer's **Interval property**. As you learned in Chapter 6, a millisecond is 1/1000 of a second. In other words, there are 1000 milliseconds in a second. The timer's state—either running or stopped—is determined by its Enabled property. When its Enabled property is set to True, the timer is running; when it is set to False, the timer is stopped. If the timer is running, its **Tick event** occurs each time an interval has elapsed. Each time the Tick event occurs, the code contained in the Tick event procedure is processed. If the timer is stopped, the Tick event does not occur; as a result, the code entered in the Tick event procedure is not processed.

The Timer Example application uses a timer to blink a label 10 times. Figure 8-24 shows the application's interface, and Figure 8-25 shows the code entered in two procedures: the Blink button's Click event procedure and the timer's Tick event procedure.

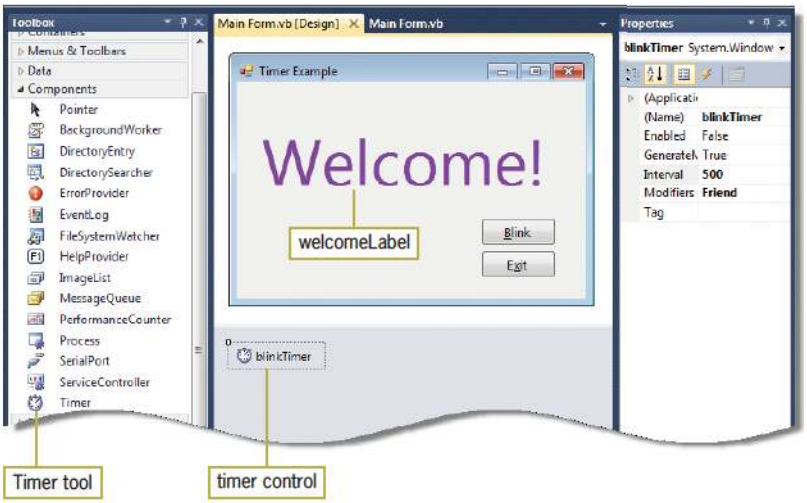

Figure 8-24 Interface for the Timer Example application

If you want to experiment with the Timer Example application, open the solution contained in the Try It 7! folder.

```
Private Sub blinkButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles blinkButton.Click
     ' turns the timer on
     blinkTimer.Enabled = True
End Sub

Private Sub blinkTimer_Tick(ByVal sender As Object,
ByVal e As System.EventArgs) Handles blinkTimer.Tick
     ' blinks the welcomeLabel 10 times

     Static numBlinks As Integer = 1

     If numBlinks < 11 Then
          If welcomeLabel.Visible = True Then
               welcomeLabel.Visible = False
          Else
               welcomeLabel.Visible = True
               numBlinks += 1
          End If
     Else
          blinkTimer.Enabled = False
     End If
End Sub
```

blinkButton Click event procedure

blinkTimer Tick event procedure

Figure 8-25 Code entered in two of the application's procedures

Mini-Quiz 8-3

The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following Handles clauses associates a procedure with the TextChanged events of the nameTextBox and salesTextBox?

   a. `Handles nameTextBox_TextChanged, salesTextBox_TextChanged`

   b. `Handles nameTextBox.TextChanged AndAlso salesTextBox.TextChanged`

c. `Handles nameTextBox-TextChanged, salesTextBox-TextChanged`

d. `Handles nameTextBox.TextChanged, salesTextBox.TextChanged`

2. Which of the following procedure headers indicates that the procedure returns a Decimal number?

   a. `Private Function Calc() As Decimal`

   b. `Private Sub Calc() As Decimal`

   c. `Private Function Calc(Decimal)`

   d. both a and b

3. A function can return _____.

   a. zero or more values

   b. one or more values

   c. one value only

4. Which of the following converts the `sender` parameter to the Label data type, assigning the result to a Label variable named `currentLabel`?

   a. `TryCast(sender, Label, currentLabel)`

   b. `currentLabel = TryCast(sender, Label)`

   c. `currentLabel = TryCast(Label, sender)`

   d. `sender = TryCast(currentLabel, Label)`

5. To turn on a timer, you set its _____ property to True.

   a. Enabled

   b. Running

   c. Start

   d. none of the above

You have completed the concepts section of Chapter 8. The Programming Tutorial section is next.

PROGRAMMING TUTORIAL 1

Coding the Tri-County Electricity Application

In this tutorial, you code an application for the Tri-County Electricity Company. Figures 8-26 and 8-27 show the application's TOE chart and MainForm, respectively. The MainForm contains a group box, two radio buttons, two text boxes, six labels, and two buttons. The interface allows the user to enter three items of data: the rate code, previous meter reading, and current meter reading. When the user clicks the Calculate button, the button's Click event procedure should verify that the current meter reading is greater than or equal to the previous meter reading. If it is, the application should both calculate and display the number of electrical units used during the month and the total charge. The total charge is based on the number of units used and the rate code. Residential customers are charged $0.09 per unit, with a minimum charge of $17.65. Commercial customers are charged $0.11 per unit, with a minimum charge of $21.75. If the current meter reading is less than the previous meter reading, the application should display an appropriate message.

Task	Object	Event
End the application	exitButton	Click
Get the rate code	residentialRadioButton, commercialRadioButton	None
Get and display the current meter reading and previous meter reading	currentTextBox, previousTextBox	None
Select the existing text	currentTextBox, previousTextBox	Enter
Allow only numbers and the Backspace key	currentTextBox, previousTextBox	KeyPress
Clear usageLabel and totalLabel	currentTextBox, previousTextBox	TextChanged
	residentialRadioButton, commercialRadioButton	Click
1. Determine whether the current meter reading is greater than or equal to the previous meter reading 2. If necessary, calculate the monthly usage and total charge and then display the results in the usageLabel and totalLabel 3. If necessary, display "The current reading must be greater than or equal to the previous reading." message in a message box	calcButton	Click
Display monthly usage (from calcButton)	usageLabel	None
Display total charge (from calcButton)	totalLabel	None

Figure 8-26 TOE chart for the Tri-County Electricity application

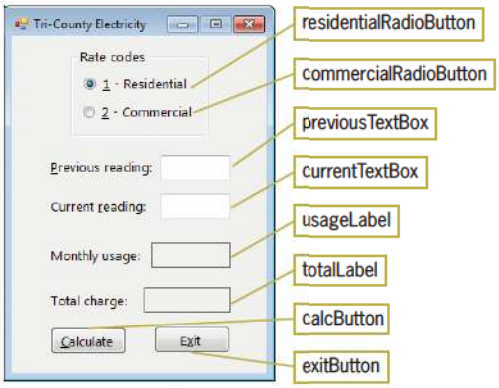

Figure 8-27 MainForm in the Tri-County Electricity application

Coding the Application

Included in the data files for this book is a partially completed Tri-County Electricity application. To complete the application, you just need to code it. According to the application's TOE chart, the Click event procedures for the two buttons and two radio buttons need to be coded. Each text box's Enter, KeyPress, and TextChanged event procedures also need to be coded.

To open the Tri-County Electricity application:

1. Start Visual Studio or the Express Edition of Visual Basic. If necessary, open the Solution Explorer window.
2. Open the **Tri-County Electricity Solution** (**Tri-County Electricity Solution.sln**) file, which is contained in the VbReloaded2010\Chap08\Tri-County Electricity Solution folder. Open the designer window (if necessary), and then auto-hide the Solution Explorer window.
3. Open the Code Editor window. Notice that the exitButton's Click event procedure has already been coded for you. In addition, the Option statements have already been entered in the General Declarations section.
4. In the comments that appear in the General Declarations section, replace <your name> and <current date> with your name and the current date.

First, you will code the Enter event procedures for both text boxes. The procedures should select the text box's existing text when the text box receives the focus. You can code each text box's Enter event procedure individually. Or, you can enter the code in a Sub procedure and then associate both Enter events with the procedure; this is the method you will use.

To code each text box's Enter event procedure:

1. Open the code template for the currentTextBox's Enter event procedure. In the procedure header, change `currentTextBox_Enter` to **SelectText**.
2. Change the Handles clause in the procedure header to the following:

   Handles currentTextBox.Enter, previousTextBox.Enter
3. In the blank line below the procedure header, type **' select existing text** and press **Enter** twice.

4. Now, enter the following code:

   Dim thisTextBox As TextBox
   thisTextBox = TryCast(sender, TextBox)
   thisTextBox.SelectAll()

5. Save the solution and then start the application. Type **12** in the Previous reading box, press **Tab**, and then type **200** in the Current reading box. Press **Tab** four times to move the focus to the Previous reading box; doing this selects the text entered in the box. Press **Tab** again to move the focus to the Current reading box, which selects that box's text.

6. Click the **Exit** button.

Next, you will code the KeyPress event procedures for both text boxes. Each procedure should allow its text box to accept only numbers and the Backspace key. Here again, you can code each KeyPress event procedure separately. Or, you can enter the code in a Sub procedure that is associated with both KeyPress events.

To code each text box's KeyPress event procedure:

1. Open the code template for the currentTextBox's KeyPress event procedure. In the procedure header, change `currentTextBox_KeyPress` to **CancelKeys**.

2. Change the Handles clause in the procedure header to the following:

   Handles currentTextBox.KeyPress, previousTextBox.KeyPress

3. In the blank line below the procedure header, type **' allow only numbers and the Backspace** and press **Enter** twice.

4. Now, enter the following code (the Code Editor will automatically enter the End If clause for you):

   ```
   If (e.KeyChar < "0" OrElse e.KeyChar > "9") AndAlso
       e.KeyChar <> ControlChars.Back Then
       e.Handled = True
   ```

5. Save the solution and then start the application. On your own, test the KeyPress event procedures. You can do this by trying to enter characters other than numbers into each text box. Also be sure to verify that the text boxes accept numbers and the Backspace key.

6. Click the **Exit** button.

Next, you will code the TextChanged event procedures for both text boxes and the Click event procedures for both radio buttons. The procedures should clear the contents of the usageLabel and totalLabel when either of those events occurs. Here too, you can code each TextChanged event procedure individually. Or, you can enter the code in a Sub procedure that is associated with both TextChanged events.

To code each text box's TextChanged event procedure and each radio button's Click event procedure:

1. Open the code template for the currentTextBox's TextChanged event procedure. In the procedure header, change `currentTextBox_TextChanged` to **ClearLabels**.

2. Change the Handles clause in the procedure header to the following:

   Handles currentTextBox.TextChanged, previousTextBox.TextChanged, residentialRadioButton.Click, commercialRadioButton.Click

3. In the blank line below the procedure header, type **' clear calculated value** and press **Enter** twice.

4. Now, enter the following code:

   usageLabel.Text = String.Empty
   totalLabel.Text = String.Empty

5. Save the solution. You won't be able to test the ClearLabels procedure until the calcButton's Click event procedure is coded.

Completing the Application's Code

Figure 8-28 shows the pseudocode for the calcButton's Click event procedure. As the pseudocode indicates, the procedure will use a Sub procedure to calculate the total charge for residential customers, and a function to calculate the total charge for commercial customers. A Sub procedure and function were chosen, rather than two Sub procedures or two function procedures, simply to allow you to practice with both types of procedures. The pseudocode for both procedures is included in Figure 8-28.

```
calcButton Click event procedure
1. assign user input (previous reading and current reading) to variables
2. if current reading is greater than or equal to previous reading
        usage = current reading – previous reading

        if the residentialRadioButton is selected
                call CalcResidentialTotal Sub procedure to calculate
                the total charge; pass the procedure the usage value
                and a variable in which to store the total charge
        else
                total charge = call the GetCommercialTotal function; pass
                the function the usage value
        end if
        display the usage and total charge in usageLabel and totalLabel
   else
        display message in a message box
   end if

CalcResidentialTotal procedure (receives the usage value and the address of a variable in which to store the total charge)
1. declare constants to store the unit charge (.09) and the minimum fee (17.65)
2. total charge = usage value * unit charge
3. if total charge is less than the minimum fee
        total charge = minimum fee
   end if

GetCommercialTotal function (receives the usage value)
1. declare constants to store the unit charge (.12) and the minimum fee (21.75)
2. declare a variable to store the total charge
3. total charge = usage value * unit charge
4. if total charge is less than the minimum fee
        total charge = minimum fee
   end if
5. return total charge
```

Figure 8-28 Pseudocode for three procedures in the Tri-County Electricity application

You will code the CalcResidentialTotal Sub procedure first. According to its pseudocode, the procedure will receive two items of data from the statement that calls it: the usage value and the address of a variable where it can place the total charge after it has been calculated. The procedure will store the information it receives in two parameters named `units` and `charge`.

To code the CalcResidentialTotal Sub procedure:

1. Scroll to the top of the Code Editor window. Click the **blank line** below the `Public Class MainForm` clause and then press **Enter** to insert a new blank line. Enter the following procedure header and comments. Press **Enter** twice after typing the second comment. (When you press Enter after typing the procedure header, the Code Editor automatically enters the End Sub clause for you.)

```
Private Sub CalcResidentialTotal(ByVal units As Integer,
                                 ByRef charge As Double)

    ' calculates the total charge for a
    ' residential customer
```

2. Step 1 in the pseudocode is to declare constants to store the unit charge and minimum fee. Enter the following declaration statements. Press **Enter** twice after typing the second declaration statement.

```
Const UnitCharge As Double = .09
Const MinFee As Double = 17.65
```

3. Step 2 is to calculate the total charge. Enter the following assignment statement:

```
    charge = units * UnitCharge
```

4. Step 3 is a single-alternative selection structure that compares the total charge with the minimum fee. If the total charge is less than the minimum fee, the selection structure's true path assigns the minimum fee as the total charge. Enter the following code (the Code Editor will automatically enter the End If clause for you):

```
If charge < MinFee Then
    charge = MinFee
```

5. Save the solution.

Next, you will code the GetCommercialTotal function. According to its pseudocode, the function will receive one item of data from the statement that calls it: the usage value. The function will store the usage value in a parameter named `units`.

To code the GetCommercialTotal function:

1. Click immediately after the letter **b** in the CalcResidentialTotal procedure's End Sub clause, and then press **Enter** twice.

2. Enter the following function header and comments. Press **Enter** twice after typing the second comment. (When you press Enter after typing the function header, the Code Editor automatically enters the End Function clause for you. Don't be concerned about the jagged line

that appears below the clause; it will disappear when you enter the Return statement.)

```
Private Function GetCommercialTotal(ByVal units As Integer) As Double
    ' calculates the total charge for a
    ' commercial customer
```

3. Step 1 in the pseudocode is to declare constants to store the unit charge and minimum fee. Enter the following declaration statements:

```
Const UnitCharge As Double = .12
Const MinFee As Double = 21.75
```

4. Step 2 is to declare a variable to store the total charge. Type the following declaration statement and then press **Enter** twice:

```
Dim charge As Double
```

5. Step 3 is to calculate the total charge. Enter the following assignment statement:

```
    charge = units * UnitCharge
```

6. Step 4 is a single-alternative selection structure that compares the total charge with the minimum fee. If the total charge is less than the minimum fee, the selection structure's true path assigns the minimum fee as the total charge. Enter the following code (the Code Editor will automatically enter the End If clause for you):

```
If charge < MinFee Then
    charge = MinFee
```

7. The final step in the pseudocode is to return the total charge. Click immediately after the letter **f** in the End If clause and then press **Enter** twice. Enter the following statement:

```
Return charge
```

8. Save the solution.

The last procedure you need to code is the calcButton's Click event procedure.

To code the calcButton's Click event procedure and then test the code:

1. Open the code template for the calcButton's Click event procedure. Type **' displays the monthly usage and total charge** and press **Enter** twice.

2. The procedure will use three Integer variables to store the previous reading, current reading, and usage amount. It also will use a Double variable to store the total charge. Enter the following declaration statements. Press **Enter** twice after typing the last declaration statement.

```
Dim previous As Integer
Dim current As Integer
Dim usage As Integer
Dim total As Double
```

3. The first step in the pseudocode is to assign the user input to variables. Enter the following TryParse methods. Press **Enter** twice after typing the last TryParse method.

```
Integer.TryParse(previousTextBox.Text, previous)
Integer.TryParse(currentTextBox.Text, current)
```

4. Step 2 in the pseudocode is a dual-alternative selection structure that determines whether the current reading is greater than or equal to the previous reading. If the selection structure's condition evaluates to True, the first instruction in the true path calculates the usage amount. Enter the additional code shown in Figure 8-29, and then position the insertion point as shown in the figure.

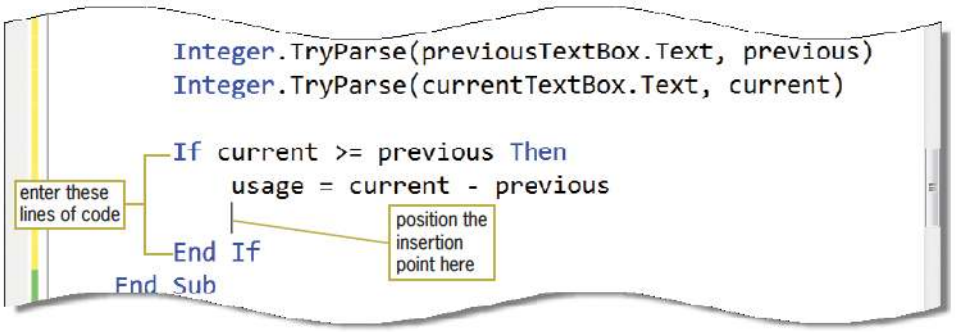

Figure 8-29 Additional code entered in the calcButton's Click event procedure

5. The next instruction in the true path is a nested dual-alternative selection structure. The nested selection structure's condition determines whether the residentialRadioButton is selected in the interface. If the condition evaluates to True, the nested selection structure's true path calls the CalcResidentialTotal Sub procedure, passing it the `usage` and `total` variables. Recall that the `usage` variable is passed *by value*, whereas the `total` variable is passed *by reference*. If the condition evaluates to False, on the other hand, the nested selection structure's false path invokes the GetCommercialTotal function, passing it the `usage` variable *by value*, and then assigns the return value to the `total` variable. Enter the nested selection structure shown in Figure 8-30, and then position the insertion point as shown in the figure.

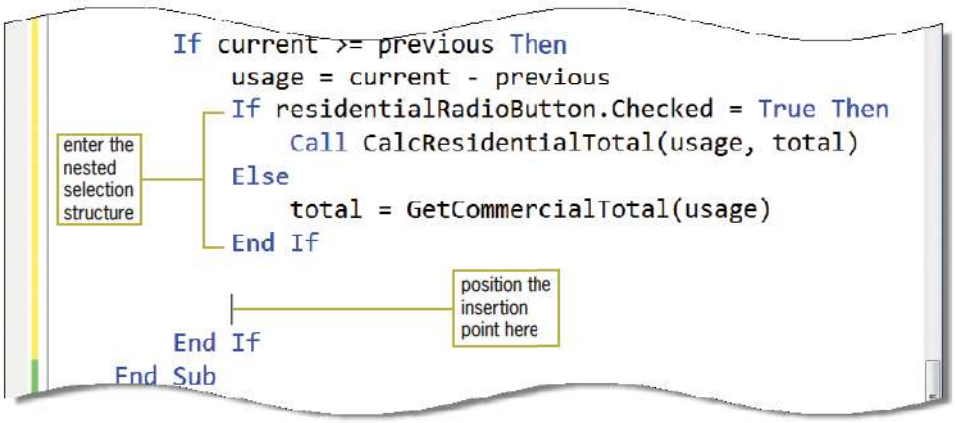

Figure 8-30 Nested selection structure entered in the calcButton's Click event procedure

6. The last instruction in the outer selection structure's true path is to display the usage and total charge in the usageLabel and totalLabel, respectively. Enter the following two assignment statements:

usageLabel.Text = usage.ToString("N0")
totalLabel.Text = total.ToString("C2")

7. Finally, you need to enter the appropriate code in the outer selection structure's false path. The code should display a message in a message box. Enter the additional code shown in Figure 8-31.

enter these lines of code

```
            usageLabel.Text = usage.ToString("N0")
            totalLabel.Text = total.ToString("C2")
        Else
            MessageBox.Show("The current reading must " &
                "be greater than or equal to the " &
                "previous reading.",
                "Tri-County Electricity",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information)

        End If
    End Sub
```

Figure 8-31 Outer selection structure's false path entered in the calcButton's Click event procedure

8. Save the solution and then start the application. Type **2500** in the Previous reading box. Press **Tab** and then type **3500** in the Current reading box. Click the **Calculate** button. The monthly usage and total charge appear in the interface, as shown in Figure 8-32.

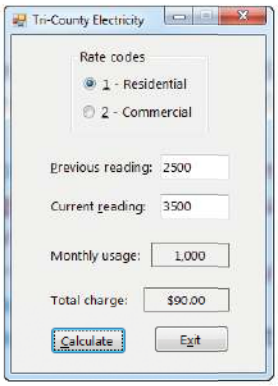

Figure 8-32 Interface showing the monthly usage and total charge

9. Click the **2 - Commercial** radio button. The radio button's Click event procedure clears the contents of the Monthly usage and Total charge boxes. Click the **Calculate** button. The interface shows that the monthly usage and total charge are 1,000 and $120.00, respectively.

10. Press **Tab** three times to place the focus in the Previous reading box. Type **4**. The previousTextBox's TextChanged event procedure clears the contents of the Monthly usage and Total charge boxes. Click the **Calculate** button.

11. Press **Tab** four times to place the focus in the Current reading box. Type **2**. The currentTextBox's TextChanged event procedure clears the contents of the Monthly usage and Total charge boxes. Click the **Calculate** button.

12. The message "The current reading must be greater than or equal to the previous reading." appears in a message box. Close the message box.

13. Click the **Exit** button to end the application. Close the Code Editor window, and then close the solution. Figure 8-33 shows the application's code.

```
1  ' Project name:          Tri-County Electricity Project
2  ' Project purpose:       Displays the monthly electric bill
3  ' Created/revised by:    <your name> on <current date>
4
5  Option Explicit On
6  Option Strict On
7  Option Infer Off
8
9  Public Class MainForm
10
11   Private Sub CalcResidentialTotal(ByVal units As Integer,
12                                    ByRef charge As Double)
13      ' calculates the total charge for a
14      ' residential customer
15
16      Const UnitCharge As Double = 0.09
17      Const MinFee As Double = 17.65
18
19      charge = units * UnitCharge
20      If charge < MinFee Then
21          charge = MinFee
22      End If
23   End Sub
24
25   Private Function GetCommercialTotal(ByVal units As Integer)
     As Double
26      ' calculates the total charge for a
27      ' commercial customer
28
29      Const UnitCharge As Double = 0.12
30      Const MinFee As Double = 21.75
31      Dim charge As Double
32
33      charge = units * UnitCharge
34      If charge < MinFee Then
35          charge = MinFee
36      End If
37
38      Return charge
39
40   End Function
```

Figure 8-33 Code for the Tri-County Electricity application *(continues)*

(continued)

```
41  Private Sub exitButton_Click(ByVal sender As Object,
    ByVal e As System.EventArgs) Handles exitButton.Click
42      Me.Close()
43  End Sub
44
45  Private Sub SelectText(ByVal sender As Object,
    ByVal e As System.EventArgs) Handles currentTextBox.Enter,
    previousTextBox.Enter
46    ' select existing text
47
48    Dim thisTextBox As TextBox
49    thisTextBox = TryCast(sender, TextBox)
50    thisTextBox.SelectAll()
51
52  End Sub
53
54  Private Sub CancelKeys(ByVal sender As Object,
    ByVal e As System.Windows.Forms.KeyPressEventArgs
    ) Handles currentTextBox.KeyPress,
    previousTextBox.KeyPress
55    ' allow only numbers and the Backspace
56
57    If (e.KeyChar < "0" OrElse e.KeyChar > "9") AndAlso
58        e.KeyChar <> ControlChars.Back Then
59        e.Handled = True
60
61    End If
62  End Sub
63
64  Private Sub ClearLabels(ByVal sender As Object,
    ByVal e As System.EventArgs
    ) Handles currentTextBox.TextChanged,
    previousTextBox.TextChanged,
65  residentialRadioButton.Click,
    commercialRadioButton.Click
66    ' clear calculated values
67
68    usageLabel.Text = String.Empty
69    totalLabel.Text = String.Empty
70
71  End Sub
72
73  Private Sub calcButton_Click(ByVal sender As Object,
    ByVal e As System.EventArgs) Handles calcButton.Click
74    ' displays the monthly usage and total charge
75
76    Dim previous As Integer
77    Dim current As Integer
78    Dim usage As Integer
79    Dim total As Double
80
81    Integer.TryParse(previousTextBox.Text, previous)
82    Integer.TryParse(currentTextBox.Text, current)
83
84    If current >= previous Then
85        usage = current - previous
86        If residentialRadioButton.Checked = True Then
```

Figure 8-33 Code for the Tri-County Electricity application *(continues)*

(continued)

```
87              Call CalcResidentialTotal(usage, total)
88          Else
89              total = GetCommercialTotal(usage)
90          End If
91
92          usageLabel.Text = usage.ToString("N0")
93          totalLabel.Text = total.ToString("C2")
94      Else
95      MessageBox.Show("The current reading must " &
96              "be greater than or equal to the " &
97              "previous reading.",
98              "Tri-County Electricity",
99              MessageBoxButtons.OK,
100             MessageBoxIcon.Information)
101
102     End If
103   End Sub
104 End Class
```

Figure 8-33 Code for the Tri-County Electricity application

PROGRAMMING TUTORIAL 2

Coding the Concentration Game Application

In this tutorial, you code an application that simulates a game called Concentration. The game board contains 16 labels. Scattered among the labels are eight pairs of matching words that are hidden from view. The user begins by clicking one of the labels to reveal a word. He or she then clicks another label to reveal another word. If the two words match, the words remain on the screen. If the words do not match, they are hidden once again. The game is over when all of the matching words are revealed. The user can start a new game by clicking the New Game button in the interface. In each game, the words will appear in different locations on the game board. This is accomplished using an independent Sub procedure that generates random numbers and then uses the random numbers to shuffle the words. The application's TOE chart and MainForm are shown in Figures 8-34 and 8-35, respectively.

Task	Object	Event
1. Fill the list box with 8 pairs of matching words 2. Call a procedure to shuffle the words in the wordListBox	MainForm	Load
End the application	exitButton	Click
1. Clear the label controls and then enable them 2. Reset the counter, which is used by the 16 labels, to 0 3. Call a procedure to shuffle the words in the wordListBox	newButton	Click
1. Enable the boardTableLayoutPanel 2. Disable the matchTimer	matchTimer	Tick
1. Clear the words from the chosen labels 2. Enable the boardTableLayoutPanel 3. Disable the noMatchTimer	noMatchTimer	Tick

Figure 8-34 TOE chart for the Concentration Game application *(continues)*

(continued)

Task	Object	Event
1. Use a counter to keep track of whether this is the first or second label clicked 2. If this is the first label clicked, display a word from the wordListBox in the label 3. If this is the second label clicked, disable the boardTableLayoutPanel, display a word from the wordListBox in the label, and then compare both words 4. If both words match, disable both labels and then turn on the matchTimer 5. If both words do not match, turn on the noMatchTimer 6. Reset the counter, which is used by the 16 labels, to 0	16 labels	Click
Store the 16 words	wordListBox	None
Display the game board	boardTableLayoutPanel	None

Figure 8-34 TOE chart for the Concentration Game application

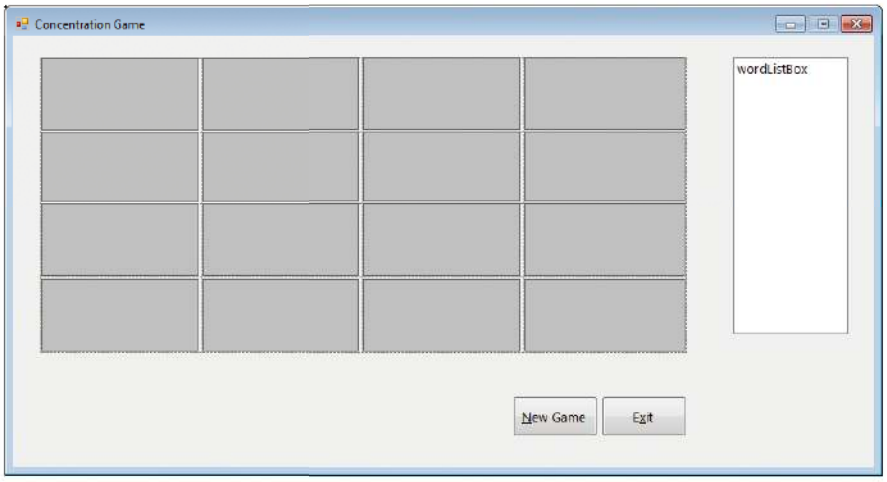

Figure 8-35 MainForm for the Concentration Game application

Coding the Concentration Game Application

Included in the data files for this book is a partially completed Concentration Game application. To complete the application, you just need to code it. According to the application's TOE chart, the MainForm's Load event procedure and the Click event procedures for the exitButton, newButton, and 16 labels need to be coded. You also need to code the Tick event procedures for the two timers. (If you need help while coding the application, you can look ahead to Figure 8-45.)

To begin coding the application:

1. Start Visual Studio or the Express Edition of Visual Basic. If necessary, open the Solution Explorer and Properties windows.
2. Open the **Concentration Game Solution** (**Concentration Game Solution.sln**) file, which is contained in the VbReloaded2010\Chap08\Concentration Game Solution folder. Open the designer

window (if necessary), and then auto-hide the Solution Explorer window. The MainForm contains a table layout panel, 16 labels, two buttons, and a list box. The component tray contains two timers.

3. On your own, click **each of the labels** in the table layout panel, one at a time. Notice that the TabIndex values in the Properties window range from 0 through 15. The TabIndex values will be used to access the appropriate word in the wordListBox, whose indexes also range from 0 through 15. Auto-hide the Properties window.
4. Open the Code Editor window, which already contains some of the application's code.
5. In the comments that appear in the General Declarations section, replace <your name> and <current date> with your name and the current date.

First, you will complete the MainForm's Load event procedure. The procedure is responsible for filling the wordListBox with eight pairs of matching words and then reordering the words. The list box provides the words that will appear on the game board. The procedure's pseudocode is shown in Figure 8-36.

```
MainForm Load event procedure
1. fill the wordListBox with 8 pairs of matching words
2. call the ShuffleWords procedure to reorder the words in the wordListBox
```

Figure 8-36 Pseudocode for the MainForm's Load event procedure

To complete the MainForm's Load event procedure:

1. Scroll down the Code Editor window (if necessary) to view the code already entered in the MainForm's Load event procedure. Notice that the first eight statements in the procedure add eight unique words to the wordListBox control, and the last eight statements duplicate the words in the control.
2. Save the solution and then start the application. The Load event procedure adds the 16 words to the list box, as shown in Figure 8-37. The words appear in the order in which they are added in the Load event procedure.

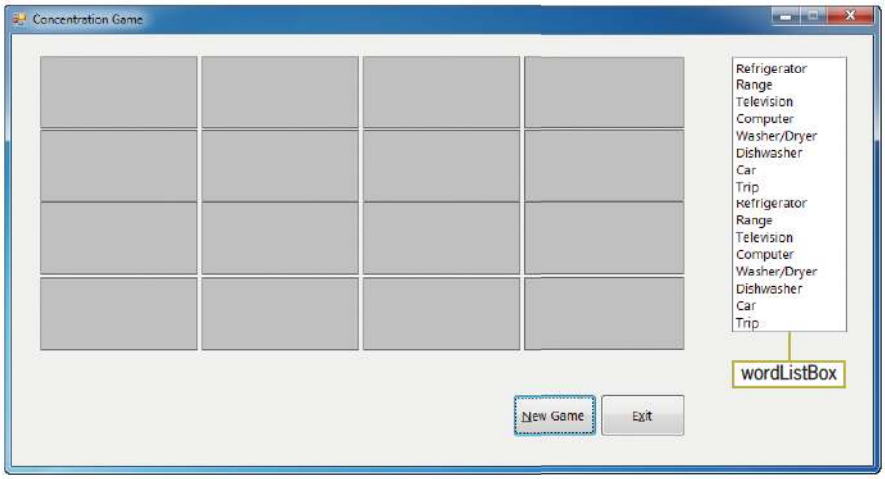

Figure 8-37 Sixteen words added to the wordListBox

3. Click the **Exit** button to end the application.
4. To complete the Load event procedure, you just need to enter a statement to call the ShuffleWords procedure, which will reorder (or shuffle) the words in the wordListBox. If you do not shuffle the words, they will appear in the exact same location on the game board each time the application is started. Shuffling the words makes the game more challenging, because the user will never be sure exactly where each word will appear on the game board. The ShuffleWords procedure will be a Sub procedure, because it will not need to return a value. The procedure will not be passed any data when it is invoked. Click the **blank line** above the End Sub clause in the Load event procedure, and then enter the appropriate Call statement. (Do not be concerned about the jagged line that appears below ShuffleWords in the Call statement. The line will disappear when you create the procedure in the next section.)

Coding the ShuffleWords Procedure

The ShuffleWords procedure is responsible for reordering the words in the wordListBox. Reordering the words will ensure that most of the words appear in different locations in each game. An easy way to reorder a list of words is to swap one word with another word. For example, you can swap the word that appears at the top of the list with the word that appears in the middle of the list. In this application, you will use random numbers to select the positions of the two words to be swapped. You will perform the swap 20 times to ensure that the words are sufficiently reordered. The procedure's pseudocode is shown in Figure 8-38.

```
ShuffleWords procedure
1.  repeat 20 times
        generate two random numbers from 0 through 15
        use the random numbers to swap words in the wordListBox
    end repeat
```

Figure 8-38 Pseudocode for the ShuffleWords procedure

To code the ShuffleWords procedure and then test the procedure:

1. Click the **blank line** immediately above the MainForm's Load event procedure. Type **Private Sub ShuffleWords()** and press **Enter**. The Code Editor enters the procedure footer (`End Sub`) for you. Notice that the jagged line no longer appears below the ShuffleWords name in the Load event procedure.
2. Type **' shuffles the words in the wordListBox** and press **Enter** twice.
3. The ShuffleWords procedure will use four variables named `randGen`, `index1`, `index2`, and `temp`. The `randGen` variable will represent the pseudo-random number generator in the procedure. The `index1` and `index2` variables will store two random integers from 0 through 15. Each integer corresponds to the index of a word in the wordListBox.

The `temp` variable will be used during the swapping process. Enter the following Dim statements. Press **Enter** twice after typing the last Dim statement.

```
Dim randGen As New Random
Dim index1 As Integer
Dim index2 As Integer
Dim temp As String
```

4. The first step in the pseudocode is a loop that repeats its instructions 20 times. Enter the following For clause:

```
For counter As Integer = 1 to 20
```

5. Change the Next clause to **Next counter**.

6. The first instruction in the loop is to generate two random numbers from 0 through 15. Click the **blank line** below the For clause, and then enter the following comment and assignment statements:

```
' generate two random numbers
index1 = randGen.Next(0, 16)
index2 = randGen.Next(0, 16)
```

7. The second instruction in the loop is to use the random numbers to swap the words in the wordListBox. You learned how to swap the contents of two variables in Chapter 4. You can use a similar process to swap two words in the wordListBox. The `index1` variable contains the index of the first word you want to swap. You begin by storing that word in the `temp` variable. Enter the following comment and assignment statement:

```
' swap two words
temp = wordListBox.Items(index1).ToString
```

8. Next, you replace the word located in the `index1` position in the wordListBox with the word located in the `index2` position. Enter the following assignment statement:

```
wordListBox.Items(index1) =
        wordListBox.Items(index2)
```

9. Finally, you replace the word located in the `index2` position in the wordListBox with the word stored in the `temp` variable. On your own, enter the appropriate assignment statement.

10. Save the solution and then start the application. The 16 words appear in the wordListBox. This time, however, they do not appear in the order in which they are entered in the Load event procedure. Instead, they appear in a random order. Click the **Exit** button to end the application.

Coding the Labels' Click Event Procedures

Next, you will code the Click event procedures for the 16 labels in the interface. Each label is associated with a word in the list box. The first label is associated with the first word, the second label with the second word, and so on. When the user clicks a label, the label's Click event procedure will access

the appropriate word in the wordListBox and then display the word in the label. For example, if the user clicks the first label on the game board, the Click event procedure will assign the first word in the list box to the label's Text property. After the user selects two labels, the procedure will determine whether the labels contain matching words. If the words match, they will remain visible in their respective labels. If the words do not match, the user will be given a short amount of time to memorize the location of the words before the words are hidden again. The pseudocode for the labels' Click event procedures is shown in Figure 8-39.

```
16 Labels' Click event procedure
1. add 1 to the selection counter, which keeps track of whether this is the first or
   second label selected on the game board
2. if this is the first label selected
        assign the current label's TabIndex property to an Integer variable named index1

        use the index1 variable to access the appropriate word in the wordListBox, and
        then display the word in the current label

   else (which means it is the second label selected)
        disable the game board to prevent the user from making another selection

        assign the current label's TabIndex property to an Integer variable named index2

        use the index2 variable to access the appropriate word in the wordListBox, and
        then display the word in the current label

        if the first label and second label contain the same word
                disable both labels on the game board
                turn the matchTimer on

        else
                turn the noMatchTimer on
        end if
        reset the selection counter to 0
   end if
```

Figure 8-39 Pseudocode for the 16 labels' Click event procedures

To code the Click event procedures for the 16 labels:

1. Locate the TestForMatch procedure in the Code Editor window. The Handles clause indicates that the procedure will be processed when the Click event occurs for any of the 16 labels.
2. The TestForMatch procedure will use two Integer variables named `index1` and `index2` to store the TabIndex property values associated with the two labels clicked by the user. The procedure will use the values to access the corresponding word in the wordListBox. For example, if the user clicks the Label1 control, which is located in the upper-left corner of the game board, the procedure will assign the control's TabIndex value—in this case, 0—to the `index1` variable. It then will use the value in the `index1` variable to access the appropriate word in the list box. The appropriate word is the one whose index value matches the TabIndex value. Click the **blank line** above the End Sub clause in the TestForMatch procedure, and then enter

the following Dim statements. Press **Enter** twice after typing the last Dim statement.

```
Dim index1 As Integer
Dim index2 As Integer
```

3. The procedure also will use three class-level variables named `selectionCounter`, `firstLabel`, and `secondLabel`. The variables need to be class-level variables, because they will be used by more than one procedure in the application. The `selectionCounter` variable will keep track of whether the user has clicked one or two labels. The `firstLabel` and `secondLabel` variables will keep track of the labels the user clicked. Click the **blank line** below the Public Class MainForm clause and then press **Enter** to insert another blank line. Enter the following Private statements:

```
Private selectionCounter As Integer
Private firstLabel As Label
Private secondLabel As Label
```

4. The first step in the pseudocode shown in Figure 8-39 is to add the number 1 to the selection counter. Click the **blank line** above the End Sub clause in the TestForMatch procedure. Enter the following comment and assignment statement. Press **Enter** twice after typing the assignment statement.

```
' update the selection counter
selectionCounter += 1
```

5. The next step is a dual-alternative selection structure that determines whether this is the first label control selected. Type the following comments and If clause:

```
' determine whether this is the first
' or second selection
If selectionCounter = 1 Then
```

6. If this is the first of two labels selected on the game board, the selection structure's true path should assign the label's TabIndex value to the `index1` variable. First, however, you will need to use the `sender` parameter to determine the label that was clicked. Recall that the parameter contains the address of the object that raised the event. Enter an assignment statement that uses the TryCast method to convert the `sender` parameter to the Label data type, and then assigns the result to the `firstLabel` variable. (Remember that if you need help, you can look ahead to Figure 8-45.)

7. Now, enter a statement that assigns the label's TabIndex property value to the `index1` variable.

8. Next, enter a statement that uses the `index1` variable to access the appropriate word in the wordListBox. The appropriate word is the one whose index matches the value contained in the `index1` variable. Assign the word to the label's Text property.

9. You have finished coding the selection structure's true path; you will code its false path next. Type **Else** and press **Tab**. Type **' second label selected** and press **Enter**.

10. If this is the second of two labels selected on the game board, the procedure will need to compare the contents of both labels before the user makes the next selection. Therefore, you will disable the game board, temporarily. Enter an assignment statement that changes the boardTableLayoutPanel control's Enabled property to **False**.

11. Next, enter an assignment statement that uses the TryCast method to convert the `sender` parameter to the Label data type, and then assigns the result to the `secondLabel` variable.

12. Now, enter a statement that assigns the label's TabIndex property value to the `index2` variable.

13. Next, enter a statement that uses the `index2` variable to access the appropriate word in the wordListBox. The appropriate word is the one whose index matches the value contained in the `index2` variable. Assign the word to the label's Text property.

14. The next instruction in the false path is a nested dual-alternative selection structure whose condition compares the contents of both labels. If both labels contain the same word, the nested selection structure's true path will disable the labels to prevent them from responding if the user inadvertently clicks them again. It also will turn on the matchTimer. If the labels do not contain the same word, the nested selection structure's false path will turn on the noMatchTimer. Enter the following comment and selection structure:

```
' compare words in both labels
If firstLabel.Text = secondLabel.Text Then
    firstLabel.Enabled = False
    secondLabel.Enabled = False
    matchTimer.Enabled = True
Else
    noMatchTimer.Enabled = True
End If
```

15. Click immediately after the letter **f** in the nested selection structure's End If clause, and then press **Enter** twice.

16. The last instruction in the pseudocode (shown earlier in Figure 8-39) is to reset the selection counter to 0. Recall that the selection counter keeps track of whether the user has clicked one or two labels. Type **'reset the selection counter** and press **Enter**, then enter an assignment statement to assign the number 0 to the `selectionCounter` variable.

17. Save the solution.

Coding Each Timer's Tick Event Procedure

Figure 8-40 shows the pseudocode for the matchTimer's Tick event procedure, which performs two tasks. First, it enables the game board so the user can make another selection. Second, it turns off the matchTimer. Turning off the timer stops the timer's Tick event and prevents its code from being processed again. The matchTimer's Tick event procedure will not be processed

again until the timer is turned back on, which happens when the user locates a matching pair of words on the game board.

matchTimer Tick event procedure
1. enable the game board
2. turn the matchTimer off

Figure 8-40 Pseudocode for the matchTimer's Tick event procedure

To code the matchTimer's Tick event procedure:

1. Locate the matchTimer's Tick event procedure in the Code Editor window.
2. Click the **blank line** above the procedure's End Sub clause. Enter an assignment statement to enable the boardTableLayoutPanel.
3. Next, enter an assignment statement to disable the matchTimer.

Figure 8-41 shows the pseudocode for the noMatchTimer's Tick event procedure, which performs three tasks. The first task clears the contents of the labels whose addresses are stored in the `firstLabel` and `secondLabel` variables. The second task enables the game board so the user can make another selection. The third task turns off the noMatchTimer to prevent the timer's Tick event from occurring and prevent its code from being processed. The noMatchTimer's Tick event procedure will not be processed again until the timer is turned back on, which happens when the two labels selected by the user contain different words.

noMatchTimer Tick event procedure
1. clear the contents of the labels associated with the firstLabel and secondLabel variables
2. enable the game board
3. turn the noMatchTimer off

Figure 8-41 Pseudocode for the noMatchTimer's Tick event procedure

To code the noMatchTimer's Tick event procedure:

1. Locate the noMatchTimer's Tick event procedure in the Code Editor window.
2. Click the **blank line** above the procedure's End Sub clause.
3. Enter two assignment statements to clear the Text properties of the labels associated with the `firstLabel` and `secondLabel` variables.
4. Enter an assignment statement to enable the boardTableLayoutPanel.
5. Next, enter an assignment statement to disable the noMatchTimer.
6. Save the solution.

Coding the New Game Button's Click Event Procedure

The last procedure you need to code is the newButton's Click event procedure. The procedure's pseudocode is shown in Figure 8-42. The first two steps have already been coded for you in the Code Editor window.

newButton Click event procedure
1. clear the contents of the 16 labels
2. enable the 16 labels
3. reset the selection counter to 0
4. call the ShuffleWords procedure to reorder the words in the wordListBox

Figure 8-42 Pseudocode for the newButton's Click event procedure

To complete the newButton's Click event procedure:

1. Locate the newButton's Click event procedure in the Code Editor window. Click the **blank line** above the procedure's End Sub clause.
2. Enter an assignment statement to assign the number 0 to the `selectionCounter` variable.
3. Enter a statement to call the ShuffleWords procedure.
4. Save the solution.

Testing the Concentration Game Application

In this section, you will test the application to verify that it is working correctly.

To test the Concentration Game application:

1. Start the application. Click the **label in the upper-left corner of the game board**. The TestForMatch procedure (which is associated with the label's Click event) assigns the first word in the wordListBox to the label's Text property. See Figure 8-43. Recall that the ShuffleWords procedure uses random numbers to reorder the list of words in the list box. Therefore, the first word in your list box, as well as the word in the Label1 control, might be different from the one shown in the figure.

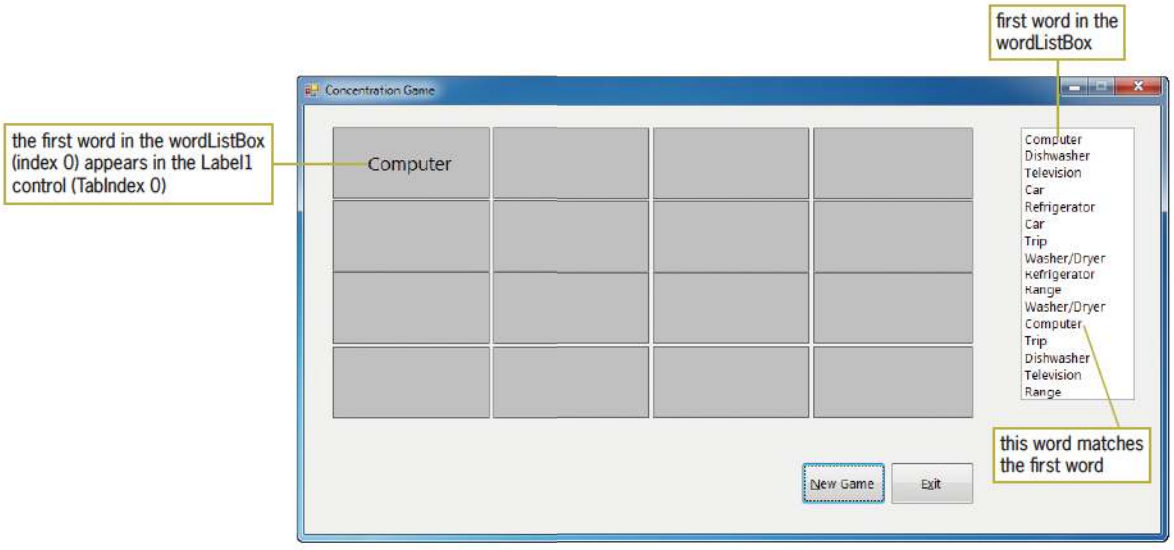

Figure 8-43 First word appears in the label on the game board

2. First, you will test the code that handles two matching words. To do this, you will need to find the word in the list box that matches the first

word, and then click its associated label on the game board. Count down the list of words in the list box, stopping when you reach the word that matches the first word in your list box. In Figure 8-43, the word that matches the first word (Computer) is the twelfth word in the list box.

3. Now count each label, from left to right, beginning with the first row on the game board. Stop counting when you reach the label whose number is the same as in the previous step. In Figure 8-43, for example, you would stop counting when you reached the twelfth label, which is located in the fourth column of the third row. Click the **label associated with the matching word**. See Figure 8-44.

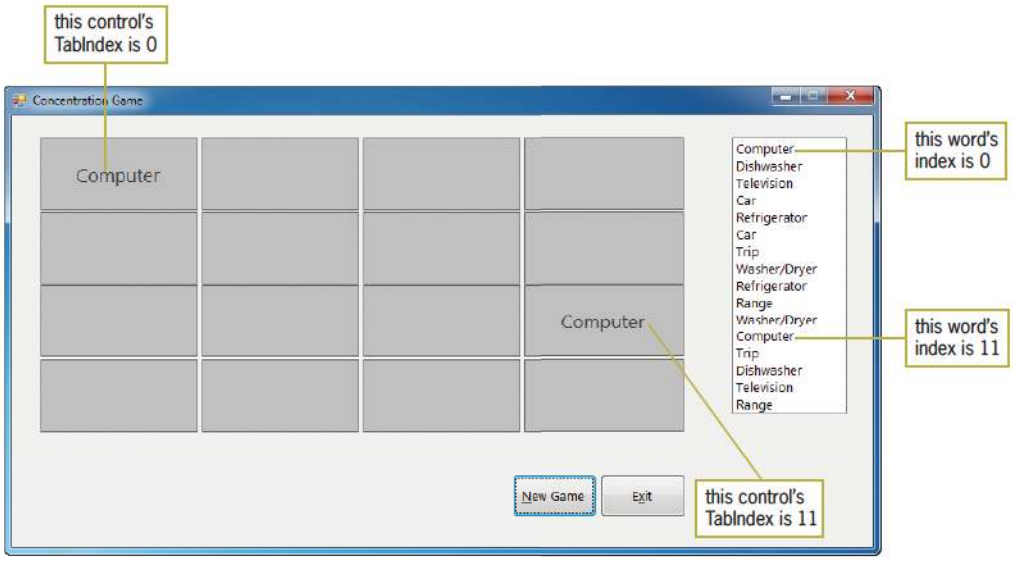

Figure 8-44 Both labels contain the same word

4. Click the **Label1 control** again. Nothing happens because the TestForMatch procedure disables the label when its matching word is found.

5. Now, you will test the code that handles two words that do not match. Click a **blank label control on the game board**, and then click **another blank label control**. However, be sure that the second label's word is not the same as the first label's word. Because both words are not the same, they are hidden after a short time.

6. Finally, verify that the code entered in the newButton's Click event procedure works correctly. Click the **New Game** button. The button's Click event procedure clears the contents of the label controls and also enables them. In addition, it resets the selection counter to 0 and calls the ShuffleWords procedure to reorder the words in the list box.

7. On your own, test the application several more times. When you are finished, click the **Exit** button to end the application.

8. Now that you know that the application works correctly, you can resize the form to hide the list box. Close the Code Editor window. Unlock the controls on the form and then drag the form's right border so that it hides the list box. Lock the controls on the form.

9. Save the solution, and then close the solution. Figure 8-45 shows the code for the Concentration Game application.

```
1  ' Project name:          Concentration Game Project
2  ' Project purpose:       Simulates the Concentration game,
3  '                        where a player tries to find
4  '                        matching pairs of words
5  ' Created/revised by:    <your name> on <current date>
6
7  Option Explicit On
8  Option Strict On
9  Option Infer Off
10
11 Public Class MainForm
12
13 Private selectionCounter As Integer
14 Private firstLabel As Label
15 Private secondLabel As Label
16
17   Private Sub ShuffleWords()
18     ' shuffles the words in the wordListBox
19
20     Dim randGen As New Random
21     Dim index1 As Integer
22     Dim index2 As Integer
23     Dim temp As String
24
25     For counter As Integer = 1 To 20
26          ' generate two random numbers
27          index1 = randGen.Next(0, 16)
28          index2 = randGen.Next(0, 16)
29          'swap two words
30          temp = wordListBox.Items(index1).ToString
31          wordListBox.Items(index1) =
32                wordListBox.Items(index2)
33          wordListBox.Items(index2) = temp
34
35     Next counter
36   End Sub
37   Private Sub MainForm_Load(ByVal sender As Object,
     ByVal e As System.EventArgs) Handles Me.Load
38     ' fills the list box with 8 pairs of matching
39     ' words, then calls a procedure to shuffle
40     ' the words
41
42     wordListBox.Items.Add("Refrigerator")
43     wordListBox.Items.Add("Range")
44     wordListBox.Items.Add("Television")
45     wordListBox.Items.Add("Computer")
46     wordListBox.Items.Add("Washer/Dryer")
47     wordListBox.Items.Add("Dishwasher")
48     wordListBox.Items.Add("Car")
49     wordListBox.Items.Add("Trip")
50     wordListBox.Items.Add("Refrigerator")
51     wordListBox.Items.Add("Range")
52     wordListBox.Items.Add("Television")
53     wordListBox.Items.Add("Computer")
54     wordListBox.Items.Add("Washer/Dryer")
55     wordListBox.Items.Add("Dishwasher")
56     wordListBox.Items.Add("Car")
57     wordListBox.Items.Add("Trip")
58
```

Figure 8-45 Code for the Concentration Game application *(continues)*

(continued)

```
    Call ShuffleWords()

End Sub

Private Sub exitButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles exitButton.Click
    Me.Close()
End Sub

Private Sub newButton_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles newButton.Click
    ' removes any words from the label controls, then
    ' enables the label controls, then resets the
    ' selection counter, and then calls a procedure
    ' to shuffle the words

    Label1.Text = String.Empty
    Label2.Text = String.Empty
    Label3.Text = String.Empty
    Label4.Text = String.Empty
    Label5.Text = String.Empty
    Label6.Text = String.Empty
    Label7.Text = String.Empty
    Label8.Text = String.Empty
    Label9.Text = String.Empty
    Label10.Text = String.Empty
    Label11.Text = String.Empty
    Label12.Text = String.Empty
    Label13.Text = String.Empty
    Label14.Text = String.Empty
    Label15.Text = String.Empty
    Label16.Text = String.Empty

    Label1.Enabled = True
    Label2.Enabled = True
    Label3.Enabled = True
    Label4.Enabled = True
    Label5.Enabled = True
    Label6.Enabled = True
    Label7.Enabled = True
    Label8.Enabled = True
    Label9.Enabled = True
    Label10.Enabled = True
    Label11.Enabled = True
    Label12.Enabled = True
    Label13.Enabled = True
    Label14.Enabled = True
    Label15.Enabled = True
    Label16.Enabled = True

    selectionCounter = 0
    Call ShuffleWords()

End Sub

Private Sub TestForMatch(ByVal sender As Object,
ByVal e As System.EventArgs) Handles Label1.Click,
```

Figure 8-45 Code for the Concentration Game application *(continues)*

(continued)

```
113   Label2.Click, Label3.Click, Label4.Click, Label5.Click,
      Label6.Click, Label7.Click,
114   Label8.Click, Label9.Click, Label10.Click, Label11.Click,
      Label12.Click, Label13.Click,
115   Label14.Click, Label15.Click, Label16.Click
116     ' displays the appropriate words and determines
117     ' whether the user selected a matching pair
118
119     Dim index1 As Integer
120     Dim index2 As Integer
121
122     ' update the selection counter
123     selectionCounter += 1
124
125     ' determine whether this is the first
126     ' or second selection
127     If selectionCounter = 1 Then
128         firstLabel = TryCast(sender, Label)
129         index1 = firstLabel.TabIndex
130         firstLabel.Text = wordListBox.Items(index1).ToString
131     Else    ' second label selected
132         boardTableLayoutPanel.Enabled = False
133         secondLabel = TryCast(sender, Label)
134         index2 = secondLabel.TabIndex
135         secondLabel.Text = wordListBox.Items(index2).ToString
136         ' compare words in both labels
137         If firstLabel.Text = secondLabel.Text Then
138            firstLabel.Enabled = False
139            secondLabel.Enabled = False
140            matchTimer.Enabled = True
141         Else
142            noMatchTimer.Enabled = True
143         End If
144
145         ' reset the selection counter
146         selectionCounter = 0
147
148     End If
149   End Sub
150
151   Private Sub matchTimer_Tick(ByVal sender As Object,
      ByVal e As System.EventArgs) Handles matchTimer.Tick
152     ' when the two words match, the game board is
153     ' enabled and the timer is turned off
154
155     boardTableLayoutPanel.Enabled = True
156     matchTimer.Enabled = False
157
158   End Sub
159
160   Private Sub noMatchTimer_Tick(ByVal sender As Object,
      ByVal e As System.EventArgs) Handles noMatchTimer.Tick
161     ' when the words do not match, the words are
162     ' removed from the labels, the game board is
163     ' enabled, and the timer is turned off
164
165     firstLabel.Text = String.Empty
```

Figure 8-45 Code for the Concentration Game application *(continues)*

(continued)

```
166      secondLabel.Text = String.Empty
167      boardTableLayoutPanel.Enabled = True
168      noMatchTimer.Enabled = False
169
170    End Sub
171 End Class
```

Figure 8-45 Code for the Concentration Game application

PROGRAMMING EXAMPLE

Rainfall Calculator Application

Create an interface that provides a text box for the user to enter monthly rainfall amounts. The application should calculate and display two amounts: the total rainfall and the average rainfall. Use the following names for the solution, project, and form file: Rainfall Solution, Rainfall Project, and Main Form.vb. Save the application in the VbReloaded2010\Chap08 folder. See Figures 8-46 through 8-51.

Task	Object	Event
1. Use a counter and accumulator to keep keep track of the number of rainfall amounts entered and the total rainfall 2. Call a procedure to calculate the average rainfall 3. Display the total rainfall and average rainfall in totalLabel and averageLabel 4. Send the focus to the monthTextBox 5. Select the monthTextBox's existing text	calcButton	Click
End the application	exitButton	Click
Display the total rainfall amount (from calcButton)	totalLabel	None
Display the average rainfall amount (from calcButton)	averageLabel	None
Get and display the monthly rainfall amounts Select the existing text Allow numbers, period, and Backspace Clear totalLabel and averageLabel	monthTextBox	None Enter KeyPress TextChanged

Figure 8-46 TOE chart for the Rainfall Calculator application

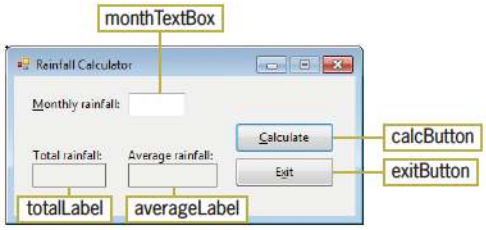

Figure 8-47 MainForm in the Rainfall Calculator application

Object	Property	Setting
MainForm	AcceptButton	calcButton
	Font	Segoe UI, 10 point
	MaximizeBox	False
	StartPosition	CenterScreen
	Text	Rainfall Calculator
totalLabel	AutoSize	False
	BorderStyle	FixedSingle
	Text	(empty)
	TextAlign	MiddleCenter
averageLabel	AutoSize	False
	BorderStyle	FixedSingle
	Text	(empty)
	TextAlign	MiddleCenter

Figure 8-48 Objects, properties, and settings

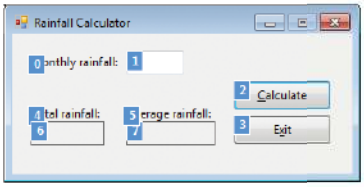

Figure 8-49 Tab order

exitButton Click event procedure
close the application

monthTextBox Enter event procedure
select the existing text

monthTextBox KeyPress event procedure
allow only numbers, the period, and the Backspace key

monthTextBox TextChanged event procedure
clear the contents of the totalLabel and averageLabel

calcButton Click event procedure
1. if the monthTextBox is not empty
   add the monthly rainfall to the total rainfall accumulator
   add 1 to the rainfall counter
   end if
2. Call the CalcAverage procedure to calculate the average rainfall; pass the procedure the rainfall counter and rainfall accumulator values and the address of a variable in which to store the average rainfall
3. send the focus to the monthTextBox
4. select the existing text in the monthTextBox

CalcAverage procedure (receives the rainfall counter and rainfall accumulator values and the address of a variable in which to store the average rainfall)
if the rainfall counter > 0
    average rainfall = rainfall accumulator / rainfall counter
else
    average rainfall = 0
end if

Figure 8-50 Pseudocode

PROGRAMMING EXAMPLE

```
1  ' Project name:          Rainfall Project
2  ' Project purpose:       Displays the total and average
3  '                        rainfall amounts
4  ' Created/revised by:    <your name> on <current date>
5
6  Option Explicit On
7  Option Strict On
8  Option Infer Off
9
10 Public Class MainForm
11
12     Private Sub CalcAverage(ByVal counter As Integer,
13                             ByVal accumulator As Decimal,
14                             ByRef avg As Decimal)
15         ' calculates the average rainfall amount
16
17         If counter > 0 Then
18             avg = accumulator / counter
19         Else
20             avg = 0
21         End If
22     End Sub
23
24     Private Sub exitButton_Click(ByVal sender As Object,
       ByVal e As System.EventArgs) Handles exitButton.Click
25         Me.Close()
26     End Sub
27
28     Private Sub calcButton_Click(ByVal sender As Object,
       ByVal e As System.EventArgs) Handles calcButton.Click
29         ' displays the total and average rainfall amount
30
31         Static rainCounter As Integer
32         Static rainAccum As Decimal
33         Dim monthRain As Decimal
34         Dim avgRain As Decimal
35
36         If monthTextBox.Text <> String.Empty Then
37             Decimal.TryParse(monthTextBox.Text, monthRain)
38             ' update the accumulator and counter
39             rainAccum += monthRain
40             rainCounter += 1
41         End If
42
43         ' calculate the average
44         Call CalcAverage(rainCounter, rainAccum, avgRain)
45
46         totalLabel.Text = rainAccum.ToString("N2")
47         averageLabel.Text = avgRain.ToString("N2")
48         monthTextBox.Focus()
49         monthTextBox.SelectAll()
50     End Sub
51
52     Private Sub monthTextBox_Enter(ByVal sender As Object,
       ByVal e As System.EventArgs) Handles monthTextBox.Enter
53         monthTextBox.SelectAll()
54     End Sub
55
```

Figure 8-51 Code *(continues)*

(continued)

```
56      Private Sub monthTextBox_KeyPress(ByVal sender As Object,
        ByVal e As System.Windows.Forms.KeyPressEventArgs
        ) Handles monthTextBox.KeyPress
57          ' allow numbers, period, and Backspace
58
59          If (e.KeyChar < "0" OrElse e.KeyChar > "9") AndAlso
60          e.KeyChar <> "." AndAlso e.KeyChar <> ControlChars.Back Then
61          e.Handled = True
62          End If
63      End Sub
64
65      Private Sub monthTextBox_TextChanged(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles monthTextBox.TextChanged
66          totalLabel.Text = String.Empty
67          averageLabel.Text = String.Empty
68      End Sub
69  End Class
```

Figure 8-51 Code

Summary

- The difference between a Sub procedure and a Function procedure is that a Function procedure returns a value, whereas a Sub procedure does not return a value.
- An event procedure is a Sub procedure that is associated with one or more objects and events.
- Independent Sub procedures and Function procedures are not associated with any specific object or event. The names of independent Sub procedures and Function procedures typically begin with a verb.
- Procedures allow programmers to avoid duplicating code in different parts of a program. They also allow the programmer to concentrate on one small piece of a program at a time. In addition, they allow a team of programmers to work on large and complex programs.
- You can use the Call statement to invoke an independent Sub procedure. The Call statement allows you to pass arguments to the Sub procedure.
- When calling a procedure, the number of arguments listed in the argumentList should agree with the number of parameters listed in the parameterList in the procedure header. Also, the data type and position of each argument in the argumentList should agree with the data type and position of its corresponding parameter in the parameterList.
- You can pass information to a Sub or Function procedure either *by value* or *by reference*. To pass a variable *by value*, you precede the variable's corresponding parameter with the keyword `ByVal`. To pass a variable *by reference*, you precede the variable's corresponding parameter with the keyword `ByRef`. The procedure header indicates whether a variable is being passed *by value* or *by reference*.

- When you pass a variable *by value*, only a copy of the variable's contents is passed. When you pass a variable *by reference*, the variable's address is passed.
- Variables that appear in the parameterList in a procedure header have procedure scope, which means they can be used only by the procedure.
- You can use an event procedure's Handles clause to associate the procedure with more than one object and event.
- You invoke a Function procedure, also called a function, by including its name and any arguments in a statement. Usually the statement assigns the function's return value to a variable. However, it also may use the return value in a calculation or display the return value.
- You can use the TryCast operator to convert an Object variable to a different data type.
- The purpose of a timer control is to process code at one or more specified intervals. You start a timer by setting its Enabled property to True. You stop a timer by setting its Enabled property to False. You use a timer's Interval property to specify the number of milliseconds that must elapse before the timer's Tick event occurs.

Key Terms

Call statement—the statement used to invoke an independent Sub procedure in a Visual Basic program

Casting—another term for type casting

Desk-checking—the process of manually walking through your code, using sample data; also called hand-tracing

e parameter—one of the parameters in an event procedure's header; contains additional information provided by the object that raised the event

Function procedure—a procedure that returns a value after performing its assigned task

Functions—another term for Function procedures

Hand-tracing—another term for desk-checking

Independent Sub procedure—a procedure that is not associated with any specific object or event and is processed only when invoked (called) from code

Interval property—a property of a timer control; stores the length of each interval

Parameters—the memory locations listed in a procedure header

Passing by reference—the process of passing a variable's address to a procedure

Passing by value—the process of passing a copy of a variable's value to a procedure

Return statement—the Visual Basic statement that returns a function's value to the statement that invoked the function

sender parameter—one of the parameters in an event procedure's header; contains the memory address of the object that raised the event

Sub procedure—a procedure that does not return a value after performing its assigned task

Tick event—one of the events of a timer control; occurs each time an interval has elapsed

Timer control—the control used to process code at one or more regular intervals

TryCast operator—used to convert an Object variable to a different data type

Type casting—the process of converting a variable from one data type to another; also called casting

Review Questions

1. To determine whether a variable is being passed to a procedure *by value* or *by reference*, you will need to examine _____.
   a. the Call statement
   b. the procedure header
   c. the statements entered in the procedure
   d. either a or b

2. Which of the following statements invokes the CalcArea Sub procedure, passing it two variables *by value*?
   a. `Call CalcArea(length, width)`
   b. `Call CalcArea(ByVal length, ByVal width)`
   c. `Invoke CalcArea ByVal(length, width)`
   d. `CalcArea(length, width) As Double`

3. Which of the following is a valid header for a procedure that receives an integer followed by a number with a decimal place?
   a. `Private Sub CalcFee(base As Integer, rate As Decimal)`
   b. `Private Sub CalcFee(ByRef base As Integer, ByRef rate As Decimal)`
   c. `Private Sub CalcFee(ByVal base As Integer, ByVal rate As Decimal)`
   d. none of the above

4. Which of the following indicates that the procedure should be processed when the user clicks either the firstCheckBox or the secondCheckBox?
   a.
```
Private Sub Clear(ByVal sender As Object,
ByVal e As System.EventArgs) Handles firstCheckBox.Click,
secondCheckBox.Click
```

b. `Private Sub Clear(ByVal sender As Object, ByVal e As System.EventArgs) Handles firstCheckBox_Click, secondCheckBox_Click`

c. `Private Sub Clear_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles firstCheckBox, secondCheckBox`

d. `Private Sub Clear(ByVal sender As Object, ByVal e As System.EventArgs) Handles firstCheckBox.Click AndAlso secondCheckBox.Click`

5. Which of the following is false?

a. The sequence of the arguments listed in the Call statement should agree with the sequence of the parameters listed in the receiving procedure's header.

b. The data type of each argument in the Call statement should match the data type of its corresponding parameter in the procedure header.

c. The name of each argument in the Call statement should be identical to the name of its corresponding parameter in the procedure header.

d. When you pass information to a procedure *by value*, the procedure stores the value of each item it receives in a separate memory location.

6. Which of the following instructs a function to return the contents of the `stateTax` variable?

a. `Return stateTax`

b. `Return stateTax ByVal`

c. `Return ByVal stateTax`

d. `Return ByRef stateTax`

7. Which of the following is a valid header for a procedure that receives the value stored in an Integer variable first, and the address of a Decimal variable second?

a.
```
Private Sub CalcFee(ByVal base As Integer,
  ByAdd rate As Decimal)
```

b. `Private Sub CalcFee(base As Integer, rate As Decimal)`

c.
```
Private Sub CalcFee(ByVal base As Integer,
  ByRef rate As Decimal)
```

d. none of the above

8. Which of the following is false?

a. When you pass a variable *by reference*, the receiving procedure can change its contents.

b. To pass a variable *by reference* in Visual Basic, you include the keyword `ByRef` before the variable's name in the Call statement.

c. When you pass a variable *by value*, the receiving procedure creates a procedure-level variable that it uses to store the value passed to it.

d. Unless you specify otherwise, all variables in Visual Basic are passed *by value*.

9. A Sub procedure named CalcEndingInventory is passed four Integer variables named `begin`, `sales`, `purchases`, and `ending`. The procedure should calculate the ending inventory using the beginning inventory, sales, and purchase amounts passed to the procedure. The result should be stored in the `ending` variable. Which of the following procedure headers is correct?

a.
```
Private Sub CalcEndingInventory(ByVal b As Integer,
ByVal s As Integer, ByVal p As Integer,
ByRef final As Integer)
```

b.
```
Private Sub CalcEndingInventory(ByVal b As Integer,
ByVal s As Integer, ByVal p As Integer,
ByVal final As Integer)
```

c.
```
Private Sub CalcEndingInventory(ByRef b As Integer,
ByRef s As Integer, ByRef p As Integer,
ByVal final As Integer)
```

d.
```
Private Sub CalcEndingInventory(ByRef b As Integer,
ByRef s As Integer, ByRef p As Integer,
ByRef final As Integer)
```

10. Which of the following statements should you use to call the CalcEndingInventory procedure described in Question 9?

a.
```
Call CalcEndingInventory(begin, sales,
purchases, ending)
```

b.
```
Call CalcEndingInventory(ByVal begin, ByVal sales,
ByVal purchases, ByRef ending)
```

c.
```
Call CalcEndingInventory(ByRef begin, ByRef sales,
ByRef purchases, ByRef ending)
```

d.
```
Call CalcEndingInventory(ByVal begin, ByVal sales,
ByVal purchases, ByVal ending)
```

Exercises

Pencil and Paper

INTRODUCTORY

1. Explain the difference between a Sub procedure and a function.

INTRODUCTORY

2. Explain the difference between passing a variable *by value* and passing it *by reference*.

3. Explain the difference between invoking a Sub procedure and invoking a function. INTRODUCTORY

4. Write the code for a Sub procedure that receives an integer passed to it. The procedure should divide the integer by 2 and then display the result in the numLabel. Name the procedure DivideByTwo. Then write a statement to invoke the procedure, passing it the number 120. INTRODUCTORY

5. Write the code for a Sub procedure that prompts the user to enter the name of a city and then stores the user's response in the String variable whose address is passed to the procedure. Name the procedure GetCity. Then write a statement to invoke the procedure, passing it the `city` variable. INTRODUCTORY

6. Write the code for a function that prompts the user to enter the name of a state and then returns the user's response. Name the function GetState. Then write a statement to invoke the GetState function. Display the function's return value in a message box. INTRODUCTORY

7. Write the code for a Sub procedure that receives three Double variables: the first two *by value* and the last one *by reference*. The procedure should divide the first variable by the second variable and then store the result in the third variable. Name the procedure CalcQuotient. INTRODUCTORY

8. Write the code for a function that receives a copy of the value stored in an Integer variable. The procedure should divide the value by 2 and then return the result, which may contain a decimal place. Name the function GetQuotient. Then write an appropriate statement to invoke the function, passing it the `number` variable. Assign the function's return value to the `answer` variable. INTRODUCTORY

9. Write the code for a function that receives a copy of the contents of four Integer variables. The function should calculate the average of the four integers and then return the result, which may contain a decimal place. Name the function GetAverage. Then write a statement to invoke the function, passing it the `num1`, `num2`, `num3`, and `num4` variables. Assign the function's return value to a Double variable named `average`. INTERMEDIATE

10. Write the code for a Sub procedure that receives four Integer variables: the first two *by value* and the last two *by reference*. The procedure should calculate the sum of and the difference between the two variables passed *by value*, and then store the results in the variables passed *by reference*. When calculating the difference, subtract the contents of the second variable from the contents of the first variable. Name the procedure GetSumAndDiff. Then write an appropriate statement to invoke the procedure, passing it the `first`, `second`, `sum`, and `difference` variables. INTERMEDIATE

INTERMEDIATE

11. Write the procedure header for a Sub procedure named CalculateTax. The procedure should be invoked when any of the following occurs: the rate1Button's Click event, the rate2Button's Click event, and the salesListBox's SelectedValueChanged event.

INTERMEDIATE

12. Write the statement to convert the `sender` parameter to a radio button. Assign the result to a RadioButton variable named `currentRadioButton`.

Computer

MODIFY THIS

13. In this exercise, you experiment with passing variables *by value* and *by reference*.

    a. Open the Passing Solution (Passing Solution.sln) file contained in the VbReloaded2010\Chap08\Passing Solution folder. Open the Code Editor window and review the existing code. Notice that the `myName` variable is passed *by value* to the GetName procedure. Start the application. Click the Display Name button. When prompted to enter a name, type your name and press Enter. Explain why the button's Click event procedure does not display your name in the nameLabel. Stop the application.

    b. Modify the Display Name button's Click event procedure so that it passes the `myName` variable *by reference* to the GetName procedure. Save the solution and then start the application. Click the Display Name button. When prompted to enter a name, type your name and press Enter. This time, your name appears in the nameLabel. Explain why the button's Click event procedure now works correctly. Close the solution.

MODIFY THIS

14. Open the Gross Pay Solution (Gross Pay Solution.sln) file contained in the VbReloaded2010\Chap08\Gross Pay Solution-Ex14 folder. Modify the code so that it uses one procedure to clear the contents of the grossLabel when the SelectedValueChanged event occurs for either list box. Save the solution and then start and test the application. Close the solution.

MODIFY THIS

15. If necessary, complete the Tri-County Electricity application from this chapter's Programming Tutorial 1, and then close the solution. Use Windows to make a copy of the Tri-County Electricity Solution folder. Rename the folder Tri-County Electricity Solution-Modified. Open the Tri-County Electricity Solution (Tri-County Electricity Solution.sln) file contained in the VbReloaded2010\Chap08\Tri-County Electricity Solution-Modified folder. Modify the code so that it uses a function named GetResidentialTotal (rather than the CalcResidentialTotal Sub procedure) to calculate the charge for residential customers. Also modify the code so that it uses a Sub procedure named CalcCommercialTotal (rather than the GetCommercialTotal function) to calculate the charge for commercial customers. Save the solution and then start and test the application. Close the solution.

16. If necessary, complete the Concentration Game application from this chapter's Programming Tutorial 2, and then close the solution. Use Windows to make a copy of the Concentration Game Solution folder. Rename the folder Concentration Game Solution-Color. Open the Concentration Game Solution (Concentration Game Solution.sln) file contained in the VbReloaded2010\Chap08\Concentration Game Solution-Color folder. When the user finds a matching pair of words, change the BackColor property of the corresponding labels to a different color. Be sure to return the labels to their original color when the user clicks the New Game button. Also modify the application so that it displays the message "Game Over" when the user has located all of the matching pairs. Save the solution and then start and test the application. Close the solution. MODIFY THIS

17. If necessary, complete the Rainfall Calculator application from this chapter's Programming Example, and then close the solution. Use Windows to make a copy of the Rainfall Solution folder. Rename the folder Rainfall Solution-Modified. Open the Rainfall Solution (Rainfall Solution.sln) file contained in the VbReloaded2010\Chap08\Rainfall Solution-Modified folder. Replace the CalcAverage procedure with a function named GetAverage. Save the solution and then start and test the application. Close the solution. MODIFY THIS

18. Open the Gross Pay Solution (Gross Pay Solution.sln) file contained in the VbReloaded2010\Chap08\Gross Pay Solution-Ex18 folder. Replace the CalcGrossPay procedure with a function named GetGrossPay. Save the solution and then start and test the application. Close the solution. INTRODUCTORY

19. Open the Bonus Calculator Solution (Bonus Calculator Solution.sln) file contained in the VbReloaded2010\Chap08\Bonus Calculator Solution folder. Code the application, using a Sub procedure to both calculate and display a 10% bonus. Also use a Sub procedure named ClearLabel to clear the contents of the bonusLabel when the TextChanged event occurs for either text box. In addition, code each text box's Enter event procedure. Save the solution and then start and test the application. Close the solution. INTRODUCTORY

20. Open the Car Solution (Car Solution.sln) file contained in the VbReloaded2010\Chap08\Car Solution folder. When the Click Me button is clicked, the "I WANT THIS CAR!" message should blink 14 times. Add a timer control to the application. Code the control's Tick event procedure. Save the solution and then start and test the application. Close the solution. INTRODUCTORY

21. If necessary, complete the Rainfall Calculator application from this chapter's Programming Example, and then close the solution. Use Windows to make a copy of the Rainfall Solution folder. Rename the folder Rainfall Solution-Intermediate. Open the Rainfall Solution (Rainfall Solution.sln) file contained in the VbReloaded2010\Chap08\Rainfall Solution-Intermediate folder. Modify the code so that it INTERMEDIATE

uses two functions rather than the CalcAverage procedure. One of the functions should calculate and return the total rainfall; the other should calculate and return the average rainfall. Save the solution and then start and test the application. Close the solution.

INTERMEDIATE

22. Use Windows to make a copy of the Temperature Solution folder. Rename the folder Temperature Solution-Subs. Open the Temperature Solution (Temperature Solution.sln) file contained in the VbReloaded2010\Chap08\Temperature Solution-Subs folder. Code the application so that it uses two independent Sub procedures: one to convert a temperature from Fahrenheit to Celsius, and the other to convert a temperature from Celsius to Fahrenheit. Save the solution and then start and test the application. Close the solution.

INTERMEDIATE

23. Use Windows to make a copy of the Temperature Solution folder. Rename the folder Temperature Solution-Functions. Open the Temperature Solution (Temperature Solution.sln) file contained in the VbReloaded2010\Chap08\Temperature Solution-Functions folder. Code the application so that it uses two functions: one to convert a temperature from Fahrenheit to Celsius, and the other to convert a temperature from Celsius to Fahrenheit. Save the solution and then start and test the application. Close the solution.

INTERMEDIATE

24. Open the Translator Solution (Translator Solution.sln) file contained in the VbReloaded2010\Chap08\Translator Solution folder. Code the application so that it uses functions to translate the English words into French, Spanish, or Italian. Save the solution and then start and test the application. Close the solution.

INTERMEDIATE

25. If necessary, complete the Concentration Game application from this chapter's Programming Tutorial 2, and then close the solution. Use Windows to make a copy of the Concentration Game Solution folder. Rename the folder Concentration Game Solution-Wild. Open the Concentration Game Solution (Concentration Game Solution.sln) file contained in the VbReloaded2010\Chap08\Concentration Game Solution-Wild folder. Replace the Washer/Dryer values in the list box with two Wild Card values. A Wild Card value matches any other value on the game board. The game is over either when all of the words are revealed or when two unmatched words remain on the game board. Make the appropriate modifications to the code. Save the solution and then start and test the application. Close the solution.

ADVANCED

26. If necessary, complete the Tri-County Electricity application from this chapter's Programming Tutorial 1, and then close the solution. Use Windows to make a copy of the Tri-County Electricity Solution folder. Rename the folder Tri-County Electricity Solution-Advanced1. Open the Tri-County Electricity Solution (Tri-County Electricity Solution.sln) file contained in the VbReloaded2010\Chap08\Tri-County Electricity Solution-Advanced1 folder. Replace the CalcResidentialTotal Sub procedure and the GetCommercialTotal function with a Sub procedure named CalcTotal. Modify the Calculate button's

Click event procedure so that it uses the CalcTotal procedure for both residential and commercial customers. Save the solution and then start and test the application. Close the solution.

27. If necessary, complete the Tri-County Electricity application from this chapter's Programming Tutorial 1, and then close the solution. Use Windows to make a copy of the Tri-County Electricity Solution folder. Rename the folder Tri-County Electricity Solution-Advanced2. Open the Tri-County Electricity Solution (Tri-County Electricity Solution.sln) file contained in the VbReloaded2010\Chap08\Tri-County Electricity Solution-Advanced2 folder. Replace the CalcResidentialTotal Sub procedure and the GetCommercialTotal function with a function named GetTotal. Modify the Calculate button's Click event procedure so that it uses the GetTotal function for both residential and commercial customers. Save the solution and then start and test the application. Close the solution. ADVANCED

28. If necessary, complete the Concentration Game application from this chapter's Programming Tutorial 2, and then close the solution. Use Windows to make a copy of the Concentration Game Solution folder. Rename the folder Concentration Game Solution-Counters. Open the Concentration Game Solution (Concentration Game Solution.sln) file contained in the VbReloaded2010\Chap08\Concentration Game Solution-Counters folder. Modify the application so that it displays (in two labels) the number of times the user selects a matching pair of words, and the number of times the user does not select a matching pair of words. Save the solution and then start and test the application. Close the solution. ADVANCED

29. In this exercise, you learn how to specify that one or more arguments are optional in a Call statement. Open the Optional Solution (Optional Solution.sln) file contained in the VbReloaded2010\Chap08\Optional Solution folder. DISCOVERY

    a. Open the Code Editor window and review the existing code. The calcButton's Click event procedure contains two Call statements. The first Call statement passes three variables to the CalcBonus procedure. The second call statement, however, passes only two variables to the procedure. (Do not be concerned about the jagged line that appears below the second Call statement.) Notice that the `rate` variable is omitted from the second Call statement. You indicate that the `rate` variable is optional in the Call statement by including the keyword `Optional` before the variable's corresponding parameter in the procedure header; you enter the `Optional` keyword before the `ByVal` keyword. You also assign a default value that the procedure will use for the missing parameter when the procedure is called. You assign the default value by entering the assignment operator and the default value after the parameter in the procedure header. In this case, you will assign the number .1 as the default value for the `rate` variable. (Optional parameters must be listed at the end of the procedure header.)

b. Change the `ByVal bonusRate As Double` in the procedure header appropriately. Save the solution and then start the application. Calculate the bonus for a salesperson with an "a" code, $1000 in sales, and a rate of .05. The `Call CalcBonus(sales, bonus, rate)` statement calls the CalcBonus procedure, passing it the number 1000, the address of the `bonus` variable, and the number .05. The CalcBonus procedure stores the number 1000 in the `totalSales` variable. It also assigns the name `bonusAmount` to the `bonus` variable and stores the number .05 in the `bonusRate` variable. The procedure then multiplies the contents of the `totalSales` variable (1000) by the contents of the `bonusRate` variable (.05), assigning the result (50) to the `bonusAmount` variable. The `bonusLabel.Text = bonus.ToString("C2")` statement then displays $50.00 in the bonusLabel.

c. Now calculate the bonus for a salesperson with a code of "b" and a sales amount of $2000. The `Call CalcBonus(sales, bonus)` statement calls the CalcBonus procedure, passing it the number 2000 and the address of the `bonus` variable. The CalcBonus procedure stores the number 2000 in the `totalSales` variable and assigns the name `bonusAmount` to the `bonus` variable. Because the Call statement did not supply a value for the `bonusRate` variable, the default value (.1) is assigned to the variable. The procedure then multiplies the contents of the `totalSales` variable (2000) by the contents of the `bonusRate` variable (.1), assigning the result (200) to the `bonusAmount` variable. The `bonusLabel.Text = bonus.ToString("C2")` statement then displays $200.00 in the bonusLabel. Close the solution.

SWAT THE BUGS

30. Open the Debug Solution (Debug Solution.sln) file contained in the VbReloaded2010\Chap08\Debug Solution folder. Open the Code Editor window and review the existing code. Start and then test the application. Locate and then correct any errors. When the application is working correctly, close the solution.

Case Projects

Car Shoppers Inc.

In an effort to boost sales, Car Shoppers Inc. is offering buyers a choice of either a large cash rebate or an extremely low financing rate, much lower than the rate most buyers would pay by financing the car through their local bank. Jake Miller, the manager of Car Shoppers Inc., wants you to create an application that helps buyers decide whether to take the lower financing rate from his dealership, or take the rebate and then finance the car through their local bank. Be sure to use one or more independent Sub or Function procedures in the application. (Hint: Use the Financial.Pmt method to calculate the payments.) Use the following names for the solution, project, and form file: Car Shoppers Solution, Car Shoppers

Project, and Main Form.vb. Save the solution in the VbReloaded2010\Chap08\ folder. You can create either your own interface or the one shown in Figure 8-52.

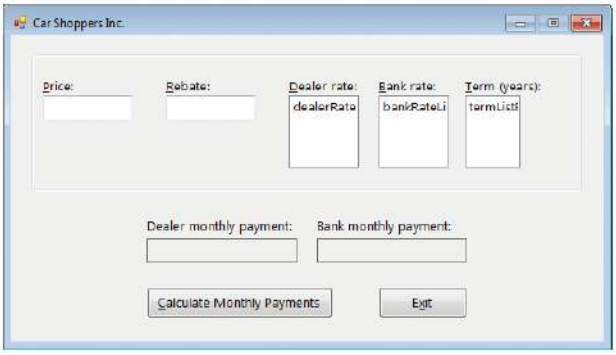

Figure 8-52 Sample interface for the Car Shoppers Inc. application

Wallpaper Warehouse

Last year, Johanna Liu opened a new wallpaper store named Wallpaper Warehouse. Johanna would like you to create an application that the sales-clerks can use to quickly calculate and display the number of single rolls of wallpaper required to cover a room. Be sure to use one or more independent Sub or Function procedures in the application. Use the following names for the solution, project, and form file: Wallpaper Warehouse Solution, Wallpaper Warehouse Project, and Main Form.vb. Save the solution in the VbReloaded2010\Chap08 folder. You can create either your own interface or the one shown in Figure 8-53.

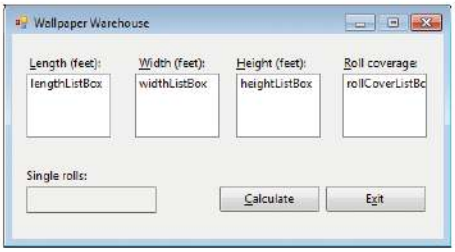

Figure 8-53 Sample interface for the Wallpaper Warehouse application

Cable Direct

Sharon Barrow, the billing supervisor at Cable Direct (a local cable company), has asked you to create an application that calculates and displays a customer's bill. The cable rates are shown in Figure 8-54. Business customers must have at least one connection. Be sure to use one or more independent Sub or Function procedures in the application. Use the following names for the solution, project, and form file: Cable Direct Solution, Cable Direct Project, and Main Form.vb. Save the solution in the VbReloaded2010\Chap08 folder. You can create either your own interface or the one shown in Figure 8-55.

Residential customers:
Processing fee: $4.50
Basic service fee: $30
Premium channels: $5 per channel
Business customers:
Processing fee: $16.50
Basic service fee: $80 for first 10 connections; $4 for each additional connection
Premium channels: $50 per channel for any number of connections

Figure 8-54 Cable Direct rates

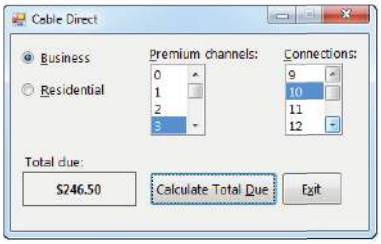

Figure 8-55 Sample run of the Cable Direct application

Harvey Industries

Khalid Patel, the payroll manager at Harvey Industries, manually calculates each employee's weekly gross pay, Social Security and Medicare (FICA) tax, federal withholding tax (FWT), and net pay—a very time-consuming process and one that is prone to mathematical errors. Mr. Patel has asked you to create an application that performs the payroll calculations both efficiently and accurately. Use the following names for the solution, project, and form file: Harvey Industries Solution, Harvey Industries Project, and Main Form.vb. Save the solution in the VbReloaded2010\Chap08 folder.

Create an appropriate interface. Employees at Harvey Industries are paid every Friday. All employees are paid on an hourly basis, with time and one-half paid for the hours worked over 40. The amount of FICA tax to deduct from an employee's weekly gross pay is calculated by multiplying the gross pay amount by 7.65%. The amount of FWT to deduct from an employee's weekly gross pay is based on the employee's filing status—either single (including head of household) or married—and his or her weekly taxable wages. You calculate the weekly taxable wages by first multiplying the number of withholding allowances by $70.19 (the value of a withholding allowance), and then subtracting the result from the weekly gross pay. For example, if your weekly gross pay is $400 and you have two withholding allowances, your weekly taxable wages are $259.62. You use the weekly taxable wages, along with the filing status and the appropriate weekly Federal Withholding Tax table, to determine the amount of FWT to withhold. The weekly tax tables for the year 2010 are shown in Figure 8-56. Be sure to use one or more independent Sub or Function procedures in the application.

FWT Tables – Weekly Payroll Period

Single person (including head of household)

If the taxable wages are: Over	But not over	Base amount	Percentage	Of excess over
	$ 116	0		
$ 116	$ 200	0	10%	$ 116
$ 200	$ 693	$ 8.40 plus	15%	$ 200
$ 693	$1,302	$ 82.35 plus	25%	$ 693
$1,302	$1,624	$ 234.60 plus	27%	$1,302
$1,624	$1,687	$ 321.54 plus	30%	$1,624
$1,687	$3,344	$ 340.44 plus	28%	$1,687
$3,344	$7,225	$ 804.40 plus	33%	$3,344
$7,225		$2,085.13 plus	35%	$7,225

Married person

If the taxable wages are: Over	But not over	Base amount	Percentage	Of excess over
	$ 264	0		
$ 264	$ 471	0	10%	$ 264
$ 471	$1,457	$ 20.70 plus	15%	$ 471
$1,457	$1,809	$ 168.80 plus	25%	$1,457
$1,809	$2,386	$ 256.60 plus	27%	$1,809
$2,386	$2,789	$ 412.39 plus	25%	$2,386
$2,789	$4,173	$ 513.14 plus	28%	$2,789
$4,173	$7,335	$ 900.66 plus	33%	$4,173
$7,335		$1,944.12 plus	35%	$7,335

Figure 8-56 Weekly FWT tables