Thanks for downloading this chapter from Murach’s Visual Basic 2015. 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 website. 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 related topics.

Thanks for your interest in our books!
What developers have said about previous editions

“This book did exactly what the authors claimed and taught me how to build business applications. I would rate it a 10 out of 10.”
Chas Profitt, DevCity.net

“I can’t praise this book highly enough. The clarity Anne uses in picking what to include, when to introduce it, and how to do so is remarkable.”
Charles Ferguson, Software Developer, Sydney, Australia

“I now own at least 5 of your books and am finishing up my second week at my new job as a professional software developer. I was given two different programming tests while interviewing for this position, and I am confident that I would not have passed had it not been for your VB book.
Stephen Lajoie, MIS Developer, California

“This book is terrific! I’m very impressed with the simplicity and straight-to-the-point approach.”
Leonardo Ingles de Sousa, Finance Manager, Brazil

“You guys do an excellent job of presenting the material in an understandable fashion. I tried some books from other publishers and almost gave up! [But] after reading the first chapter in your VB book, I realized I could do it.”
Bart McKinney, IS Coordinator, North Carolina

“Murach’s was the book that got me writing code to access databases. I refer to it constantly and recommend it highly.”
Posted at an online bookseller

“I have a huge library of computer books. Yours are THE best! Your Visual Basic book is so good, I opened it and did the whole application from scratch (even though I’d never tried VB before).”
T. Tikhonko, Developer, New York
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 ...................................................58
Introduction to object-oriented programming..........................58
How to refer to properties, methods, and events .........................60
How an application responds to events..................................62

How to add code to a form ..................................................64
How to create an event handler for the default event of a form or control...64
How IntelliSense helps you enter the code for a form..................66
Coding rules for Visual Basic statements..................................68
The event handlers for the Invoice Total form..........................70
How to code with a readable style..........................................72
How to code comments ........................................................74
How to detect and correct syntax errors...................................76

More skills for working with code........................................78
How to use the toolbar buttons.............................................78
How to collapse or expand code.............................................78
How to zoom in and out.......................................................80
How to highlight symbols....................................................80
How to print the source code.................................................80
How to use code snippets....................................................82
How to refactor code........................................................84
How to perform quick actions with light bulbs..........................86
How to use the annotations in the scroll bar.........................88
How to use the My feature...................................................90
How to get help information.................................................92

How to run, test, and debug a project.................................94
How to run a project.........................................................94
How to test a project........................................................96
How to debug runtime errors..............................................98

Perspective .........................................................................100
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 Forms 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.
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 or entering a space. If the member name is followed by another character, such as another period, you can also complete the entry by typing that character.

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

\[
\text{ClassName.} \text{MemberName} \\
\text{objectName.} \text{MemberName}
\]

Statements that refer to properties

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>txtTotal.Text = 10</code></td>
<td>Assigns the value 10 to the Text property of the text box named txtTotal.</td>
</tr>
<tr>
<td><code>txtTotal.ReadOnly = True</code></td>
<td>Assigns the True value to the ReadOnly property of the text box named txtTotal so the user can’t change its contents.</td>
</tr>
</tbody>
</table>

Statements that refer to methods

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>txtMonthlyInvestment.Select()</code></td>
<td>Uses the Select method to move the focus to the text box named txtMonthlyInvestment.</td>
</tr>
<tr>
<td><code>Me.Close()</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

Code that refers to an event

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>btnExit.Click</code></td>
<td>Refers to the Click event of a button named btnExit.</td>
</tr>
</tbody>
</table>

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, you can type one or more letters of the member name, and Visual Studio will select the first entry in the list that starts with those letters. You can also scroll through the list to select the member you want. Once it’s selected, press the Tab or Enter key to insert the member into your code.

- 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.
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 clicks 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

![Invoice Total form](image)

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

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

**Common control events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurs when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>the user clicks the control.</td>
</tr>
<tr>
<td>DoubleClick</td>
<td>the user double-clicks the control.</td>
</tr>
<tr>
<td>Enter</td>
<td>the focus is moved to the control.</td>
</tr>
<tr>
<td>Leave</td>
<td>the focus is moved from the control.</td>
</tr>
</tbody>
</table>

**Common form events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurs when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>the form is loaded into memory.</td>
</tr>
<tr>
<td>Closing</td>
<td>the form is closing.</td>
</tr>
<tr>
<td>Closed</td>
<td>the form is closed.</td>
</tr>
</tbody>
</table>

**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.
How to add code to a form

Now that you understand some of the concepts behind object-oriented programming, 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 on its tab.

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 event handler is stored in the vb file for the form.
• 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.
• If you’re using Visual Studio Professional or Enterprise Edition, the Code Editor indicates how many times each class and member is referred to by your application. This can be helpful when you code your own classes as shown in chapter 11.
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. 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. And it 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 completion list that IntelliSense displays when you start to enter a new line of code. Here, because I entered the letter d, the list is positioned on the last item I used that begins with that letter. In this case, it’s positioned on the Dim keyword. If you continue to enter letters, Visual Studio will filter the list so it includes only items that contain those letters and it will select the appropriate item based on your entry. 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.

When you select an item in a list, Visual Studio displays information about that item in a tool tip. For example, the tool tip for the Dim keyword indicates what this statement does and shows its syntax. 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 use these IntelliSense features, you’ll see that they can help you avoid introducing errors into your code. 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.

Although it’s not shown here, Visual Studio also lets you see the code that’s behind an IntelliSense list without closing the list. To do that, you simply press and hold the Ctrl key and the list is hidden until you release that key.
The list that’s displayed when you enter a letter at the beginning of a line of code

![Image](image.png)

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

![Image](image.png)

Description

- The IntelliSense that’s provided for Visual Basic 2015 lists keywords, data types, functions, variables, objects, classes, and so on as you type so you can enter them correctly.
- When you highlight an item in a completion 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 and hold the Ctrl key. Then, the list is hidden until you release the Ctrl key.

Figure 3-5  How IntelliSense helps you enter the code for a form
Coding rules for Visual Basic statements

When you enter Visual Basic code, you must be aware of the two coding rules summarized in figure 3-6. 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 can do that by coding a space followed by a line-continuation character, which is an underscore (_). Starting with Visual Basic 2010, though, you can often continue a line without using the line-continuation character. Some of the code elements that allow implicit line continuation are listed in this figure.

To illustrate how this works, take a look at the two examples in this figure. Here, the first example uses line-continuation characters to continue the declaration for an event handler onto four lines. In contrast, the second example uses implicit line continuation. Here, the line-continuation characters that followed the first two lines of code in the first example have been omitted. The line-continuation character that follows the third line of code is still required, though.

By the way, don’t worry if you don’t understand most of the code elements listed in this figure. You’ll see how to use all of them later in this book. And when I present these elements, I’ll show you how to use implicit line continuation with them whenever that makes sense.
Coding rules for Visual Basic statements

- Use spaces to separate the words in each statement.
- To continue a statement to the next line, you can type a space followed by an underscore (the line-continuation character). In the cases listed below, you can continue a statement without using a line-continuation character.

Common code elements that allow implicit line continuation

- After a comma
- After an open parenthesis or before a close parenthesis
- After a dot operator and before the member name
- After a concatenation operator
- After an assignment operator
- After a binary operator
- After the Is and IsNot operators
- After an open curly brace or before a close curly brace
- After the In keyword in a For Each statement
- After the From keyword in a collection initializer
- Before and after query operators

Code that uses line-continuation characters

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

Code that uses implicit line continuation

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

Description

- *Implicit line continuation* lets you continue a statement onto two or more lines without using the line-continuation character. This feature became available with Visual Basic 2010.
- You can’t use implicit line continuation after a dot operator when it’s coded within a With statement or within an object initialization list.
The event handlers for the Invoice Total form

Figure 3-7 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.
The event handlers for the Invoice Total form

Public Class frmInvoiceTotal

Private Sub btnCalculate_Click(sender As Object, e As 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(sender As Object, e As EventArgs) Handles btnExit.Click

    Me.Close()

End Sub

End Class

Description

• When you double-click the Calculate and Exit buttons in the Form Designer, Visual Studio 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.
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 3-8. 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 continue long statements onto two or more lines so they’re 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(sender As Object, e As 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.
- Continue long lines of code onto additional lines 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 and the Basic group and then select the Advanced category.
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 3-9 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 code that has comments, be sure to change the comments too.

Incidentally, all comments are displayed in the Code Editor in green by default, which is different from the color of the words in the Visual Basic statements. That makes it easy to identify the comments.
A procedure with comments

Private Sub btnCalculate_Click(sender As Object, e As 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, including a line that uses implicit line continuation.
• 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 Out button in the Standard toolbar (see figure 3-11).
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 3-10, 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 Studio will display a description of the error. In addition, Visual Studio will display a link that you can click to list and apply potential fixes. You can also display this list by clicking the light bulb that appears. You’ll learn more about using light bulbs to correct syntax errors later in this chapter.

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 click 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.
- To display a list of potential fixes for an error, you can click the Show Potential Fixes link that’s displayed below the description of the error when you point to the error as shown above. You can also display this list by clicking the light bulb that appears below the error when you point to it. For more information on using light bulbs, see figure 3-15.
- You can highlight a potential fix in the list to preview the changes, and you can apply a fix by clicking on it or highlighting it and pressing Enter.
- 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 3-10  How to detect and correct syntax errors*
More 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 3-11.

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.

You can 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.

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 3-11. 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.
The Code Editor and the Text Editor toolbar

![Code Editor and Text Editor toolbar](image)

**How to use the Standard toolbar to comment out 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 move quickly between lines of code, you can use the last four buttons on the Text Editor toolbar to set, move between, and clear 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 its code is displayed.
**How to zoom in and out**

Visual Studio also provides the ability to zoom in and out of the code in the Code Editor. You may want to zoom into the code if it’s difficult to read. In contrast, you may want to zoom out of the code if you want to be able to see more of it at one time.

Figure 3-12 illustrates how the zoom feature works. Here, the Code Editor is zoomed in to 121% as indicated in the lower left corner of the window.

**How to highlight symbols**

Figure 3-12 also illustrates Visual Studio’s ability to highlight symbols in the Code Editor. Here, you can see that I clicked on the variable named discountPercent in the declaration for that variable. When I did that, all the occurrences of that variable in the code for the form were highlighted. This can help you easily see where and how a symbol is used.

In addition to variable names, you can highlight the names of classes, objects, properties, methods, and procedures. You can also highlight the keywords for a control structure. If I clicked on the If keyword in the If statement in this figure, for example, that keyword, along with the Then, ElseIf, Else, and End If keywords, would be highlighted.

**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. When the code is printed, any lines that extend beyond the width of the printed page are automatically wrapped to the next line.
How to zoom in and out of the Code Editor

- If you have a mouse with a scroll wheel, you can zoom in and out of the Code Editor by holding down the Ctrl key as you move the wheel forward and backward.
- Before you can zoom in or out, you must click in the Code Editor to move the focus to that window.
- The current zoom percent is displayed in the lower left corner of the window.
- If you have a touch screen, you can also use standard stretch and pinch gestures to zoom in and out.

How to highlight symbols

- If you click on a symbol in your code, all the occurrences of that symbol are highlighted.
- Symbols include the names of variables, classes, objects, properties, methods, and procedures and the keywords for control structures like the If statement.
- To move to the next highlighted symbol, press Ctrl+Shift+Down arrow. To move to the previous highlighted symbol, press Ctrl+Shift+Up arrow.
- To turn the highlighting feature off, use the Tools → Options command to display the Options dialog box. Then, expand the Text Editor node and the Basic node, click on the Advanced category, and deselect the Highlight References To Symbol Under Cursor and Highlight Related Keywords Under Cursor options.
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. Code 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 3-13, you can 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 common 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. To find out what the shortcut is for a code snippet, just highlight the snippet in the list of snippets like the one in the first screen in this figure. Then, a tool tip will be displayed to the right of the list that includes a brief description of the snippet along with its shortcut. To insert a code snippet using its shortcut, you just enter the shortcut and press the Tab key. You can also insert a snippet from a completion list by selecting the statement and then pressing the Tab key twice.

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. Instead, 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 default list of code snippets for the conditionals and loops group within the common code patterns 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, go to the group that the snippet is in and double-click the snippet to insert it. You can also insert a snippet by entering its shortcut and pressing the Tab key.

- You can also insert a code snippet by selecting an item from a completion list that has a code snippet and then pressing the Tab key twice.

- 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.
How to refactor code

As you work on the code for an application, you will often find that you want to revise your code. For example, you may want to change the name of an identifier, such as a variable, procedure, or class name, to make the name more meaningful and readable. However, if you change the name in one place, you need to change it throughout your code. This is known as refactoring, and Visual Studio 2015’s live code analysis feature makes it easy to refactor your code.

Figure 3-14 shows how you can use Visual Studio to quickly and easily change the names that you use within your code. In this figure, for example, the first screen shows the Code Editor after the Rename dialog box was displayed for one occurrence of the discountPercent variable and the name of the variable was changed to discountPct. Then, all occurrences of that variable were changed. This is referred to as inline renaming.

When you refactor a name like this, the Rename dialog box lets you choose if you want to include names that appear in comments and strings. It also lets you choose if you want to preview the changes before they’re applied. To apply the changes or to preview the changes if that option is selected, you can click the Apply button. Or, to cancel the changes, you can click the close button in the upper right corner of the dialog box.

If you select the Preview Changes option, a Preview Changes dialog box like the one shown here is displayed. This dialog box lets you review all of the changes that will be made. It also lets you deselect any changes that you don’t want to make.

Although this figure just shows how to change a name that’s used by the code, you can also use refactoring to modify the structure of your code by introducing constants and variables, extracting methods, and so on. To do that, you begin by selecting the code you want to refactor. Then, you can click Ctrl + period (.) to display a menu of possible actions. You’ll learn more about using refactoring to extract methods in chapter 6.

You can also use this technique to change a name instead of using inline renaming. For example, suppose you changed just the first occurrence of the discountPercent variable in this figure to discountPct. Then, if you placed the insertion point in that variable and pressed Ctrl + period, a menu would be displayed that lets you rename the other occurrences of the variable. Because using the Rename command is more intuitive and provides more options, though, we recommend you use it instead.

If you already have experience with another object-oriented language, these refactoring features should make sense to you. If not, don’t worry. You’ll learn more about these features as you progress through this book.
The options that are displayed when you rename a variable

The Preview Changes - Rename dialog box

Description

- The process of revising and restructuring existing code is known as refactoring. With Visual Studio 2015, refactoring is part of the live code analysis feature.

- To change the name of an identifier that’s used in your code, such as a variable, procedure, or class name, you can right-click on the name and select Rename from the shortcut menu that’s displayed. Then, enter the new name and click Apply in the dialog box that’s displayed.

- You can also use refactoring to modify the structure of your code by introducing constants or variables, extracting methods, and so on. To do that, you begin by selecting the code you want to refactor. Then, you press Ctrl + period (.) to display a menu of actions and select the appropriate refactoring action. You can also use this technique to change all occurrences of a name after you change one occurrence of the name.

- Some refactoring commands display a dialog box that lets you preview the changes before you make them. Then, you can deselect any changes that you don’t want to make.
How to perform quick actions with light bulbs

In figure 3-10, you learned the basic skills for detecting and correcting syntax errors. In many cases, though, you can also use quick actions to correct syntax errors. Figure 3-15 shows you how.

As you learned in figure 3-10, Visual Studio puts a wavy line under any syntax errors that it detects while you’re entering code. In most cases, though, Visual Studio takes that one step further by providing a Quick Actions menu that provides options for correcting an error. To display a Quick Actions menu, you point to an error with your mouse and then click the light bulb that appears. A light bulb also appears in the margin to the left of a line that contains an error if you click in that line.

In the example in this figure, you can see the light bulb and the Quick Actions menu that were displayed when I pointed to an error in the Code Editor and then clicked the light bulb. Here, the menu contains a single item that indicates that the error can be corrected by inserting a missing cast. If you point to this item, the existing code and the corrected code is displayed as shown here. To make the correction, you simply click the menu item.

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 a Quick Actions menu displayed

Description

- When Visual Studio detects a syntax error, it highlights the error with a wavy underline in the Code Editor.

- If you place the mouse pointer over an error and a light bulb appears, you can click on the light bulb to display a Quick Actions menu of the corrections you can make. If you point to a menu item, you can see the change that will be made. And if you click on a menu item, the change will be applied.

- If you click in a line that contains an error and a light bulb appears in the left margin, you can use the Quick Actions menu for that light bulb to apply corrections as described above. The only difference is that the menu will include corrections for all of the errors on that line.

- **Quick actions** are part of the live code analysis feature of Visual Studio 2015.

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 3-15   How to perform quick actions with light bulbs
How to use the annotations in the scroll bar

As you may have noticed, the vertical scroll bar in the Code Editor includes graphic indicators that Microsoft refers to as *annotations*. If you look back at figure 3-15, for example, you’ll see annotations that indicate the position of errors and the text cursor relative to the other code in the document. Annotations are also used to indicate lines with changes, bookmarks, and so on.

By default, the vertical scroll bar is displayed in *bar mode* so it looks like any other scroll bar. You can also display the scroll bar in *map mode* as shown in figure 3-16. Then, a minimized version of the code is displayed in the bar. In this example, the bar includes annotations for the text cursor, two changed lines, and two errors. You can click on any of these annotations to jump to the line of code in the Code Editor. That makes this feature particularly useful for working with long code files.

Another useful feature of map mode is that it lets you preview the code for an annotated line without having to move to that line in the Code Editor. To do that, you simply point to the annotation in the scroll bar. In this figure, for example, a preview is shown of the second error in the file.
A scroll bar in map mode with annotations

![Image of a scroll bar in map mode with annotations]

**Description**

- By default, the vertical scroll bar in the Code Editor is displayed in *bar mode*, and it contains *annotations* for lines with changes, errors, bookmarks, etc. as well as the current position of the text cursor.

- Annotations are most useful with long code files. Then, you can quickly move to a line of code that contains an annotation by clicking on it in the scroll bar.

- You can also display the scroll bar in *map mode* as shown above. Then, a minimized version of the code is displayed by default, and you can preview the code at a specific location by pointing to it with the mouse. That way, you can display the code without having to scroll to it.

- To change to map mode, display the options dialog box (**Tools** → **Options**), expand the Text Editor and Basic groups, and select the Use Map Mode option in the Scroll Bars category. You can also select the width of the bar from the Source Overview drop-down list.

---

*Figure 3-16  How to use the annotations in the scroll bar*
How to use the My feature

The *My feature* 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 3-17 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 statements in this figure illustrate how you can use two of these objects. To get more information about any of these objects, you can use online help as described in the next figure.

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 main My objects for Windows Forms applications

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

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

A statement that checks if a directory exists

```vbnet
If My.Computer.FileSystem.DirectoryExists("C:\VB 2015\Files") Then ...
```

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.
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, or event, or some other aspect of Visual Basic programming. Figure 3-18 shows how to use Microsoft’s MSDN Online Library to 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 displays the available information about the selected keyword or object in your default browser. Another way to display the help information is to select the View Help command from Visual Studio’s Help menu. Then, you can use the full-text search feature at the top of the window or the table of contents in the left pane of the window to locate and display the information you need.

To display the topic shown in this figure, for example, I entered “writing code” into the Search text box and then pressed the Enter key. When I did that, a list of topics that contain the search text was displayed. Then, I clicked on the link for the “Writing Code in the Code and Text Editor” topic to display the information shown here.

The table of contents that’s displayed in the left pane of this window includes ancestors of the current topic and the current topic and its peers. You can click on any of these topics to display the information for that topic in the center pane. You can also click on any of the links in the right pane to jump to different parts of the help topic.

Note that because the help information is displayed in your web browser, you can work with it just as you would any other web page. To jump to a related topic, for example, you can click on a link in the current topic. 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 the help information.

In addition to using online help, you can install help locally. By default, some basic information about Visual Studio and the .NET Framework is installed locally when you install Visual Studio. To display this local help, you use the Help Viewer. To open the Help Viewer instead of your default browser when you display help information, you set the help preference as described in this figure.

You can also use the Help Viewer to manage your local help content. To do that, you display the Manage Content tab of the Help Viewer. This tab lists the available help documentation by category and indicates if it’s installed locally. Then, you can click the Add or Remove link for a category to add or remove the documentation for that category from local help.

One advantage of using local help is that it’s always available even if you don’t have access to the Internet. On the other hand, local help may not always be up-to-date like online help is. Because of that, you may want to occasionally update your local help.
Online help for Visual Studio

Description

- Visual Studio lets you display help content from Microsoft’s MSDN Online Library in your default web browser.
- You can display context-sensitive help 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 Visual Studio online help by selecting the View Help command from Visual Studio’s Help menu.
- Online help is divided into three panes. The left pane displays the table of contents, the center pane displays the last help topic that you accessed, and the right pane contains an outline of the help topic so you can jump to different parts of the topic.
- Controls for performing a full-text search appear in the upper right corner of the help window. To use full-text search, click on the magnifying glass, enter the text into the text box that appears, and then press the Enter key. A list of topics that contain the search text will be displayed, and you can click on the topic you want to display.
- To use the table of contents, click on a topic to display it in the center pane. Ancestors of the topic, the topic’s peers, and peers of the topic’s parent topic are displayed in the left pane.
- To display help content that resides on your local system, you use the Help Viewer. To do that, select the Set Help Preference → Launch in Help Viewer command from the Help menu.
- You can also use the Manage Content tab of the Help Viewer to add or remove content from local help.

Figure 3-18    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 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 3-19. 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 as described earlier in this chapter. 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.

Notice in this figure that when you run a project, the Diagnostic Tools window is displayed by default. This window provides a variety of information, including memory and CPU usage. Because you won’t typically need this information, you may want to set the option that will keep it from being displayed as described in this figure.
The form that’s displayed when you run the Invoice Total project

![Image of the Invoice Total form]

**Description**

- To *run* a project, click the Start 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.

- When you run a project, the Diagnostic Tools window is displayed by default. This window lists events that have occurred during the session, and it lets you monitor memory and CPU usage. If you don’t want this window to be displayed, you can display the Options dialog box (Tools ➔ Options), expand the Debugging group, select the General category, and remove the check mark from the Enable Diagnostic Tools option.

- 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.
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 3-20 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

![Exception Assistant window]

**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.
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 3-21. 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 3-20. 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 four 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 gives 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 3-21    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 still have 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

<table>
<thead>
<tr>
<th>object-oriented programming</th>
<th>application</th>
<th>refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>object-oriented language</td>
<td>event handler</td>
<td>live code analysis</td>
</tr>
<tr>
<td>object</td>
<td>procedure declaration</td>
<td>inline renaming</td>
</tr>
<tr>
<td>object</td>
<td>procedure name</td>
<td>quick renaming</td>
</tr>
<tr>
<td>class</td>
<td>completion list</td>
<td>annotation</td>
</tr>
<tr>
<td>instance</td>
<td>tool tip</td>
<td>bar mode</td>
</tr>
<tr>
<td>instantiation</td>
<td>line-continuation</td>
<td>map mode</td>
</tr>
<tr>
<td>property</td>
<td>character</td>
<td>My feature</td>
</tr>
<tr>
<td>method</td>
<td>comment</td>
<td>build a project</td>
</tr>
<tr>
<td>event</td>
<td>syntax error</td>
<td>run a project</td>
</tr>
<tr>
<td>member</td>
<td>build error</td>
<td>test a project</td>
</tr>
<tr>
<td>dot operator</td>
<td>comment out a line</td>
<td>runtime error</td>
</tr>
<tr>
<td>dot</td>
<td>bookmark</td>
<td>exception</td>
</tr>
<tr>
<td>IntelliSense</td>
<td>collapse</td>
<td>bug</td>
</tr>
<tr>
<td>argument</td>
<td>expand</td>
<td>debug</td>
</tr>
<tr>
<td>event-driven</td>
<td>code snippet</td>
<td>break mode</td>
</tr>
<tr>
<td></td>
<td>identifier</td>
<td>data tip</td>
</tr>
</tbody>
</table>

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 2015\Chapter 02 directory to the C:\VB 2015\Chapter 03 directory.

2. Open the Invoice Total solution (InvoiceTotal.sln) that’s now in the C:\VB 2015\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 3-7. 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 on the Exit button to generate the procedure declaration for the Click event of this object. Enter the statement shown in figure 3-7 for this event handler.
5. Open the Error List window as described in figure 3-10. 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 Quick Actions 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. Click the drop-down arrow in the data tip 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 the Standard toolbar to end the application.

Experiment with the Visual Studio features
11. Right-click on the name of the discountPercent variable and select the Rename command from the shortcut menu that’s displayed. When the Rename dialog box appears, enter the name discountPct and notice that all occurrences of the variable are changed.
12. If the Preview Changes option in the Rename dialog box is selected, deselect it. Then, click the Apply button or press the Enter key to apply the changes. Now, run the form to make sure it still works correctly.

13. Select the lines that contain the ElseIf clauses and click on the Comment Out button in the Standard toolbar. 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 again so you can see its code.

15. Click on the If keyword in the btnCalculate_Click procedure, and notice that this keyword, along with the Then, ElseIf, Else, and End If keywords are highlighted. Then, click on any occurrence of txtSubtotal to see that all occurrences are highlighted.

16. If your mouse has a scroll wheel, hold down the Ctrl key and scroll the wheel forward to zoom into the code in the Code Editor window. Then, scroll it backward to zoom back out.

17. 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**

18. 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 online help in your default browser and display a topic that tells you more about this method.

19. Click the magnifying glass at the top of the window, type “quick actions” into the Search text box, and then press the Enter key to see the entries that are listed in the center pane. Click on the “Perform quick actions with light bulbs” topic to display it.

20. Click the last link in the right pane to jump to that part of the topic. Then, click on one or more topics in the table of contents to display them.

**Exit from Visual Studio**

21. 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!
How to build your VB programming skills

The easiest way is to let Murach’s Visual Basic 2015 be your guide! So if you’ve enjoyed this chapter, I hope you’ll get your own copy of the book today. You can use it to:

- Teach yourself how to code Windows Forms applications in VB
- Take advantage of all the time- and work-saving features of Visual Studio 2015 as you develop VB applications
- Build database applications using RAD features like data sources and the DataGridView control…and start using features like ADO.NET code, LINQ, and the Entity Framework as well
- Use object-oriented programming techniques the way the pros do
- Pick up a new skill whenever you want or need to by focusing on material that’s new to you
- Look up coding details or refresh your memory on forgotten details when you’re in the middle of developing a VB application
- Loan to your colleagues who will be asking you more and more questions about VB programming

To get your copy, you can order online at www.murach.com or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you’ve ever tried, or send it back within 60 days for a full refund…no questions asked.

Thanks for your interest in Murach books!