TRAINING & REFERENCE

murach's Visual Basic 2008

Anne Boehm

(Chapter 3)

Thanks for downloading this chapter from ***Murach's Visual Basic 2008***. We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its "how-to" headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our **web site**. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on .NET development.

Thanks for your interest in our books!

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**

Contents

3

How to code and test a Windows Forms application

In the last chapter, you learned how to design a form for a Windows Forms application. In this chapter, you'll learn how to code and test a Windows Forms application. Here, the emphasis will be on the Visual Studio skills that you need for entering, editing, and testing the Visual Basic code for your applications. You'll learn how to write that code in the rest of this book.

An introduction to coding

Before you learn the mechanics of adding code to a form, it's important to understand some of the concepts behind object-oriented programming.

Introduction to object-oriented programming

Whether you know it or not, you are using *object-oriented programming* as you design a Windows form with Visual Studio's Form Designer. That's because each control on a form is an object, and the form itself is an object. These objects are derived from classes that are part of the .NET Class Library.

When you start a new project from the Windows Application template, you are actually creating a new *class* that inherits the characteristics of the Form class that's part of the .NET Class Library. Later, when you run the form, you are actually creating an *instance* of your form class, and this instance is known as an *object*.

Similarly, when you add a control to a form, you are actually adding a control object to the form. Each control is an instance of a specific class. For example, a text box control is an object that is an instance of the TextBox class. Similarly, a label control is an object that is an instance of the Label class. This process of creating an object from a class can be called *instantiation*.

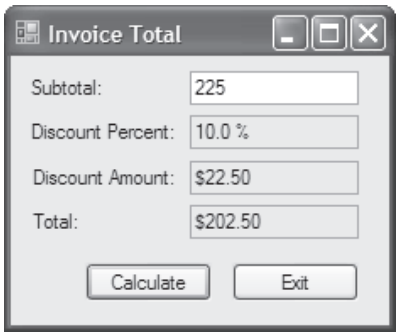
As you progress through this book, you will learn much more about classes and objects because Visual Basic is an *object-oriented language*. In chapter 11, for example, you'll learn how to use the Visual Basic language to create your own classes. At that point, you'll start to understand what's actually happening as you work with classes and objects. For now, though, you just need to get comfortable with the terms and accept the fact that a lot is going on behind the scenes as you design a form and its controls.

Figure 3-1 summarizes what I've just said about classes and objects. It also introduces you to the properties, methods, and events that are defined by classes and used by objects. As you've already seen, the *properties* of an object define the object's characteristics and data. For instance, the Name property gives a name to a control, and the Text property determines the text that is displayed within the control. In contrast, the *methods* of an object determine the operations that can be performed by the object.

An object's *events* are signals sent by the object to your application that something has happened that can be responded to. For example, a Button control object generates an event called Click if the user clicks the button. Then, your application can respond by running a Visual Basic procedure to handle the Click event.

By the way, the properties, methods, and events of an object or class are called the *members* of the object or class. You'll learn more about properties, methods, and events in the next three figures.

A form object and its ten control objects



Class and object concepts

- An *object* is a self-contained unit that combines code and data. Two examples of objects you have already worked with are forms and controls.
- A *class* is the code that defines the characteristics of an object. You can think of a class as a template for an object.
- An object is an *instance* of a class, and the process of creating an object from a class is called *instantiation*.
- More than one object instance can be created from a single class. For example, a form can have several button objects, all instantiated from the same Button class. Each is a separate object, but all share the characteristics of the Button class.

Property, method, and event concepts

- *Properties* define the characteristics of an object and the data associated with an object.
- *Methods* are the operations that an object can perform.
- *Events* are signals sent by an object to the application telling it that something has happened that can be responded to.
- Properties, methods, and events can be referred to as *members* of an object.
- If you instantiate two or more instances of the same class, all of the objects have the same properties, methods, and events. However, the values assigned to the properties can vary from one instance to another.

Objects and forms

- When you use the Form Designer, Visual Studio automatically generates Visual Basic code that creates a new class based on the Form class. Then, when you run the project, a form object is instantiated from the new class.
- When you add a control to a form, Visual Studio automatically generates Visual Basic code in the class for the form that instantiates a control object from the appropriate class and sets the control's default properties. When you move and size a control, Visual Studio automatically sets the properties that specify the location and size of the control.

Figure 3-1 Introduction to object-oriented programming

How to refer to properties, methods, and events

As you enter the code for a form in the Code Editor window, you often need to refer to the properties, methods, and events of its objects. To do that, you type the name of the object, a period (also known as a *dot operator*, or *dot*), and the name of the member. This is summarized in figure 3-2.

In some cases, you will refer to the properties and methods of a class instead of an object that's instantiated from the class. You'll see examples of that in later chapters. For now, you just need to realize that you refer to these properties and methods using the same general syntax that you use to refer to the properties and methods of an object. You enter the class name, a dot, and the property or method name.

To make it easier for you to refer to the members of an object or class, Visual Studio's IntelliSense feature displays a list of the members that are available for that class or object after you type a class or object name and a period. Then, you can highlight the entry you want by clicking on it, typing one or more letters of its name, or using the arrow keys to scroll through the list. In most cases, you can then complete the entry by pressing the Tab or Enter key.

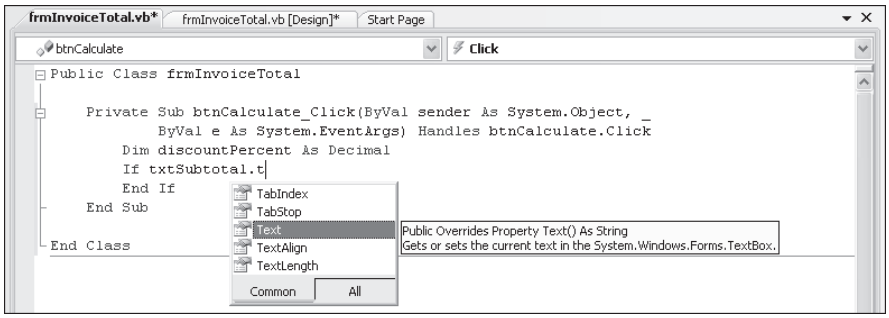
To give you an idea of how properties, methods, and events are used in code, this figure shows examples of each. In the first example for properties, code is used to set the value that's displayed for a text box to 10. In the second example, code is used to set the ReadOnly property of a text box to True. Although you can also use the Properties window to set these values, that just sets the properties at the start of the application. By using code, you can change the properties as an application is running.

In the first example for methods, the Select method of a text box is used to move the focus to that text box. In the second example, the Close method of a form is used to close the active form. In this example, the Me keyword is used instead of the name of the form. Here, Me refers to the current instance of the active form. Note also that the names of the methods are followed by parentheses. If a method requires parentheses like these, they're added automatically when you press the Enter key after entering the method name.

As you progress through this book, you'll learn how to use the methods for many types of objects, and you'll learn how to supply *arguments* within the parentheses of a method. For now, though, just try to understand that you can call a method from a class or an object.

Although you'll frequently refer to properties and methods as you code an application, you'll rarely need to refer to an event. That's because Visual Studio automatically generates the code for working with events, as you'll see later in this chapter. To help you understand the code that Visual Studio generates, however, the last example in this figure shows how you refer to an event. In this case, the code refers to the Click event of a button named btnExit.

A member list that's displayed in the Code Editor window



The syntax for referring to a member of a class or object

```
ClassName.MemberName
objectName.MemberName
```

Statements that refer to properties

`txtTotal.Text = 10`	Assigns the value 10 to the Text property of the text box named txtTotal.
`txtTotal.ReadOnly = True`	Assigns the True value to the ReadOnly property of the text box named txtTotal so the user can't change its contents.

Statements that refer to methods

`txtMonthlyInvestment.Select()`	Uses the Select method to move the focus to the text box named txtMonthlyInvestment.
`Me.Close()`	Uses the Close method to close the form that contains the statement. In this example, Me is a keyword that is used to refer to the current instance of the form class.

Code that refers to an event

`btnExit.Click`	Refers to the Click event of a button named btnExit.

How to enter member names when working in the Code Editor

- To display a list of the available members for a class or an object, type the class or object name followed by a period (called a *dot operator*, or just *dot*). Then, type one or more letters of the member name, and Visual Studio will filter the list so that only the members that start with those letters are displayed. You can also scroll through the list to select the member you want.
- Once you've selected a member, you can press the Tab key to insert it into your code, or you can press the Enter key to insert the member and start a new line of code.
- By default, all the available members are displayed in the list. To display just the common members, click the Common tab at the bottom of the list.
- If a member list isn't displayed, select the Tools➔Options command to display the Options dialog box. Then, expand the Text Editor group, select the Basic group, and check the Auto List Members and Parameter Information boxes.

Figure 3-2 How to refer to properties, methods, and events

How an application responds to events

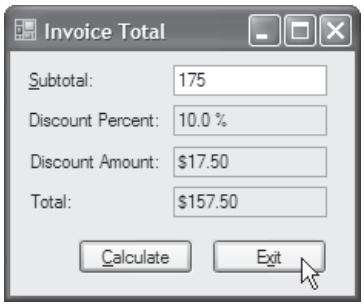
Windows Forms applications are *event-driven*. That means they work by responding to the events that occur on objects. To respond to an event, you code a procedure known as an *event handler*. In figure 3-3, you can see the event handler for the event that occurs when the user clicks the Exit button on the Invoice Total form. In this case, this event handler contains a single statement that uses the Close method to close the form.

This figure also lists some common events for controls and forms. One control event you'll respond to frequently is the Click event. This event occurs when the user clicks an object with the mouse. Similarly, the DoubleClick event occurs when the user double-clicks an object.

Although the Click and DoubleClick events are started by user actions, that's not always the case. For instance, the Enter and Leave events typically occur when the user moves the focus to or from a control, but they can also occur when code moves the focus to or from a control. Similarly, the Load event of a form occurs when a form is loaded into memory. For the first form of an application, this typically happens when the user starts the application. And the Closed event occurs when a form is closed. For the Invoice Total form in this figure, this happens when the user selects the Exit button or the Close button in the upper right corner of the form.

In addition to the events shown here, most objects have many more events that the application can respond to. For example, events occur when the user positions the mouse over an object or when the user presses or releases a key. However, you don't typically respond to those events.

Event: The user clicks the Exit button



Response: The procedure for the Click event of the Exit button is executed

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

Common control events

Event	Occurs when…
Click	…the user clicks the control.
DoubleClick	…the user double-clicks the control.
Enter	…the focus is moved to the control.
Leave	…the focus is moved from the control.

Common form events

Event	Occurs when…
Load	…the form is loaded into memory.
Closing	…the form is closing.
Closed	…the form is closed.

Concepts

- Windows Forms applications work by responding to events that occur on objects.
- To indicate how an application should respond to an event, you code an *event handler*, which is a Visual Basic procedure that handles the event.
- An event can be an action that's initiated by the user like the Click event, or it can be an action initiated by program code like the Closed event.

Figure 3-3 How an application responds to events

How to add code to a form

Now that you understand some of the concepts behind object-oriented coding, you're ready to learn how to add code to a form. Because you'll learn the essentials of the Visual Basic language in the chapters that follow, though, I won't focus on the coding details right now. Instead, I'll focus on the concepts and mechanics of adding the code to a form.

How to create an event handler for the default event of a form or control

Although you can create an event handler for any event of any object, you're most likely to create event handlers for the default event of a form or control. So that's what you'll learn to do in this chapter. Then, in chapter 6, you'll learn how to create event handlers for other events.

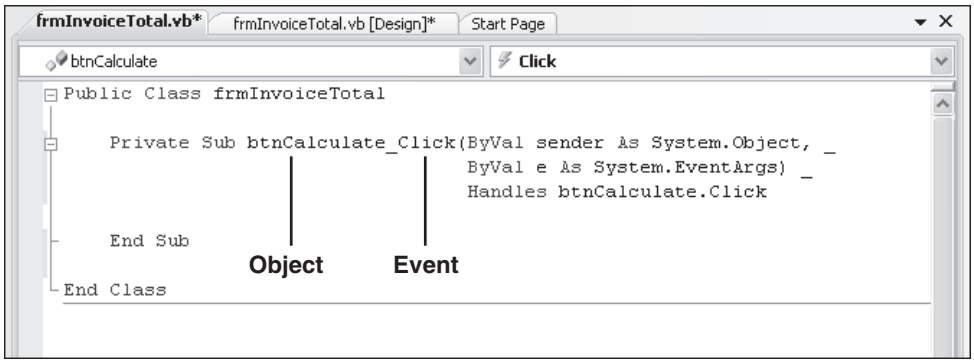
To create an event handler for the default event of a form or control, you double-click the object in the Form Designer. When you do that, Visual Studio opens the Code Editor, generates a *procedure declaration* for the default event of the object, and places the insertion point between the Sub and End Sub statements that it has generated. Then, you can enter the Visual Basic statements for the procedure between the Sub and End Sub statements.

To illustrate, figure 3-4 shows the Sub and End Sub statements that were generated when I double-clicked the Calculate button on the Invoice Total form. In the Sub statement, Visual Studio generated a *procedure name* that consists of the name of the object that the event occurred on (btnCalculate), an underscore, and the name of the event (Click).

This procedure name is followed by two arguments in parentheses that you'll learn more about later. And the arguments are followed by a Handles clause that says that the procedure is designed to handle the Click event of the button named btnCalculate. It is this clause, not the procedure name, that determines what event the procedure handles.

For now, you should avoid modifying the procedure declaration that's generated for you when you create an event handler. In chapter 6, though, you'll learn how to modify the declaration so a single procedure can provide for more than one event.

The procedure that handles the Click event of the Calculate button



How to handle the Click event of a button

1. In the Form Designer, double-click the control. This opens the Code Editor, generates the declaration for the procedure that handles the event, and places the cursor within this declaration.
2. Type the Visual Basic code between the Sub statement and the End Sub statement.
3. When you finish entering the code, you can return to the Form Designer by clicking the View Designer button in the Solution Explorer window.

How to handle the Load event for a form

- Follow the procedure shown above, but double-click the form itself.

Description

- The *procedure declaration* for the event handler that's generated when you double-click on an object in the Form Designer includes a *procedure name* that consists of the object name, an underscore, and the event name.
- The Handles clause in the procedure declaration determines what event the procedure handles using the object name, dot, event name syntax.
- In chapter 6, you'll learn how to handle events other than the default event.

Figure 3-4 How to create an event handler for the default event of a form or control

How IntelliSense helps you enter the code for a form

In figure 3-2, you saw how IntelliSense displays a list of the available members for a class or an object. IntelliSense can also help you select a type for the variables you declare, which you'll learn how to do in chapter 4. And it can help you use the correct syntax to call a procedure as shown in chapter 6 or to call a method as shown in chapter 11.

With Visual Basic 2008, IntelliSense has been improved to help you even more as you enter the basic code for an application. In particular, IntelliSense can help you enter statements and functions as well as the names of variables, objects, and classes. Figure 3-5 illustrates how this works.

The first example in this figure shows the list that IntelliSense displays when you start to enter a new line of code. Here, because I entered the letter *d*, the list includes only those items that start with that letter. As described earlier in this chapter, you can enter as many letters as you want, and Visual Studio will continue to filter the list so it contains only the items that begin with those letters. You can also scroll through the list to select an item, and you can press the Tab or Enter key to insert the item into your code.

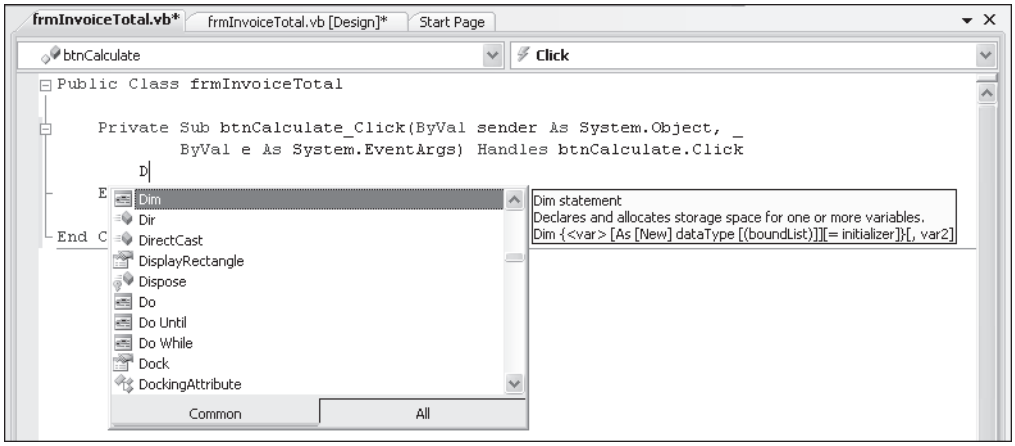
Note that when you select a keyword that begins a statement, a description of the statement is displayed in a *tool tip* along with the syntax of the statement. That can help you enter the statement correctly. In addition, as you enter the statement, you're prompted for any additional keywords that are required by the statement.

The second example in this figure shows the list that's displayed as you enter the code for an If statement. You'll learn more about this statement in chapter 5. For now, just notice that after I typed a space and the letter *t* following the If keyword, Visual Studio displayed a list of all the items that begin with the letter T. That made it easy to select the item I wanted, which in this case was the name of a control.

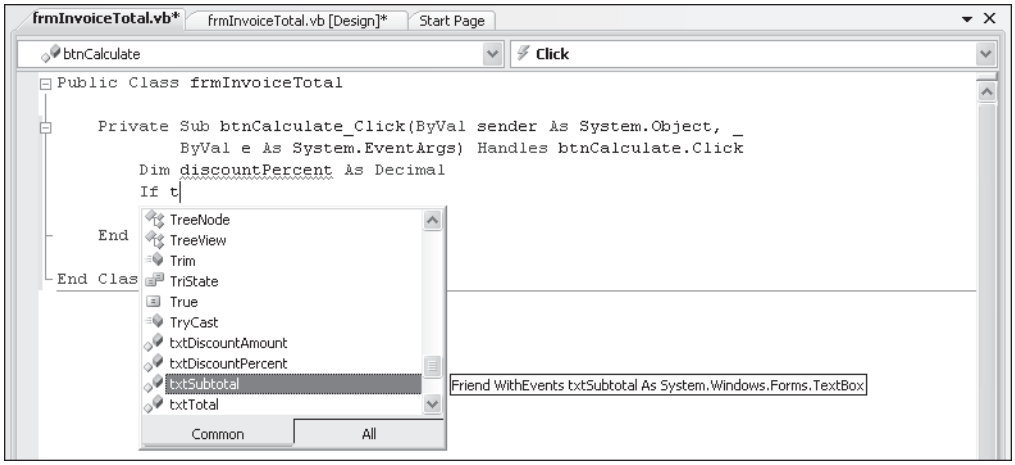
If you've used previous versions of Visual Basic, you'll appreciate these expanded IntelliSense features. For example, it's easy to forget the exact syntax of a statement or function, so the tool tip that's displayed when you select a statement or function can help refresh your memory. Similarly, it's easy to forget the names you've given to items such as controls and variables, so the list that's displayed can help you locate the appropriate name. And that can help you avoid introducing errors into your code.

Although it's not shown here, Visual Basic 2008 IntelliSense also lets you see the code that's behind a list while the list is still displayed. To do that, you simply press the Ctrl key and the list becomes semi-transparent. This eliminates the frustration a lot of programmers felt when code was hidden by an IntelliSense list in previous versions of Visual Studio.

The list that's displayed when you enter a letter at the beginning of a line of code



The list that's displayed as you enter code within a statement



Description

- The IntelliSense that's provided for Visual Basic 2008 lists keywords, functions, variables, objects, and classes as you type so you can enter them correctly.
- When you highlight an item in a list, a tool tip is displayed with information about the item.
- If you need to see the code behind a list without closing the list, press the Ctrl key. Then, the list becomes semi-transparent.

Figure 3-5 How IntelliSense helps you enter the code for a form

The event handlers for the Invoice Total form

Figure 3-6 presents the two event handlers for the Invoice Total form. The code that's shaded in this example is the code that's generated when you double-click the Calculate and Exit buttons in the Form Designer. You have to enter the rest of the code yourself.

I'll describe this code briefly here so you have a general idea of how it works. If you're new to programming, however, you may not understand the code completely until after you read the next two chapters.

The event handler for the Click event of the Calculate button calculates the discount percent, discount amount, and invoice total based on the subtotal entered by the user. Then, it displays those calculations in the appropriate text box controls. For example, if the user enters a subtotal of $1000, the discount percent will be 20%, the discount amount will be $200, and the invoice total will be $800.

In contrast, the event handler for the Click event of the Exit button contains just one statement that executes the Close method of the form. As a result, when the user clicks this button, the form is closed, and the application ends.

In addition to the code that's generated when you double-click the Calculate and Exit buttons, Visual Studio generates other code that's hidden in the Designer.vb file. When the application is run, this is the code that implements the form and controls that you designed in the Form Designer. Although you may want to look at this code to see how it works, you shouldn't modify this code with the Code Editor as it may cause problems with the Form Designer.

When you enter Visual Basic code, you must be aware of the two coding rules summarized in this figure. First, you must separate the words in each statement by one or more spaces. Note, however, that you don't have to use spaces to separate the words from operators, although Visual Basic adds spaces for you by default. Second, if you want to continue a statement, you do that by coding a space followed by a *line-continuation character*, which is an underscore (_). You can see an example of that in both of the procedure declarations in this figure.

The event handlers for the Invoice Total form

```
Public Class frmInvoiceTotal

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

        Dim discountPercent As Decimal
        If txtSubtotal.Text >= 500 Then
            discountPercent = 0.2
        ElseIf txtSubtotal.Text >= 250 And txtSubtotal.Text < 500 Then
            discountPercent = 0.15
        ElseIf txtSubtotal.Text >= 100 And txtSubtotal.Text < 250 Then
            discountPercent = 0.1
        Else
            discountPercent = 0
        End If

        Dim discountAmount As Decimal = txtSubtotal.Text * discountPercent
        Dim invoiceTotal As Decimal = txtSubtotal.Text - discountAmount

        txtDiscountPercent.Text = FormatPercent(discountPercent, 1)
        txtDiscountAmount.Text = FormatCurrency(discountAmount)
        txtTotal.Text = FormatCurrency(invoiceTotal)

        txtSubtotal.Select()

    End Sub

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

End Class
```

Coding rules

- Use spaces to separate the words in each statement.
- To continue a statement to the next line, type a space followed by an underscore (the *line-continuation character*).

Description

- When you double-click the Calculate and Exit buttons in the Form Designer, it generates the shaded code shown above. Then, you can enter the rest of the code within the event handlers.
- The first event handler for the Invoice Total form is executed when the user clicks the Calculate button. This procedure calculates and displays the discount percent, discount amount, and total based on the subtotal entered by the user.
- The second event handler for the Invoice Total form is executed when the user clicks the Exit button. This procedure closes the form, which ends the application.

Figure 3-6 The event handlers for the Invoice Total form

How to code with a readable style

When you build an application, Visual Basic makes sure that your code follows all of its rules. If it doesn't, Visual Basic reports syntax errors that you have to correct before you can continue.

Besides adhering to the coding rules, though, you should try to write your code so it's easy to read, debug, and maintain. That's important for you, but it's even more important if someone else has to take over the maintenance of your code. You can create more readable code by following the four coding recommendations presented in figure 3-7. These recommendations are illustrated by the event handler in this figure.

The first coding recommendation is to use indentation and extra spaces to align related elements in your code. This is possible because you can use one or more spaces or tabs to separate the elements in a Visual Basic statement. In this example, all of the statements within the event handler are indented. In addition, the statements within each clause of the If statement are indented and aligned so you can easily identify the parts of this statement.

The second recommendation is to separate the words, values, and operators in each statement with spaces. If you don't, your code will be less readable as illustrated by the second code example in this figure. In this example, each line of code includes at least one operator. Because the operators aren't separated from the word or value on each side of the operator, though, the code is difficult to read. In contrast, the readable code includes a space on both sides of each operator.

The third recommendation is to use blank lines before and after groups of related statements to set them off from the rest of the code. This too is illustrated by the first procedure in this figure. Here, the code is separated into four groups of statements. In a short procedure like this one, this isn't too important, but it can make a long procedure much easier to follow.

The fourth recommendation is to use line-continuation characters so long statements are easier to read in the Code Editor window. This also makes these statements easier to read when you print them.

Throughout this chapter and book, you'll see code that illustrates the use of these recommendations. You will also receive other coding recommendations that will help you write code that is easy to read, debug, and maintain.

By default, the Code Editor automatically formats your code as you enter it. When you press the Enter key at the end of a statement, for example, the Editor will indent the next statement to the same level. In addition, it will capitalize all variable names so they match their declarations, and it will add a space before and after each operator.

A procedure written in a readable style

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

    Dim discountPercent As Decimal
    If txtSubtotal.Text >= 500 Then
        discountPercent = 0.2
    ElseIf txtSubtotal.Text >= 250 And txtSubtotal.Text < 500 Then
        discountPercent = 0.15
    ElseIf txtSubtotal.Text >= 100 And txtSubtotal.Text < 250 Then
        discountPercent = 0.1
    Else
        discountPercent = 0
    End If

    Dim discountAmount As Decimal = txtSubtotal.Text * discountPercent
    Dim invoiceTotal As Decimal = txtSubtotal.Text - discountAmount

    txtDiscountPercent.Text = FormatPercent(discountPercent, 1)
    txtDiscountAmount.Text = FormatCurrency(discountAmount)
    txtTotal.Text = FormatCurrency(invoiceTotal)

    txtSubtotal.Select()

End Sub
```

Statements written in a less readable style

```
dim discountAmount as Decimal=txtsubtotal.Text*discountpercent
dim invoiceTotal as Decimal=txtsubtotal.Text-discountamount
txtdiscountpercent.Text=formatpercent(discountpercent,1)
txtdiscountamount.Text=formatcurrency(discountamount)
txttotal.Text=formatcurrency(invoicetotal)
```

Coding recommendations

- Use indentation and extra spaces to align statements and clauses within statements so they reflect the structure of the program.
- Use spaces to separate the words, operators, and values in each statement.
- Use blank lines before and after groups of related statements.
- Use line-continuation characters to shorten long lines of code so they're easier to read in the Code Editor window.

Notes

- As you enter code in the Code Editor, Visual Studio may adjust the indentation, spacing, and capitalization so it's easier to read. This has no effect on the operation of the code.
- If Visual Basic doesn't adjust the code, check the Pretty Listing option in the Options dialog box. To find this option, expand the Text Editor group, the Basic group, and the VB Specific group.

Figure 3-7 How to code with a readable style

How to code comments

Comments can be used to document what the program does and what specific blocks or lines of code do. Since the Visual Basic compiler ignores comments, you can include them anywhere in a program without affecting your code. Figure 3-8 shows you how to code comments.

The basic idea is that you start a comment with an apostrophe. Then, anything after the apostrophe is ignored by the compiler. As a result, you can code whatever comments you want.

In this figure, you can see four lines of comments at the start of the procedure that describe what the procedure does. You can see one-line comments at the start of blocks of code that describe what the statements in those blocks do. And you can see one example of a comment that follows a statement on the same line.

Although some programmers sprinkle their code with comments, that shouldn't be necessary if you write your code so it's easy to read and understand. Instead, you should use comments only to clarify code that's difficult to understand. The trick, of course, is to provide comments for the code that needs explanation without cluttering the code with unnecessary comments. For example, an experienced Visual Basic programmer wouldn't need any of the comments shown in this figure.

One problem with comments is that they may not accurately represent what the code does. This often happens when a programmer changes the code, but doesn't change the comments that go along with it. Then, it's even harder to understand the code, because the comments are misleading. So if you change the code that has comments, be sure to change the comments too.

Incidentally, all comments are displayed in the Code Editor in a different color from the words in the Visual Basic statements. By default, the Visual Basic code is blue and black (blue for Visual Basic keywords and black for the rest of the code), while the comments are green. That makes it easy to identify the comments.

A procedure with comments

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

' ================================================================
' This procedure calculates the discount and total for an invoice.
' The discount depends on the invoice subtotal.
' ================================================================

' Determine the discount percent
    Dim discountPercent As Decimal
    If txtSubtotal.Text >= 500 Then
        discountPercent = 0.2
    ElseIf txtSubtotal.Text >= 250 And txtSubtotal.Text < 500 Then
        discountPercent = 0.15
    ElseIf txtSubtotal.Text >= 100 And txtSubtotal.Text < 250 Then
        discountPercent = 0.1
    Else
        discountPercent = 0
    End If

' Calculate the discount amount and invoice total
    Dim discountAmount As Decimal = txtSubtotal.Text * discountPercent
    Dim invoiceTotal As Decimal = txtSubtotal.Text - discountAmount

' Format the discount percent, discount amount, and invoice total
' and move these values to their respective text boxes
    txtDiscountPercent.Text = FormatPercent(discountPercent, 1)
    txtDiscountAmount.Text = FormatCurrency(discountAmount)
    txtTotal.Text = FormatCurrency(invoiceTotal)

    txtSubtotal.Select()      ' Move the focus to the Subtotal text box

End Sub
```

Coding recommendations

- Use comments only for portions of code that are difficult to understand.
- Make sure that your comments are correct and up-to-date.

Description

- *Comments* are used to help document what a program does and what the code within it does.
- To code a comment, type an apostrophe followed by the comment. You can use this technique to add a comment on its own line or to add a comment after the code on a line.
- During testing, you can comment out lines of code by coding an apostrophe before them. This is useful for testing new statements without deleting the old statements. Another way to comment out one or more lines of code is to select the lines and click on the Comment or Uncomment button in the Text Editor toolbar (see figure 3-10).

Figure 3-8 How to code comments

How to detect and correct syntax errors

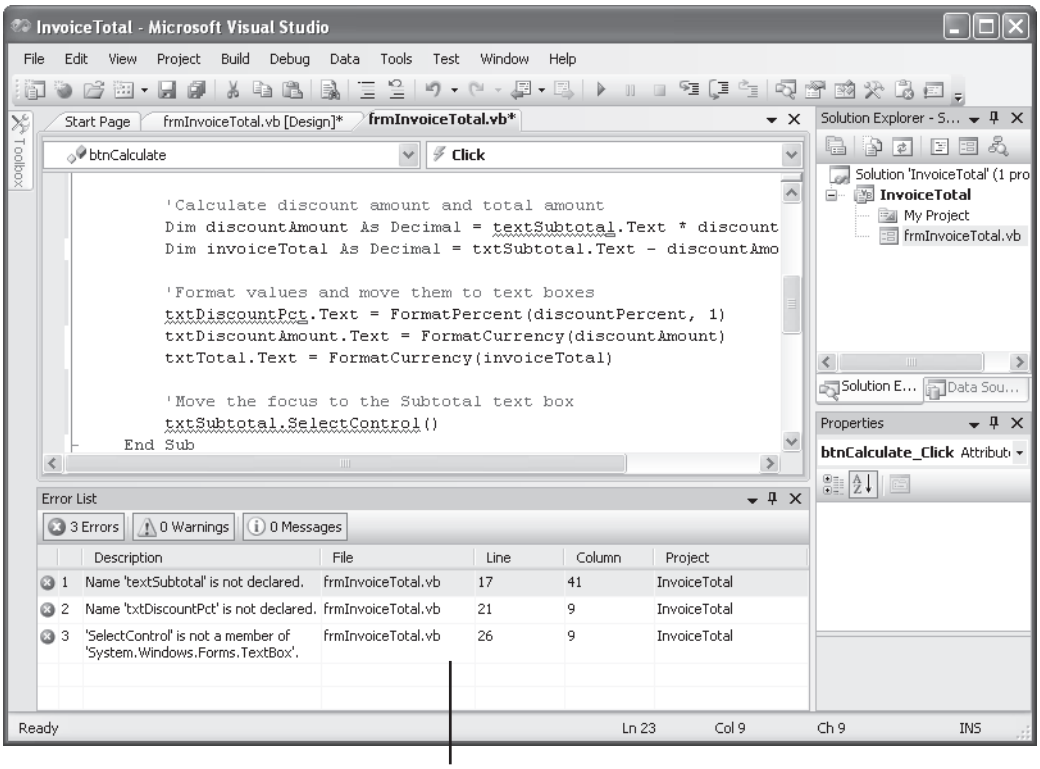
As you enter code, Visual Studio checks the syntax of each statement. If a *syntax error*, or *build error*, is detected, Visual Studio displays a wavy line under the code in the Code Editor. In the Code Editor in figure 3-9, for example, you can see wavy lines under three different portions of code. Then, if you place the mouse pointer over one of the errors, Visual Basic will display a description of the error.

If the Error List window is open as shown in this figure, any errors that Visual Studio detects are also displayed in that window. Then, you can double-click on an error message to jump to the related code in the Code Editor. After you correct a coding problem, its message is removed from the Error List window.

If the Error List window isn't open, you can display it by selecting the Error List command from the View menu. When you're learning Visual Basic, you're going to make a lot of coding errors, so it makes sense to keep this window open. But after you get used to Visual Basic, you can conserve screen space by using the Auto Hide button so this window is only displayed when you point to the Error List tab.

By the way, Visual Studio isn't able to detect all syntax errors as you enter code. So some syntax errors aren't detected until the project is built. You'll learn more about building projects later in this chapter.

The Code Editor and Error List windows with syntax errors displayed



Description

- Visual Studio checks the syntax of your Visual Basic code as you enter it. If a *syntax error* (or *build error*) is detected, it's highlighted with a wavy underline in the Code Editor, and you can place the mouse pointer over it to display a description of the error.
- If the Error List window is open, all of the build errors are listed in that window. Then, you can double-click on any error in the list to take you to its location in the Code Editor. When you correct the error, it's removed from the error list.
- If the Error List window isn't open, you can display it by selecting the Error List command from the View menu.
- Visual Studio doesn't detect some syntax errors until the project is built. As a result, you may encounter more syntax errors when you build and run the project.

Figure 3-9 How to detect and correct syntax errors

Other skills for working with code

The topics that follow present some other skills for working with code. You'll use many of these skills as you code and test your applications.

How to use the toolbar buttons

Whenever you work with a Windows application like Visual Studio, it's worth taking a few minutes to see what's available from the toolbar buttons. When you're entering or editing code, both the Standard and Text Editor toolbars provide some useful functions, and they are summarized in figure 3-10.

During testing, you can *comment out* several lines of code by selecting the lines of code and clicking the Comment Out button in the Standard toolbar. Then, you can test the application without those lines of code. Later, if you decide you want to use them after all, you can select the lines and click the Uncomment button to restore them. Similarly, you can use the Increase Indent and Decrease Indent buttons in the Text Editor toolbar to adjust the indentation for selected lines of code.

You can also use the Text Editor toolbar to work with *bookmarks*. After you use the Toggle button to set bookmarks on specific lines of code, you can move between the marked lines by clicking the next and previous buttons. Although you usually don't need bookmarks when you're working with simple applications, bookmarks can be helpful when you're working with large applications.

As you get more Visual Basic experience, you should also experiment with the first three buttons on the Text Editor toolbar. You'll find that they provide quick access to information like the member list for an object or the parameter list for a function.

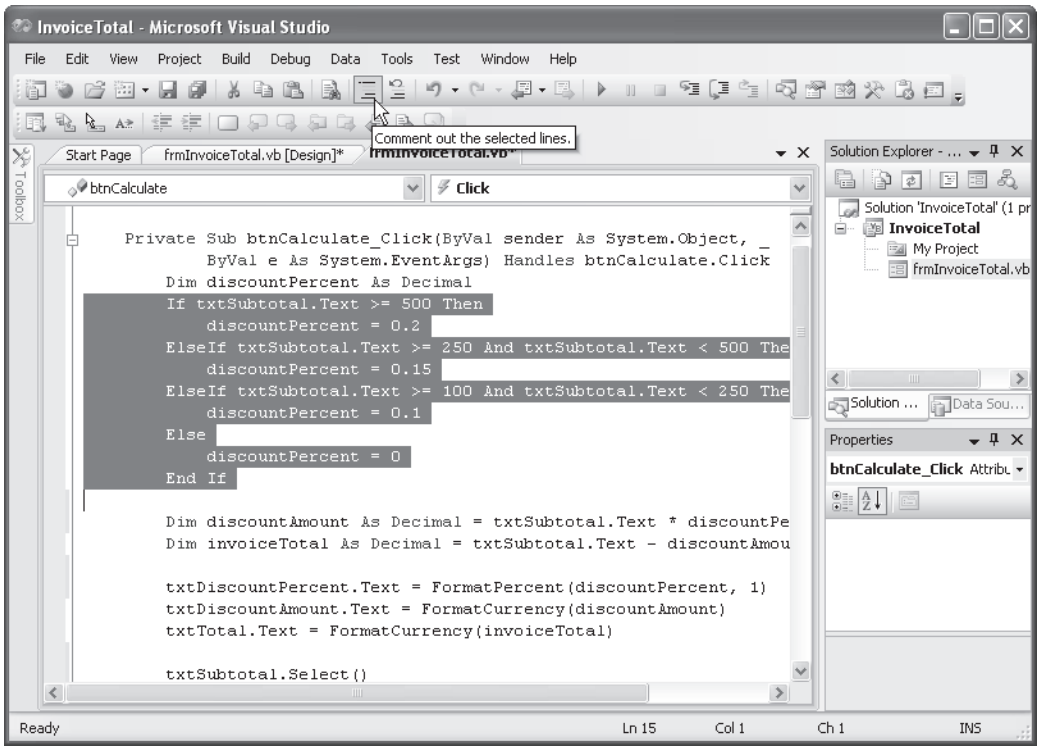
How to collapse or expand code

As you write the code for an application, you may want to *collapse* or *expand* some of the code. To do that, you can use the techniques described in figure 3-10. When you collapse the procedures that are already tested, it's easier to find what you're looking for in the rest of the code. And if you want to print the source code for a class, you don't have to print the collapsed code.

How to print the source code

Sometimes, it helps to print the code for the class that you're working on in the Code Editor window. To do that, you use the Print command in the File menu. Then, if you don't want to print the collapsed code, you check the Hide Collapsed Regions box. When the code is printed, any lines that extend beyond the width of the printed page are automatically wrapped to the next line.

The Code Editor and the Text Editor toolbar



How to use the Standard toolbar to comment or uncomment lines

- Select the lines and click the Comment Out or Uncomment button. When you *comment out* coding lines during testing, you can test new statements without deleting the old ones.

How to use the Text Editor toolbar

- To display or hide the Text Editor toolbar, right-click in the toolbar area and choose Text Editor from the shortcut menu.
- To increase or decrease the indentation of several lines of code, select the lines and click the Increase Indent or Decrease Indent button. Or, press the Tab and Shift+Tab keys.
- To move quickly between lines of code, you can use the last eight buttons on the Text Editor toolbar to set and move between *bookmarks*.

How to collapse or expand regions of code

- If a region of code appears in the Code Editor with a minus sign (-) next to it, you can click the minus sign to *collapse* the region so just the first line is displayed.
- If a region of code appears in the Code Editor with a plus sign (+) next to it, you can click the plus sign to *expand* the region so all of it is displayed.

Figure 3-10 How to use the toolbars and collapse or expand code

How to use code snippets

When you add code to an application, you will often find yourself entering the same pattern of code over and over. For example, you often enter a series of If blocks like the ones in the previous figures. To make it easy to enter patterns like these, Visual Studio provides a feature known as *code snippets*. These snippets make it easy to enter common control structures like the ones that you'll learn about in chapter 5.

To insert a code snippet on a blank line of text as shown in figure 3-11, right-click on the blank line in the Code Editor and select the Insert Snippet command from the shortcut menu. Then, double-click the name of the group (like Code Patterns), double-click the name of the subgroup (like Conditionals and Loops), and double-click the name of the snippet you want to insert.

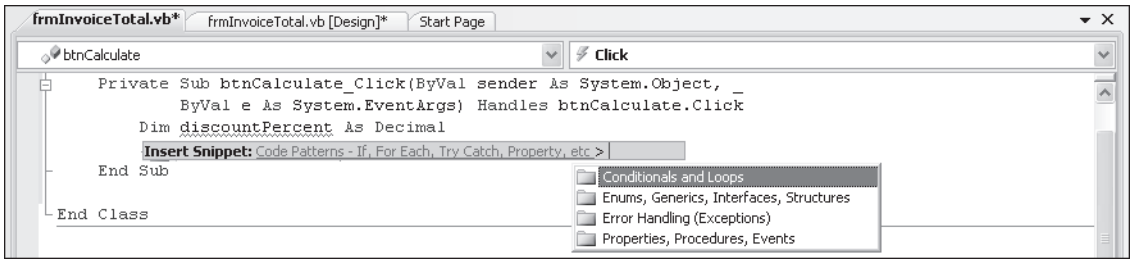
At that point, the code snippet is inserted into the Code Editor. In this figure, for example, you can see that If, ElseIf, Else, and End If lines have been inserted into the code. Now, you just need to replace the words True and False with conditions and enter the Visual Basic statements that you want executed for the If, ElseIf, and Else clauses.

Although code snippets make it easy to enter common patterns of code, it can be cumbersome to access them using the shortcut menu. Because of that, you may want to use the shortcuts for the code snippets you use most often. In the second screen in this figure, for example, the tool tip for the selected code snippet indicates that the shortcut for that code snippet is IfElseIf. To insert this code snippet using its shortcut, you just enter the shortcut and press the Tab key.

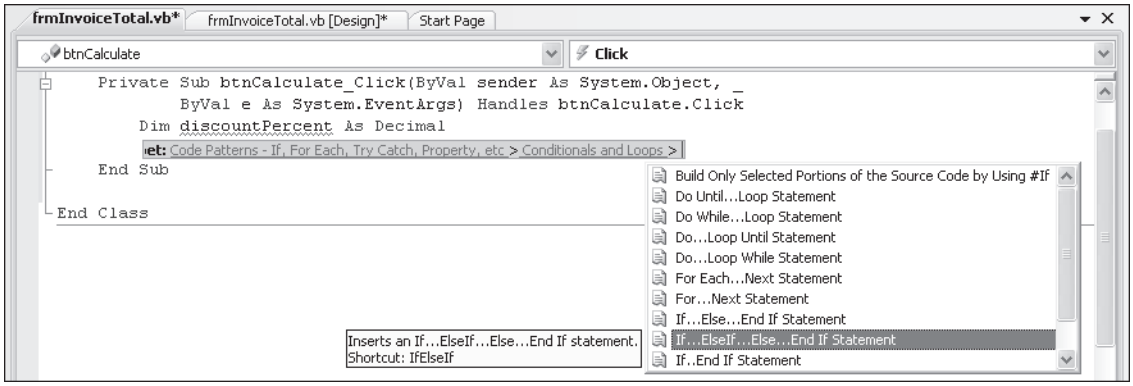
If you find that you like using code snippets, you should be aware that it's possible to add or remove snippets from the default list. To do that, you can choose the Code Snippets Manager command from the Tools menu. Then, you can use the resulting dialog box to remove code snippets that you don't use or to add new code snippets. Be aware, however, that writing a new code snippet requires creating an XML file that defines the code snippet. To learn how to do that, you can consult the documentation for Visual Studio.

Incidentally, if you're new to programming and don't understand the If statements in this chapter, don't worry about that. Just focus on the mechanics of using code snippets. In chapter 5, you'll learn everything you need to know about coding If statements.

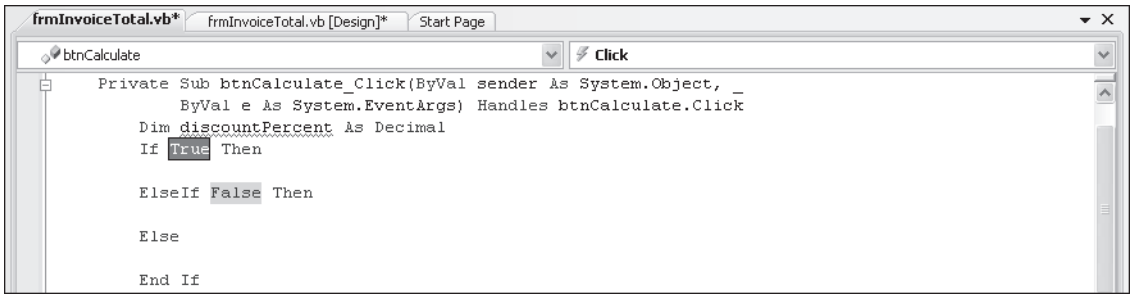
The list that's displayed for the Code Patterns group



The default list of code snippets for the Conditionals and Loops group



The If…ElseIf…Else…End If code snippet after it has been inserted



Description

- To insert a *code snippet*, right-click in the Code Editor and select the Insert Snippet command from the resulting menu. Then, select the code snippet you wish to insert. You can also insert a snippet by entering its shortcut and pressing the Tab key.
- Before you can select a snippet, you must get to the group that it's in. For instance, to use the snippet shown above, you must select the Code Patterns group followed by the Conditionals and Loops group.
- Once a snippet has been inserted into your code, you can replace the highlighted portions with your own code and add any other required code. To move from one highlighted portion of code to the next, you can press the Tab key.
- You can use the Tools➔Code Snippets Manager command to display a dialog box that you can use to edit the list of available code snippets and to add custom code snippets.

Figure 3-11 How to use code snippets

How to rename identifiers

As you work on the code for an application, you may decide to change the name of a variable, procedure, class, or other *identifier*. When you do that, you'll want to be sure that you change all occurrences of the identifier. Figure 3-12 shows you two techniques you can use to rename an identifier.

The first technique shown in this figure is to rename the identifier from its declaration. Here, the name of the discountPercent variable is being changed to discountPct. When you change an identifier like this, a bar appears under the last character of the name as shown in the first screen. Then, you can move the mouse pointer over the bar and click the drop-down arrow that appears to display a *smart tag menu*. This menu shows a Rename command that you can use to change all occurrences of the identifier in your project.

You can also rename an identifier from any occurrence of that identifier. To do that, just right-click on the identifier, select the Rename command, and enter the new name into the Rename dialog box that's displayed.

How to rename an identifier from its declaration

The bar that appears under a renamed identifier

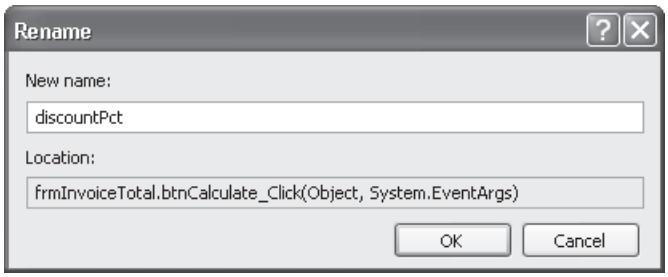
```
Dim discountPct As Decimal
```

The menu that's available from the bar

```
Dim discountPct As Decimal
If txtSubtot ext >= 500 Then
    discount
ElseIf txtSu                              al.Text < 500 Then
    discountPercent = 0.15
```

Rename 'discountPercent' to 'discountPct'

How to rename an identifier from any occurrence



Description

- Visual Studio lets you rename *identifiers* in your code, such as variable, procedure, and class names. This works better than using search-and-replace because when you use rename, Visual Studio is aware of how the identifier is used in your project.
- When you change the declaration for an identifier in your code, Visual Studio displays a bar beneath the last character of the identifier. Then, you can move the mouse pointer over the bar, click the drop-down arrow that's displayed, and select the Rename command from the *smart tag menu*.
- You can also rename an identifier from anywhere it's used in your project. To do that, right-click the identifier and select the Rename command from the shortcut menu to display the Rename dialog box. Then, enter the new name for the identifier.

Figure 3-12 How to rename identifiers

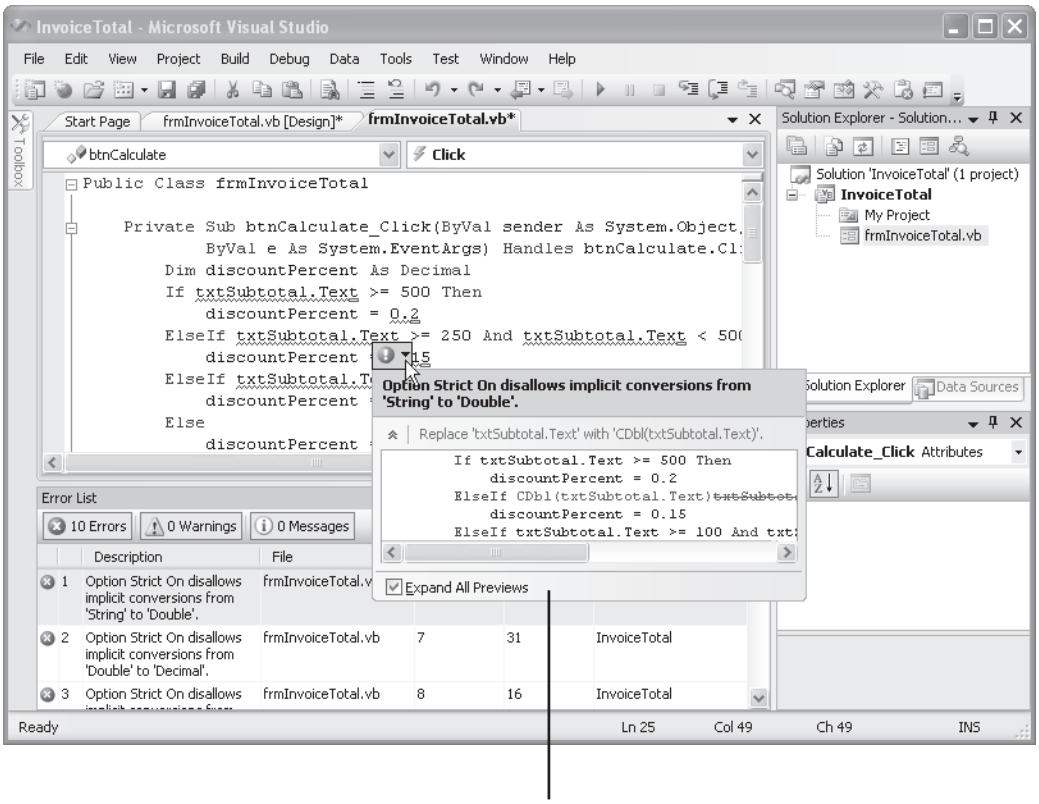
How to use the Smart Compile Auto Correction feature

As you learned in figure 3-9, Visual Studio puts a wavy line under any syntax errors that it detects while you're entering code. In some cases, though, Visual Studio takes that one step further with its Smart Compile Auto Correction feature. In those cases, a bar appears at the right end of the wavy underline.

To use this feature, you place the mouse pointer over this bar to display a smart tag. Then, you can click the drop-down arrow that appears to display the Error Correction Options window shown in figure 3-13. This window includes a description of the error, suggestions for correcting the error, and a preview of how the code will look if you apply the corrections. If you like the suggested corrections, you just click on the suggestion to apply them.

For this example, I set the Option Strict option on, which you'll learn how to do in the next chapter. Because that forces you to do some data conversion before comparisons or arithmetic calculations can be done, the suggested changes do those data conversions. This illustrates the power of this feature, so you're going to want to use it whenever it's available.

The Code Editor with the Error Correction Options window displayed



Error Correction Options window

Description

- When Visual Studio detects a syntax error, it highlights the error with a wavy underline in the Code Editor.
- If a bar appears at the right end of a wavy underline, you can use the Error Correction Options window to view and apply suggested corrections.
- To display the Error Correction Options window, place the mouse pointer over the bar, then over the *smart tag* that's displayed, and click the drop-down arrow.
- To apply a correction, click the appropriate "Replace…" link.

Note

- To get the errors and the suggested corrections in the screen above, I turned the Option Strict option on. You'll learn more about that in the next chapter.

Figure 3-13 How to use the Smart Compile Auto Correction feature

How to use the My feature

The *My feature* that was introduced with Visual Basic 2005 can improve your productivity by making it easy to access .NET Framework classes and functions that would be otherwise difficult to find. If you're new to programming, this feature may not make much sense to you, but it will as you learn more about Visual Basic. So for now, it's enough to know that this feature is available and trust that you'll learn more about it later.

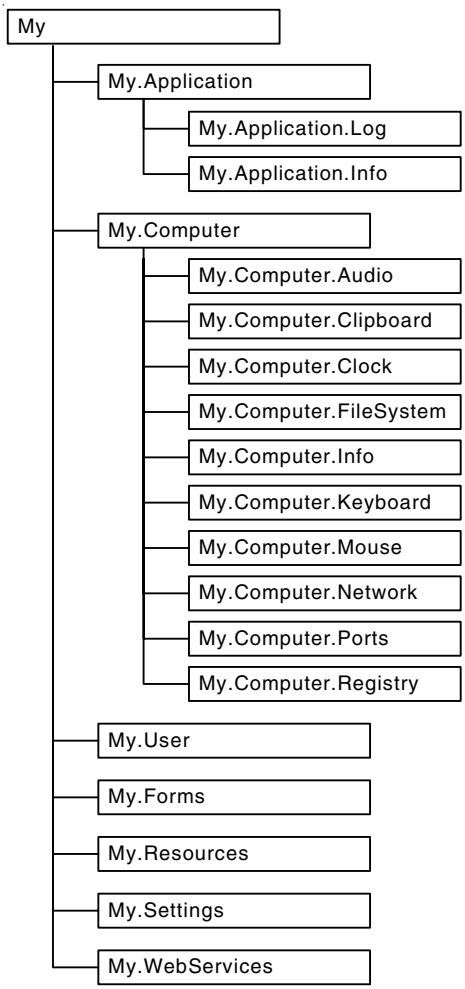
Figure 3-14 illustrates how this feature works. As you can see, the My feature exposes a hierarchy of objects that you can use to access information. These objects are created automatically when you run an application. The three statements in this figure illustrate how you can use some of these objects. To get more information about any of these objects, you can use the Help feature as described later in this chapter.

The first statement in this figure shows how you can use the Name property of the My.User object to get the name of the user of an application. By default, this property returns both the domain name and the user name. To get this information without using this object, you would have to use the UserName and UserDomainName properties of the Windows.Forms.SystemInformation class. This illustrates how the My objects can make finding the information you need more intuitive.

The second statement shows how you can use the My.Computer.FileSystem object to check if a specified directory exists on the user's computer. To do that, it uses the DirectoryExists method of this object. In chapter 21, you'll learn about many of the properties and methods of this object that you can use to work with drives, directories, and files.

The third statement shows how you can use the My.Forms object to display a form. Note that when you use this technique, an instance of the form is created automatically if it doesn't already exist. This wasn't possible with Visual Basic 2002 or 2003. You'll learn more about how to use the My.Forms object in chapter 24.

The main My objects for Windows Forms applications



A statement that gets the name of the current user of an application

```
lblName.Text = My.User.Name
```

A statement that checks if a directory exists

```
If My.Computer.FileSystem.DirectoryExists("C:\VB 2008\Files") Then ...
```

A statement that creates an instance of a form and displays it

```
My.Forms.frmInvestment.Show()
```

Description

- The My feature makes it easy to access frequently used .NET Framework classes and functions using objects that are grouped by the tasks they perform. These objects are created automatically when an application is run.

Figure 3-14 How to use the My feature

How to get help information

As you develop applications in Visual Basic, it's likely that you'll need some additional information about the IDE, the Visual Basic language, an object, property, method, event, or some other aspect of Visual Basic programming. Figure 3-15 shows several ways you can get that information.

When you're working in the Code Editor or the Form Designer, the quickest way to get help information is to press F1 while the insertion point is in a keyword or an object is selected. Then, Visual Studio opens a separate Help window like the one shown in this figure and displays the available information about the selected keyword or object. Another way to launch a Help window is to select a command from Visual Studio's Help menu such as the Search, Contents, or Index command.

The Help window is split into two panes. The right pane shows the last help topic that you accessed. In this figure, for example, the right pane displays a help topic that provides information about working with the Code Editor.

The left pane, on the other hand, displays the Index, Contents, and Help Favorites tabs that help you locate help topics. In this figure, for example, the left pane displays the Index tab. At the top of this tab, the drop-down list has been used to filter help topics so they're appropriate for Visual Basic programmers. In addition, "code e" has been entered to navigate to the index entries that begin with those letters, and the Code Editor entry has been selected.

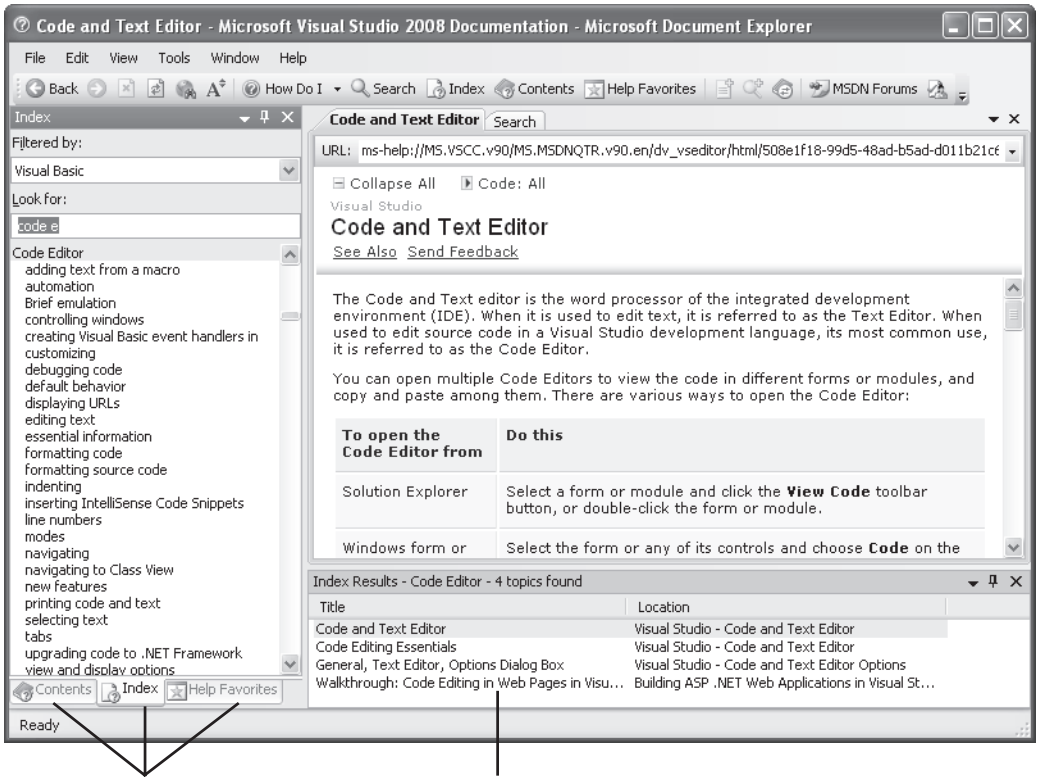
In addition to the topic that's displayed in the right pane, all the topics that are available for a selected entry are listed in the Index Results window that's displayed across the bottom of the screen. When the Code Editor entry was selected in this figure, for example, four topics were listed in the Index Results window and the first topic was displayed by default. To display another topic, you simply click on it.

In the left pane, you can click on the Contents tab to display a list of help topics that are grouped by category. Or, you can click on the Help Favorites tab to view a list of your favorite help topics. At first, the Help Favorites tab won't contain any help topics. However, you can add topics to this tab by displaying a topic and clicking on the Add To Help Favorites button that's available from the toolbar.

You can display a Search tab in the right pane by clicking on the Search button in the toolbar. From this tab, you can enter a word or phrase to search for and select the languages, technologies, and content you want to search. Then, when you click the Search button, the results are displayed in the tab and you can click a topic to display it.

When you display information in the Help window, you should realize that the Help window uses a built-in web browser to display help topics that are available from your computer and from the Internet. In addition, the Help window works much like a web browser. To jump to a related topic, you can click on a hyperlink. To move forward and backward through previously displayed topics, you can use the Forward and Back buttons. As a result, with a little practice, you shouldn't have much trouble using this window.

The Help window



Description

- You can display a Help window by selecting an object in the Form Designer or positioning the insertion point in a keyword in the Code Editor and pressing F1.
- You can also display a Help window by selecting a command (such as Index, Contents, or Search) from Visual Studio's Help menu.
- The Help window works like a web browser and can display help topics that are available from your computer or from the Internet. You can use the buttons in its toolbar to navigate between help topics or to add topics to your list of favorite topics.
- The Help window is divided into two panes. The left pane displays the Index, Content, and Help Favorites tabs that let you locate the help topics you want to display. The right pane displays the last help topic that you accessed.
- If you click on the Search button, the right pane will display a Search tab that lets you search for help topics by entering a word or phrase.
- If you click on the How Do I button, the right pane will display a How Do I tab that lets you go to a topic by clicking on a link.
- To close a tab, click on the Close button when the tab is active. To display a tab, click the tab or select it from the Active Files drop-down list that's next to the Close button.

Figure 3-15 How to get help information

How to run, test, and debug a project

After you enter the code for a project and correct any syntax errors that are detected as you enter this code, you can run the project. When the project runs, you can test it to make sure it works the way you want it to, and you can debug it to remove any programming errors you find.

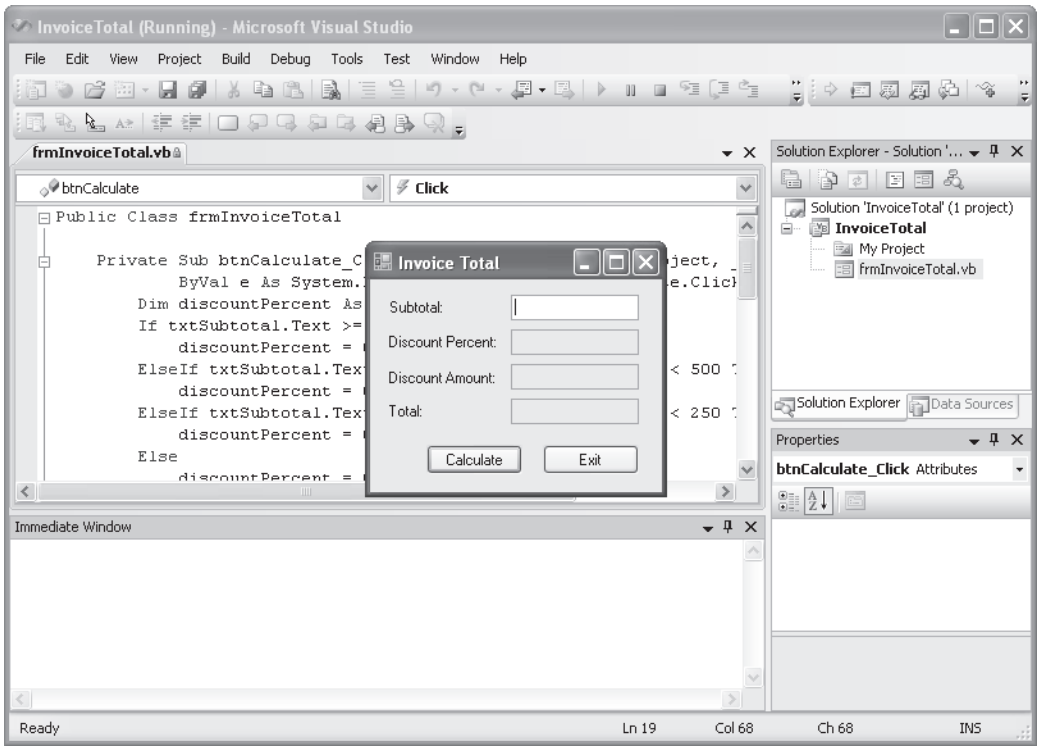
How to run a project

As you learned in chapter 1, you can *run* a project by clicking the Start Debugging button in the Standard toolbar, selecting the Start Debugging command from the Debug menu, or pressing the F5 key. This *builds* the project if it hasn't been built already and causes the project's form to be displayed, as shown in figure 3-16. When you close this form, the application ends. Then, you're returned to Visual Studio where you can continue working on your program.

You can also build a project without running it as described in this figure. In most cases, though, you'll run the project so you can test and debug it.

If build errors are detected when you run a project, the errors are displayed in the Error List window, and you can use this window to identify and correct the errors. If it isn't already displayed, you can display this window by clicking on the Error List tab that's usually displayed at the bottom of the window or by using the View➔Error List command.

The form that's displayed when you run the Invoice Total project



Description

- To *run* a project, click the Start Debugging button in the Standard toolbar, select the Debug➔Start Debugging menu command, or press the F5 key. This causes Visual Studio to *build* the project and create an assembly. Then, if there are no build errors, the assembly is run so the project's form is displayed as shown above.
- If syntax errors are detected when a project is built, they're listed in the Error List window and the project does not run.
- You can build a project without running it by selecting the Build➔Build Solution command.
- When you build a project for the first time, all of the components of the project are built. After that, only the components that have changed are rebuilt. To rebuild all components whether or not they've changed, use the Build➔Rebuild Solution command.

Figure 3-16 How to run a project

How to test a project

When you *test* a project, you run it and make sure the application works correctly. As you test your project, you should try every possible combination of input data and user actions to be certain that the project works correctly in every case. In other words, your goal is to make the project fail. Figure 3-17 provides an overview of the testing process for Visual Basic applications.

To start, you should test the user interface. Make sure that each control is sized and positioned properly, that there are no spelling errors in any of the controls or in the form's title bar, and that the navigation features such as the tab order and access keys work properly.

Next, subject your application to a carefully thought-out sequence of valid test data. Make sure you test every combination of data that the project will handle. If, for example, the project calculates the discount at different values based on the value of the subtotal, use subtotals that fall within each range.

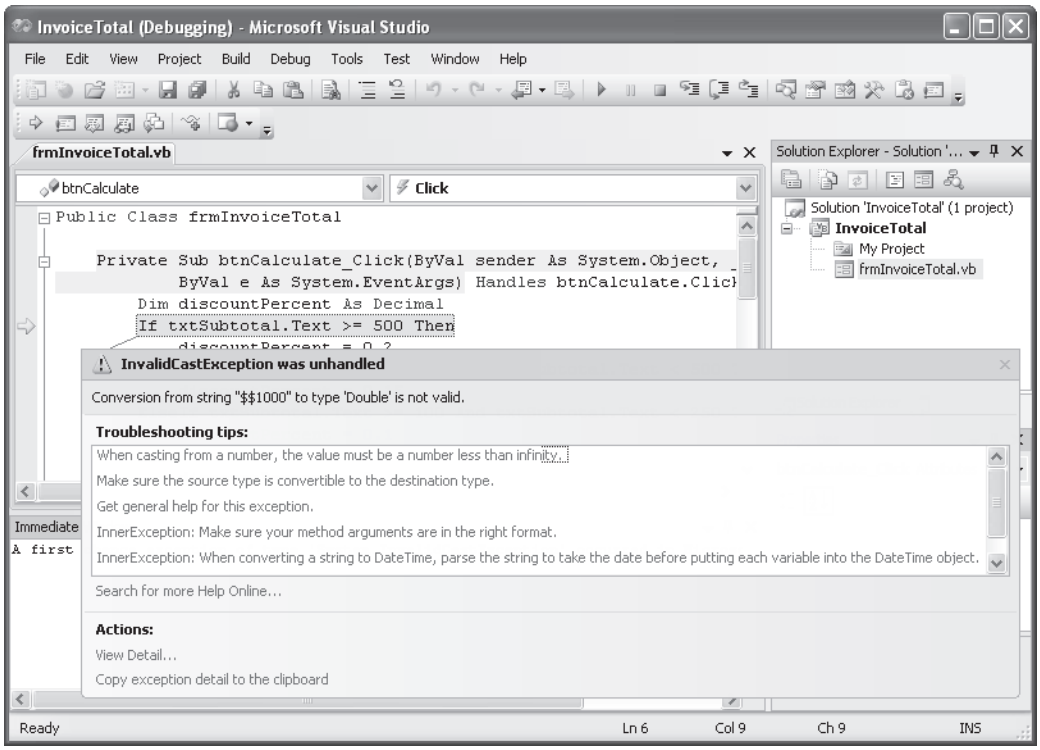
Finally, test the program to make sure that it properly handles invalid data entered by users. For example, type text information into text boxes that expect numeric data. Leave fields blank. Use negative numbers where they shouldn't be allowed. Remember that the goal of testing is to find all of the problems.

As you test your projects, you'll encounter *runtime errors*. These errors, also known as *exceptions*, occur when Visual Basic encounters a problem that prevents a statement from being executed. If, for example, a user enters "ABC" into the Subtotal text box on the Invoice Total form, a runtime error will occur when the program tries to assign that value to a decimal variable.

When a runtime error occurs, Visual Studio breaks into the debugger and displays an Exception Assistant window like the one in this figure. Then, you can use the debugging tools that you'll be introduced to in the next figure to debug the error.

Runtime errors, though, should only occur when you're testing a program. Before an application is put into production, it should be coded and tested so all runtime errors are caught by the application and appropriate messages are displayed to the user. You'll learn how to do that in chapter 7 of this book.

The Exception Assistant that's displayed when a runtime error occurs



How to test a project

1. Test the user interface. Visually check all the controls to make sure they are displayed properly with the correct text. Use the Tab key to make sure the tab order is set correctly, verify that the access keys work right, and make sure that the Enter and Esc keys work properly.
2. Test valid input data. For example, enter data that you would expect a user to enter.
3. Test invalid data or unexpected user actions. For example, leave required fields blank, enter text data into numeric input fields, and use negative numbers where they are not appropriate. Try everything you can think of to make the application fail.

Description

- To *test* a project, you run the project to make sure it works properly no matter what combinations of valid or invalid data you enter or what sequence of controls you use.
- If a statement in your application can't be executed, a *runtime error*, or *exception*, occurs. Then, if the exception isn't handled by your application, the statement that caused the exception is highlighted and an Exception Assistant window like the one above is displayed. At that point, you need to debug the application.

Figure 3-17 How to test a project

How to debug runtime errors

When a runtime error occurs, Visual Studio enters *break mode*. In that mode, Visual Studio displays the Code Editor and highlights the statement that couldn't be executed, displays the Debug toolbar, and displays an Exception Assistant dialog box like the one shown in figure 3-17. This is designed to help you find the cause of the exception (the *bug*), and to *debug* the application by preventing the exception from occurring again or by handling the exception.

Often, you can figure out what caused the problem just by knowing what statement couldn't be executed, by reading the message displayed by the Exception Assistant, or by reading the troubleshooting tips displayed by the Exception Assistant. But sometimes, it helps to find out what the current values in some of the variables or properties in the program are.

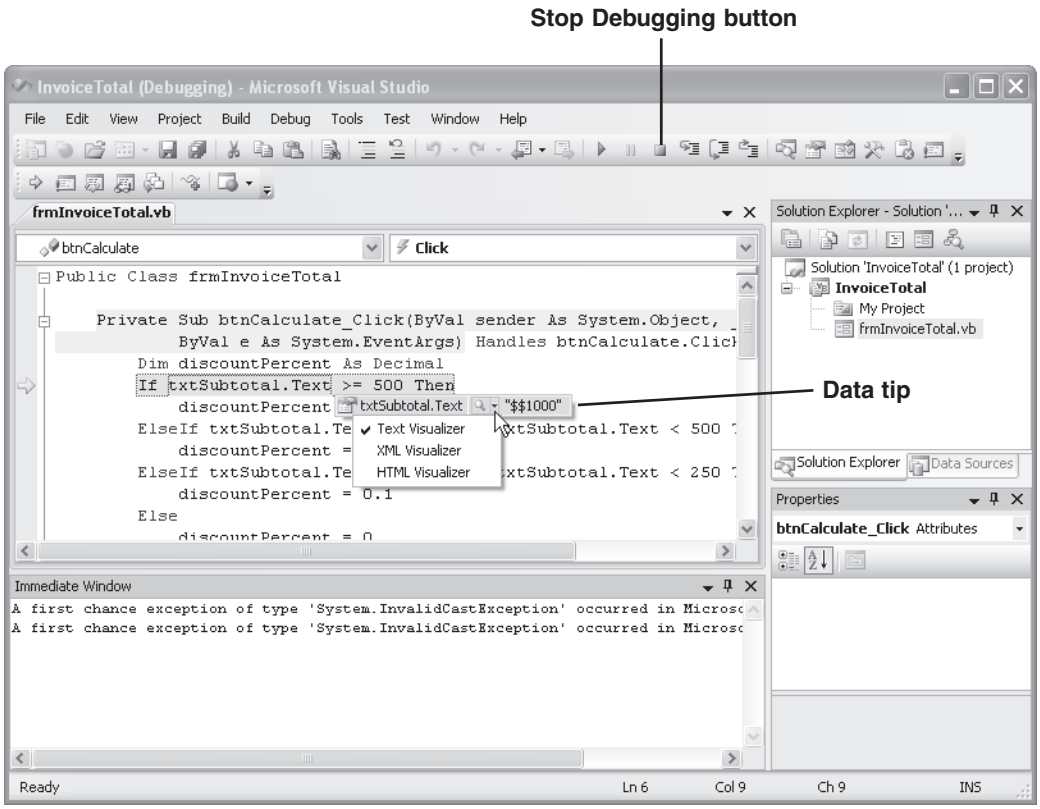
To do that, you can place the mouse pointer over a variable or property in the code so a *data tip* is displayed as shown in figure 3-18. This tip displays the current value of the variable or property. You can do this with the Exception Assistant still open, or you can click on its Close button to close it. Either way, the application is still in break mode. In this figure, the data tip for the Text property of the txtSubtotal control is "$$1000", which shows that the user didn't enter valid numeric data.

Within the data tip, you'll see a magnifying glass and an arrow for a drop-down list. If you click on this arrow, you'll see the three choices shown in this figure. Then, if you click on Text Visualizer, the value in the data tip will be shown in the Text Visualizer dialog box the way it actually is. So in this simple example, the value will show as $$1000, not "$$1000". Although that isn't much different than what the data tip shows, the differences are more dramatic when the data is more complex.

Once you find the cause of a bug, you can correct it. Sometimes, you can do that in break mode and continue running the application. Often, though, you'll exit from break mode before fixing the code. To exit, you can click the Stop Debugging button in the Standard toolbar. Then, you can correct the code and test the application again.

For now, don't worry if you don't know how to correct the problem in this example. Instead, you can assume that the user will enter valid data. In chapter 7, though, you'll learn how to catch exceptions and validate all user entries for an application because that's what a professional application has to do. And in chapter 12, after you've learned the Visual Basic essentials, you'll learn a lot more about debugging.

How a project looks in break mode



Description

- When an application encounters a runtime error, you need to fix the error. This is commonly referred to as *debugging*, and the error is commonly referred to as a *bug*.
- When an application encounters a runtime error, it enters *break mode*. In break mode, the Debug toolbar is displayed along with the Exception Assistant window.
- The information in the Exception Assistant window should give you an idea of what the error might be. You can also click on the links in the Troubleshooting Tips list to display more information in a Help window.
- If you close the Exception Assistant window, the application remains in break mode.
- To display a *data tip* for a property or variable, move the mouse pointer over it in the Visual Basic code.
- If the data tip includes a drop-down arrow to the right of a magnifying glass, you can click the error and select Text Visualizer to see exactly what the data looks like.
- To exit break mode and end the application, click the Stop Debugging button in the Standard toolbar or press Shift+F5.
- You'll learn more about debugging and the Exception Assistant window in chapter 12.

Figure 3-18 How to debug runtime errors

Perspective

If you can code and test the Invoice Total project that's presented in this chapter, you've already learned a lot about Visual Basic programming. You know how to enter the code for the event handlers that make the user interface work the way you want it to. You know how to build and test a project. And you know some simple debugging techniques.

On the other hand, you've still got a lot to learn. In particular, you haven't learned much about the Visual Basic language. That's why the next six chapters present the Visual Basic essentials.

Terms

object-oriented programming
object-oriented language
object
class
instance
instantiation
property
method
event
member
dot operator
dot
argument
event-driven application
event handler
procedure declaration
procedure name
tool tip
line-continuation character
comment
syntax error
build error
code snippet
identifier
smart tag menu
My feature
comment out a line
bookmark
collapse
expand
build a project
run a project
test a project
runtime error
exception
bug
debug
break mode
data tip

Exercise 3-1 Code the Invoice Total form

In this exercise, you'll add code to the Invoice Total form that you designed in exercise 2-1. Then, you'll build and test the project to be sure it works. You'll also experiment with debugging and review some help information.

Copy and open the Invoice Total application

1. Use the Windows Explorer to copy the Invoice Total project that you created for chapter 2 from the C:\VB 2008\Chapter 02 directory to the C:\VB 2008\Chapter 03 directory.
2. Open the Invoice Total solution (InvoiceTotal.sln) that's now in the C:\VB 2008\Chapter 03\InvoiceTotal directory.

Add code to the form and correct syntax errors

3. Display the Invoice Total form in the Form Designer, and double-click on the Calculate button to open the Code Editor and generate the procedure declaration for the Click event of this object. Then, enter the code for this procedure as shown in figure 3-6. As you enter the code, be sure to take advantage of all of the Visual Studio features for coding including snippets.
4. Return to the Form Designer, and double-click the Exit button to generate the procedure declaration for the Click event of this object. Enter the statement shown in figure 3-6 for this event handler.
5. Open the Error List window as described in figure 3-9. If any syntax errors are listed in this window, double-click on each error to move to the error in the Code Editor. If the Auto Correction feature is available for an error, check to see whether its suggested correction (or one of its suggested corrections) is the one you want to make. Then, correct the error.

Test the application

6. Press F5 to build and run the project. If any syntax errors are detected, you'll need to correct the errors and press F5 again.
7. When the application runs and the Invoice Total form is displayed, enter a valid numeric value in the first text box and click the Calculate button or press the Enter key to activate this button. Assuming that the calculation works, click the Exit button or press the Esc key to end the application. If either of these procedures doesn't work right, of course, you need to debug the problems and test the application again.

Enter invalid data and display data tips in break mode

8. Start the application again. This time, enter xx for the subtotal. Then, click the Calculate button. This will cause Visual Studio to enter break mode and display the Exception Assistant.
9. Note the highlighted statement and read the message that's displayed in the Exception Assistant. Then, close the Assistant, and move the mouse pointer over the property in this statement to display its data tip. This shows that the code for this application needs to be enhanced so it checks for invalid data.
10. Display the smart tag for the Text property in the highlighted statement, click its drop-down arrow, and select Text Visualizer. This shows the data exactly as it was entered in the Text Visualizer dialog box. Then, click the Stop Debugging button in Standard toolbar to end the application.

Experiment with the Visual Basic features

11. In the Dim statement for the discountPercent variable, change the variable name to discountPct. When you do that, a bar will appear under the last letter of the variable. Place the mouse pointer over this bar to display a drop-down arrow. Then, click on this arrow and select the Rename command. This should rename the discountPercent variable to discountPct throughout the form. But run the form to make sure it's working correctly.

12. In the If statement, right-click on one of the occurrences of the variable named discountPct. Then, select the Rename command, and use it to rename this variable to discountPercent throughout the form. To make sure this worked, run the application.
13. Select the lines that contain the ElseIf clauses and click on the Comment Out button in the Standard toolbar. That should change these coding lines to comments. Then, run the application to see how it works when these lines are ignored. When you're done, select the lines that were commented out and click on the Uncomment button to restore them.
14. In the Code Editor, click on the minus sign in front of the btnCalculate_Click procedure to collapse it. Then, expand that procedure and collapse the btnExit_Click procedure. Last, print just the expanded code for this form.
15. In the Solution Explorer, show all the files and double-click on the file named frmInvoiceTotal.Designer.vb to open it in the Code Editor. This is the code that determines how the form will look when it's instantiated. After you read chapter 11 and section 4, this code will make more sense to you. For now, though, just close the window with this code.

Experiment with the Help feature

16. To see how context-sensitive help works, place the insertion point in the Select method in the last statement of the first event handler and press F1. This should open a Help window that tells you more about this method.
17. In the left pane, select the Index tab to display the Index window. Type "snippets" into the Look For box to see the entries that are listed under this topic. Next, if Visual Basic (or Visual Basic Express Edition) isn't selected in the Filtered By drop-down list, select it to show just the topics for Visual Basic. Then, click on one or more topics to display them.
18. Use the Tools➔Options command, and review the help options. In the Online group, you may want to change the loading option to "Try local only, not online," because that can speed up the access of help topics.
19. Continue to experiment with the Index, Contents, Help Favorites, and Search features to see how they work, and try to use some of the buttons in the Standard toolbar to see how they work. Then, close the Help window.

Exit from Visual Studio

20. Click the Close button for the Visual Studio window to exit from this application. If you did everything and got your application to work right, you've come a long way!