

TRAINING & REFERENCE

murach's **C#**

7TH EDITION

(Chapter 3)

Thanks for downloading this chapter from [Murach's C# \(7th Edition\)](#). 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!



MIKE MURACH & ASSOCIATES, INC.

1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963

murachbooks@murach.com • www.murach.com

Copyright © 2021 Mike Murach & Associates. All rights reserved.

What C# developers have said about previous editions

“I personally think that Microsoft should just hand over all documentation for their technologies to Murach! You make it fun, intuitive, and interesting to learn.”

Joanne Wood, C# Web Developer, Rhode Island

“I am actually flying through the C# book! And a lot of the topics I had problems with in the past are now making perfect sense in this book.”

Jim Bonner, Test Lab Engineer, Washington

“In 20+ years of software development, I have never read another book so well organized. Topics flow in a well-thought-out, logical order and are easy to follow and understand. When I need new technical books in the future, I will always search for a Murach book first.”

Jeff Ramage, Lower Alabama .NET User Group

“Best C# book ever, precise and to the point, with lots and lots of examples. Highly recommended for beginners.”

Posted at an online bookseller

“Murach has yet again hit a home run! This book gets you from zero to Jr Dev knowledge in no time.”

Brian Knight, Founder, Pragmatic Works, Florida

“I have to tell you that your C# book is far and away the best resource I have seen to date. It is simple, straightforward, presents logical examples, and the two-page format is the best.”

Timothy Layton, Developer, Missouri

“I have learned more in the 800+ pages of this book than I have reading half a dozen other books and hours of forum posts online.”

Posted at an online bookseller

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. When you're done, you'll be able to develop simple applications of your own.

An introduction to coding	56
Introduction to object-oriented programming.....	56
How to refer to properties, methods, and events	58
How an application responds to events.....	60
How to add code to a form	62
How to create an event handler for the default event of a form or control....	62
How to delete an event handler.....	62
How IntelliSense helps you enter the code for a form.....	64
The event handlers for the Invoice Total form	66
How to detect and correct syntax errors.....	68
More coding skills	70
How to code with a readable style.....	70
How to code comments	72
How to work with the Text Editor toolbar	74
How to collapse or expand blocks of code	74
How to use code snippets	76
How to refactor code.....	78
How to get help information	78
How to run, test, and debug a project.....	80
How to run a project	80
How to test a project	82
How to debug runtime errors.....	84
Perspective	86

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 libraries.

When you start a new project from the Windows Forms App template, you are actually creating a new *class* that inherits the characteristics of the Form class that's part of the Windows.Forms 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*.

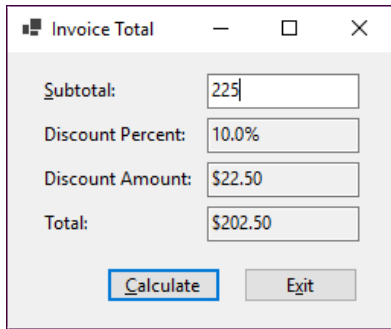
As you progress through this book, you will learn much more about classes and objects because C# is an *object-oriented language*. In chapter 12, for example, you'll learn how to use the C# language to create your own classes. At that point, you'll start to understand what's actually happening as you 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. By 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 C# method 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



The screenshot shows a Windows Forms application window titled "Invoice Total". Inside the window, there are four text boxes arranged vertically. The first text box is labeled "Subtotal:" and contains the text "225". The second text box is labeled "Discount Percent:" and contains "10.0%". The third text box is labeled "Discount Amount:" and contains "\$22.50". The fourth text box is labeled "Total:" and contains "\$202.50". Below these text boxes, there are two buttons: "Calculate" and "Exit".

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 C# 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 C# 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 addition to referring to the properties, methods, and events of objects, you can also refer to some of the properties and methods of a class directly from that class. The code shown in the Code Editor in this figure, for example, refers to the `ToDecimal()` method of the `Convert` class. A property or method that you can refer to directly from a class like this is called a *static member*. You'll learn more about static members in chapter 4. For now, you just need to realize that you can refer to static properties and methods using the same techniques that you use to refer to the properties and methods of an object.

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 object or class after you type the object or class 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.

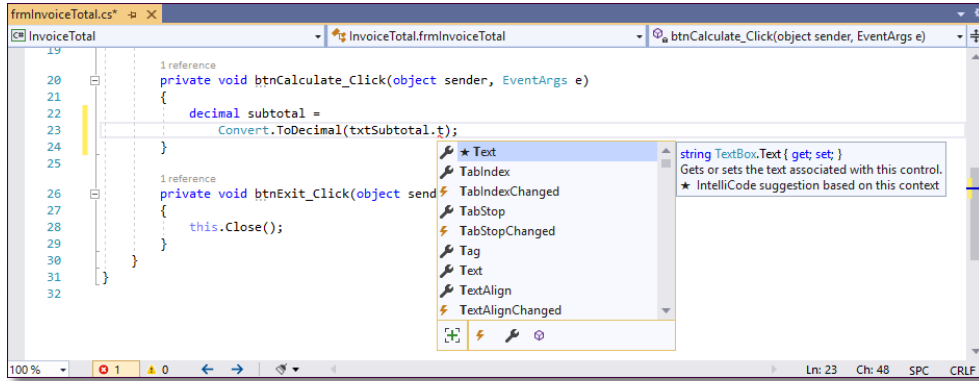
To give you an idea of how properties, methods, and events are used in code, this figure shows examples of each. In the first example for properties, code is used to set the value that's displayed for a text box to 10. In the second example, code is used to set the `ReadOnly` property of a text box to 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 `Focus()` 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 *this* keyword is used instead of the name of the form. Here, *this* refers to the current instance of the active form. Note that the names of the methods are followed by parentheses.

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 and that you must code a set of parentheses after the method.

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 a string holding the number 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.Focus();	Uses the Focus() method to move the focus to the text box named txtMonthlyInvestment.
this.Close();	Uses the Close() method to close the form that contains the statement. In this example, <i>this</i> is a keyword that is used to refer to the current instance of the 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, you can type one or more letters of the member name, and the Code Editor will select the first entry in the list that matches those letters. Or, you can scroll down 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 C# category, 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 special type of method known as an *event handler*. When you do that, Visual Studio generates a statement that connects, or wires, the event handler to the event. This is called *event wiring*, and it's illustrated in figure 3-3.

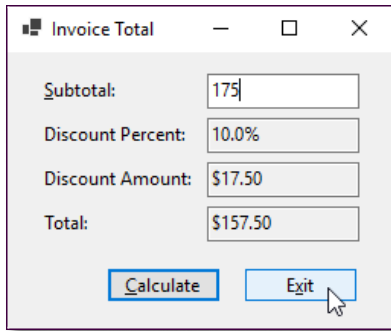
In this figure, the user clicks the Exit button on the Invoice Total form. Then, Visual Studio uses the statement it generated to wire the event to determine what event handler to execute in response to the event. In this case, the `btnExit.Click` event is wired to the method named `btnExit_Click`, so this method is executed. As you can see, 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 presented 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



Wiring: The application determines what method to execute

```
this.btnExit.Click += new System.EventHandler(this.btnExit_Click);
```

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

```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

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 special type of method that handles the event.
- To connect the event handler to the event, Visual Studio automatically generates a statement that wires the event to the event handler. This is known as *event wiring*.
- 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 programming, you're ready to learn how to add code to a form. Because you'll learn the essentials of the C# 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 events of forms and controls. 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. Then, Visual Studio opens the Code Editor, generates a *method declaration* for the default event of the object, and places the insertion point on a blank line between the opening and closing braces of that declaration. As a result, you can immediately start typing the C# statements that you want to include in the body of the method.

To illustrate, figure 3-4 shows the code that was generated when I double-clicked the Calculate button on the Invoice Total form. In this figure, the code for the form is stored in a file named `frmInvoiceTotal.cs`. In addition, the name of the method is the name of the object (`btnCalculate`), an underscore, and the name of the event (`Click`). The statement that wires the `Click` event of this button to this event handler is stored in the file named `frmInvoiceTotal.Designer.cs`.

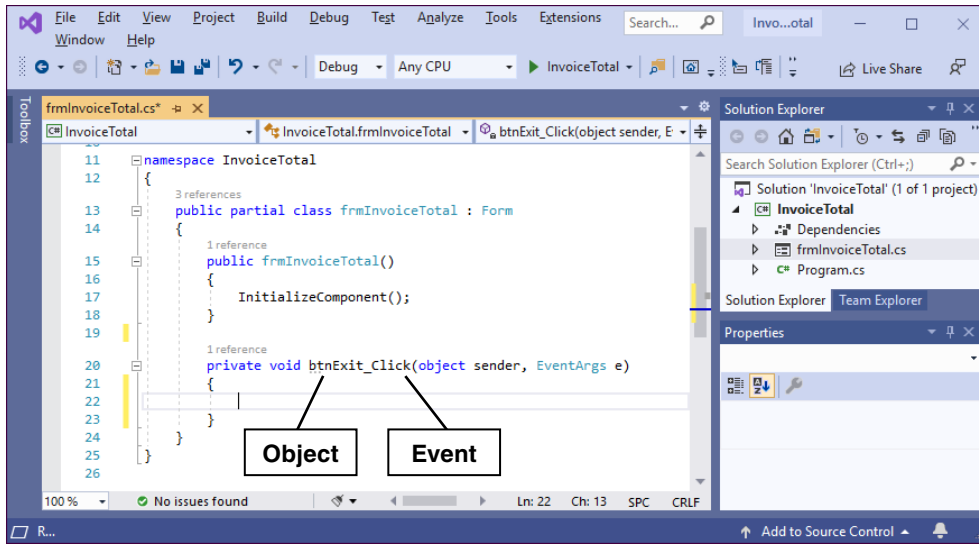
Before you start an event handler for a control, you should set the `Name` property of the control as described in chapter 2. That way, this name will be reflected in the *method name* of the event handler as shown in this figure. If you change the control name after starting an event handler for it, Visual Studio will change the name of the object in the event wiring, but it won't change the name of the object in the method name. And that can be confusing when you're first learning C#.

You should also avoid modifying the method declaration that's generated for you when you create an event handler. In chapter 6, you'll learn how to modify the method declaration. But for now, you should leave the method declaration alone and focus on adding code within the body of the method.

How to delete an event handler

If you add an event handler by mistake, you can't just delete it. If you do, you'll get an error when you try to run the application. This error will be displayed in an Error List window as shown in figure 3-7, and it will indicate that the event handler is missing.

The method 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 button. This opens the Code Editor, generates the declaration for the method that handles the event, and places the cursor within this declaration.
2. Type the C# code between the opening brace (`{`) and the closing brace (`}`) of the method declaration.
3. When you are finished writing 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 *method declaration* for the event handler that's generated when you double-click on an object in the Form Designer includes a *method name* that consists of the object name, an underscore, and the event name. The event handler is stored in the cs file for the form.
- Most of the code that's generated when you design a form, including the statement that wires the event to the event handler, is stored in the Designer.cs file for the form. If necessary, you can open this file to view or delete the event wiring.
- 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

That's because when you create an event handler, Visual Studio also generates a statement that wires the event to the event handler. As a result, if you delete an event handler, you must also delete the wiring for the event. The easiest way to do that is to double-click on the error message in the Error List window. This will open the Designer.cs file for the form and jump to the statement that contains the wiring for the missing event handler. Then, you can delete this statement.

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 method as shown in chapters 6 and 12. And it can help you enter keywords and data types, 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 *i*, the list is positioned on the last item I used that begins with that letter. In this case, it's positioned on the `if` keyword. As described earlier in this chapter, you can enter as many letters as you want, and Visual Studio 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.

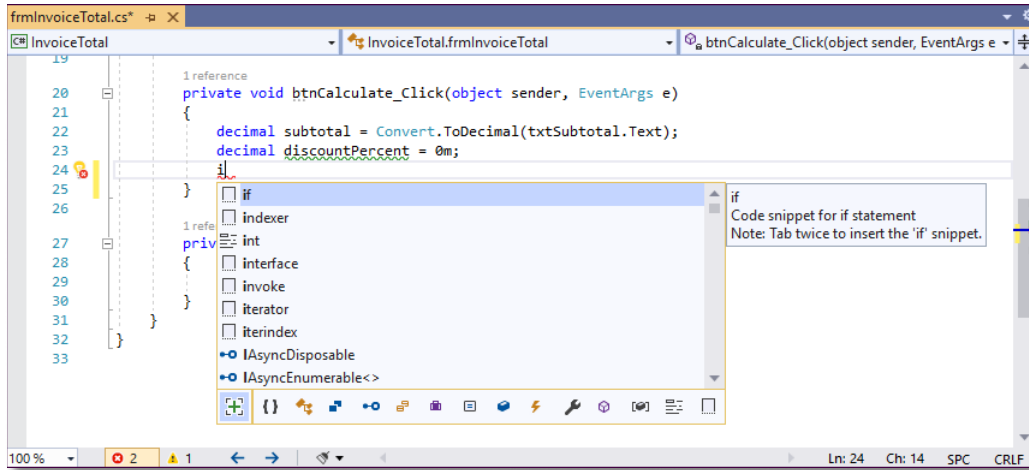
When you select an item in a list, Visual Studio displays information about that item in a *tool tip*. For example, the tool tip for the `if` keyword indicates that there is a code snippet available for the `if` statement and that you can insert it by pressing the Tab key twice. You'll learn more about using code snippets later in this chapter.

The second example in this figure shows the completion list that was displayed after I inserted the `if` keyword and then typed a space, an opening parenthesis, and the initial text `su`. That made it easy to enter the variable named `subtotal` into the code, and IntelliSense added the closing parenthesis automatically.

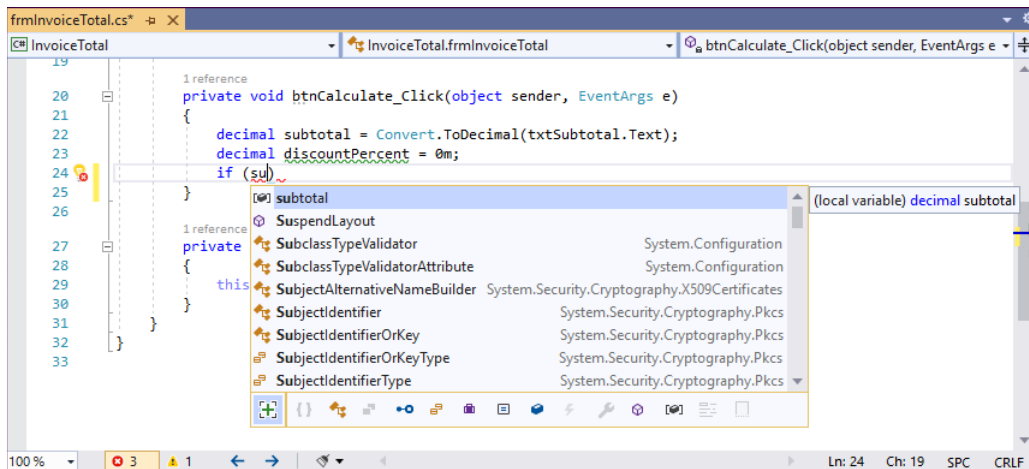
If you use these IntelliSense features, you'll see that they can help you avoid introducing errors into your code. For example, it's easy to forget the names you've given to items such as controls and variables, so the list that's displayed can help you locate the appropriate name.

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 the Ctrl key and the list is hidden until you release that key.

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



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



Description

- The IntelliSense that's provided for C# lists keywords, data types, variables, objects, and classes 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 completion list without closing the list, press the Ctrl key. Then, the list is hidden until you release the Ctrl key.
- If you enter an opening parenthesis or brace, the closing parenthesis or brace is added automatically.

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.

By 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.cs 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 one exception is deleting unnecessary event wiring statements.

When you enter C# code, you must be aware of the coding rules summarized in this figure. In particular, note that each method contains a *block* of code that's enclosed in braces. As you'll see throughout this book, braces are used frequently in C# to identify blocks of code. Also, note that each *statement* ends with a semicolon. This is true even if the statement spans several lines of code.

You should also realize that C# is a case-sensitive language. As a result, you must use exact capitalization for all C# keywords, class names, object names, variable names, and so on. If you use IntelliSense to help you enter your code, this shouldn't be a problem.

The event handlers for the Invoice Total form

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);
    decimal discountPercent = 0m;
    if (subtotal >= 500)
    {
        discountPercent = .2m;
    }
    else if (subtotal >= 250 && subtotal < 500)
    {
        discountPercent = .15m;
    }
    else if (subtotal >= 100 && subtotal < 250)
    {
        discountPercent = .1m;
    }

    decimal discountAmount = subtotal * discountPercent;
    decimal invoiceTotal = subtotal - discountAmount;

    txtDiscountPercent.Text = discountPercent.ToString("p1");
    txtDiscountAmount.Text = discountAmount.ToString("c");
    txtTotal.Text = invoiceTotal.ToString("c");

    txtSubtotal.Focus();
}

private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Coding rules

- Use spaces to separate the words in each statement.
- Use exact capitalization for all keywords, class names, object names, variable names, etc.
- End each *statement* with a semicolon.
- Each *block* of code must be enclosed in braces (`{ }`). That includes the block of code that defines the body of a method.

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 method 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 method closes the form, which ends the application.

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

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 3-7, for example, you can see the lines under `txtPercent` and `txtAmount`.

If you place the mouse pointer over the code in error, a brief description of the error is displayed. In this case, the error message indicates that the name does not exist. That's because the names entered in the Code Editor don't match the names used by the Form Designer. If the names are correct in the Form Designer, you can easily correct these errors by editing the names in the Code Editor. In this figure, for example, the names of the text boxes should be `txtDiscountPercent` and `txtDiscountAmount`.

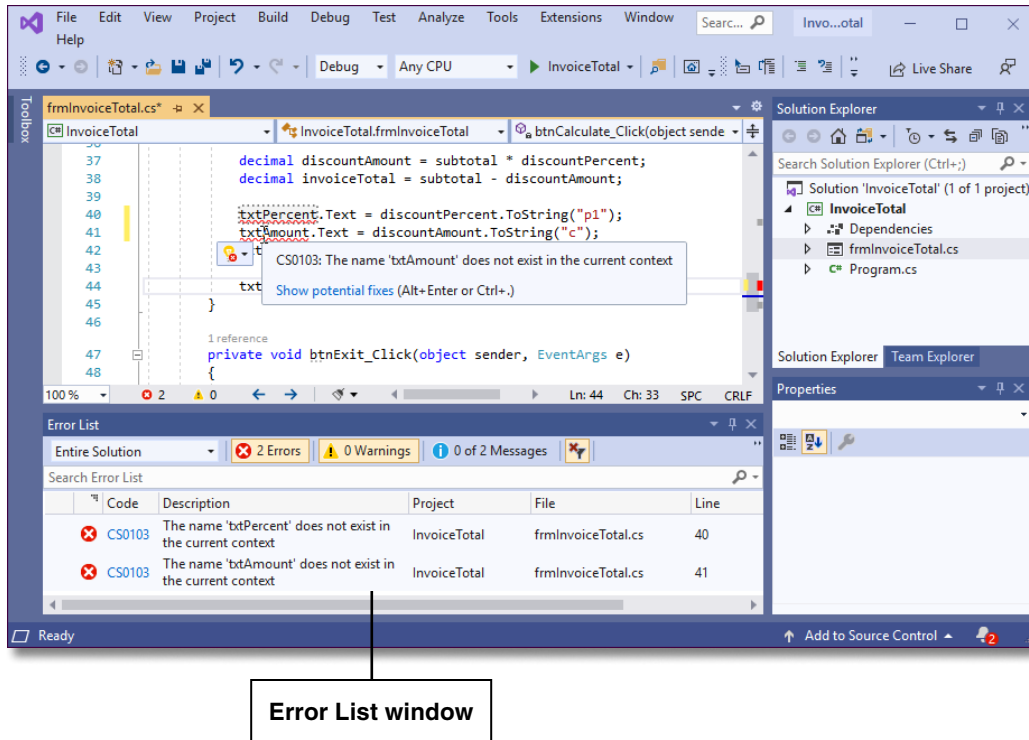
If the Error List window is open as shown in this figure, any errors that Visual Studio detects will also be displayed in that window. If the Error List window isn't open, you can display it using the **View**→**Error List** command. Then, you can jump to the error in the Code Editor by double-clicking on it in the Error List window.

When you're first getting started with C#, you will inevitably encounter a lot of errors. As a result, you may want to keep the Error List window open all the time. This makes it easy to see errors as soon as they occur. Then, once you get the hang of working with C#, you can conserve screen space by using the Auto Hide button so this window is only displayed when you click the Error List tab at the lower edge of the screen.

If you need additional help determining the cause of a syntax error, you can use the *live code analysis* feature. To use this feature, you can click the light bulb or link that appears when you place the mouse pointer over the error to display a list of potential fixes. Then, you can highlight a fix to preview the changes that will be made, and you can press the Enter key to apply the fix. Alternatively, you can click on a fix to apply it.

By the way, Visual Studio isn't able to detect all syntax errors as you enter code. Instead, 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 C# 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. Then, if you want to hide this window, you can click its Auto Hide button.
- 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. You can also display this list by clicking the light bulb that appears below the error when you point to it or in the margin to the left of the error when you click in it. This is part of a feature called *live code analysis*.
- 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 the project.

Figure 3-7 How to detect and correct syntax errors

More coding skills

At this point, you should understand the mechanics of adding code to a form. To code effectively, however, you'll need some additional skills. The topics that follow present some of the most useful coding skills.

How to code with a readable style

In figure 3-6, you learned some coding rules that you must follow when you enter the code for an application. If you don't, Visual Studio reports syntax errors that you have to correct before you can continue. You saw how that worked in the last figure.

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 three coding recommendations presented in figure 3-8.

To illustrate, this figure presents two versions of an event handler. Both versions accomplish the same task. As you can see, however, the first one is easier to read than the second one because it follows our coding recommendations.

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, tabs, or returns to separate the elements in a C# statement. In this example, all of the statements within the event handler are indented. In addition, the if-else statements 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. In the less readable code example, you can see that each line of code except for the method declaration includes at least one operator. Because the operators aren't separated from the word or value on each side of the operator, the code is difficult to read. By 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 method in this figure. Here, the code is separated into five groups of statements. In a short method like this one, this isn't too important, but it can make a long method much easier to follow.

Throughout this 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.

As you enter code, the Code Editor will automatically assist you in formatting your code. 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. Although you can change how this works using the Options dialog box, you probably won't want to do that.

A method written in a readable style

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);

    decimal discountPercent = 0m;
    if (subtotal >= 500)
    {
        discountPercent = .2m;
    }
    else if (subtotal >= 250 && subtotal < 500)
    {
        discountPercent = .15m;
    }
    else if (subtotal >= 100 && subtotal < 250)
    {
        discountPercent = .1m;
    }

    decimal discountAmount = subtotal * discountPercent;
    decimal invoiceTotal = subtotal - discountAmount;

    txtDiscountPercent.Text = discountPercent.ToString("p1");
    txtDiscountAmount.Text = discountAmount.ToString("c");
    txtTotal.Text = invoiceTotal.ToString("c");

    txtSubtotal.Focus();
}
```

A method written in a less readable style

```
private void btnCalculate_Click(object sender, EventArgs e){
decimal subtotal=Convert.ToDecimal(txtSubtotal.Text);
decimal discountPercent=0m;
if (subtotal>=500) discountPercent=.2m;
else if (subtotal>=250&&subtotal<500) discountPercent=.15m;
else if (subtotal>=100&&subtotal<250) discountPercent=.1m;
decimal discountAmount=subtotal*discountPercent;
decimal invoiceTotal=subtotal-discountAmount;
txtDiscountPercent.Text=discountPercent.ToString("p1");
txtDiscountAmount.Text=discountAmount.ToString("c");
txtTotal.Text=invoiceTotal.ToString("c");txtSubtotal.Focus();}
```

Coding recommendations

- Use indentation and extra spaces to align statements and blocks of code 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.

Note

- As you enter code in the Code Editor, Visual Studio automatically adjusts its formatting by default.

How to code comments

Comments are used to document what the program does and what specific blocks and lines of code do. Since the C# compiler ignores comments, you can include them anywhere in a program without affecting your code. Figure 3-9 shows you how to code two types of comments.

First, this figure shows a *delimited comment* at the start of a method. This type of comment is typically used to document information that applies to an entire method or to any other large block of code. You can include any useful or helpful information in a delimited comment such as a general description of the block, the author's name, the completion date, the files used by the block, and so on.

To document the purpose of a single line of code or a block of code, you can use *single-line comments*. Once the compiler reads the slashes (*//*) that start this type of comment, it ignores all characters until the end of the current line. In this figure, single-line comments have been used to describe each group of statements. In addition, single-line comments have been used at the end of some lines of code to clarify the code.

Although many 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 C# 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 C# statements. That makes it easy to identify the comments.

A method with comments

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    /******
    * this method calculates the total
    * for an invoice depending on a
    * discount that's based on the subtotal
    * *****/

    // get the subtotal amount from the Subtotal text box
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);

    // set the discountPercent variable based
    // on the value of the subtotal variable
    decimal discountPercent = 0m;           // the m indicates a decimal value
    if (subtotal >= 500)
    {
        discountPercent = .2m;
    }
    else if (subtotal >= 250 && subtotal < 500)
    {
        discountPercent = .15m;
    }
    else if (subtotal >= 100 && subtotal < 250)
    {
        discountPercent = .1m;
    }

    // calculate and assign the values for the
    // discountAmount and invoiceTotal variables
    decimal discountAmount = subtotal * discountPercent;
    decimal invoiceTotal = subtotal - discountAmount;

    // format the values and display them in their text boxes
    txtDiscountPercent.Text =           // percent format
        discountPercent.ToString("p1"); // with 1 decimal place
    txtDiscountAmount.Text =
        discountAmount.ToString("c");   // currency format
    txtTotal.Text =
        invoiceTotal.ToString("c");

    // move the focus to the Subtotal text box
    txtSubtotal.Focus();
}
```

Description

- *Comments* are used to help document what a program does and what the code within it does.
- To code a *single-line comment*, type `//` before the comment. You can use this technique to add a comment on its own line or to add a comment at the end of a line.
- To code a *delimited comment*, type `/*` at the start of the comment and `*/` at the end. You can also code asterisks to identify the lines in the comment, but that isn't necessary.

Figure 3-9 How to code comments

How to work with the Text Editor toolbar

Figure 3-10 shows how you can use the Text Editor toolbar to work with code. If you experiment with this toolbar, you'll find that its buttons provide some useful functions for working with comments and for moving from one place to another.

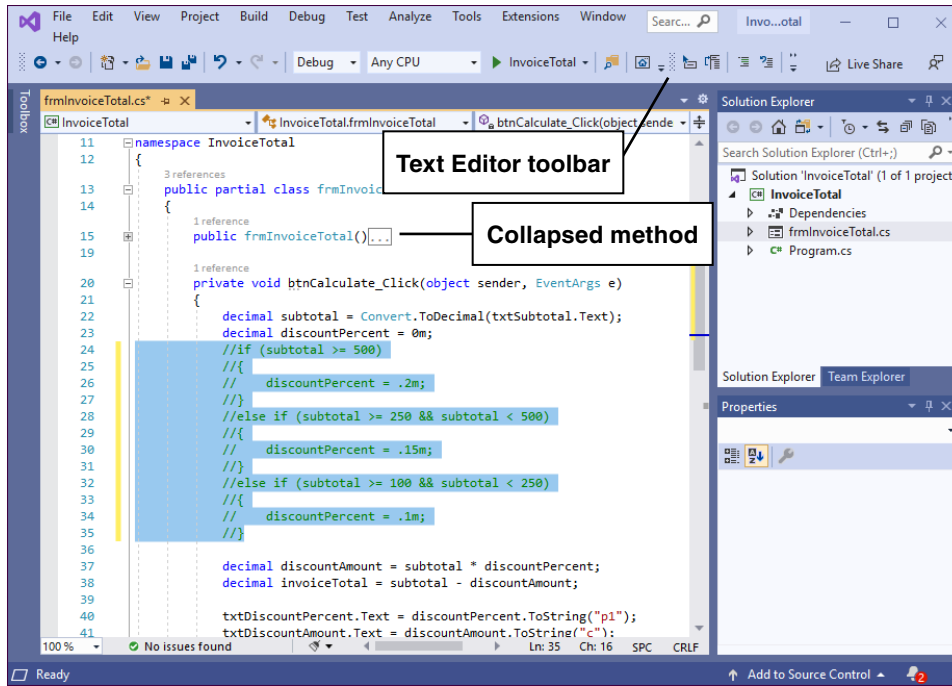
In particular, you can use the Text Editor toolbar to *comment out* several lines of code during testing by selecting the lines of code and then clicking on the Comment Out button. Then, you can test the program without those lines of code. If necessary, you can use the Uncomment button to restore those lines of code.

You can also use the Text Editor toolbar to work with *bookmarks*. After you use the Toggle Bookmark button to mark lines of code, you can easily move between the marked lines of code by using the Next and Previous buttons. Although you usually don't need bookmarks when you're working with simple applications like the one shown here, bookmarks can be helpful when you're working with applications that contain more than a few pages of code.

How to collapse or expand blocks of code

As you write the code for an application, you may want to *collapse* or *expand* some of the regions, comments, and methods to make it easier to scroll through the code and locate specific sections of code. To do that, you can use the techniques described in figure 3-10. In this figure, for example, the frmInvoiceTotal method has been collapsed so all you can see is its method declaration.

The Code Editor and the Text Editor toolbar



How to use the buttons of 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 comment or uncomment several lines of code, select the lines and click the Comment Out or Uncomment button. During testing, you can *comment out* lines of code so they won't be executed. That way, you can test new statements without deleting the old statements.
- To move quickly between lines of code, you can use the last four 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 its code is displayed.

Figure 3-10 How to use the Text Editor toolbar 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*. Code snippets make it easy to enter common control structures like the ones that you'll learn about in chapter 5.

Sometimes, you'll want to insert a code snippet on a blank line of text as shown in figure 3-11. The easiest way to do that is to select an item from a completion list and press the Tab key twice. In this figure, for example, the first screen shows the completion list for the if keyword (not the #if snippet farther down in the list). This screen includes a tool tip that indicates that the if statement has a code snippet and that you can press the Tab key twice to insert that snippet.

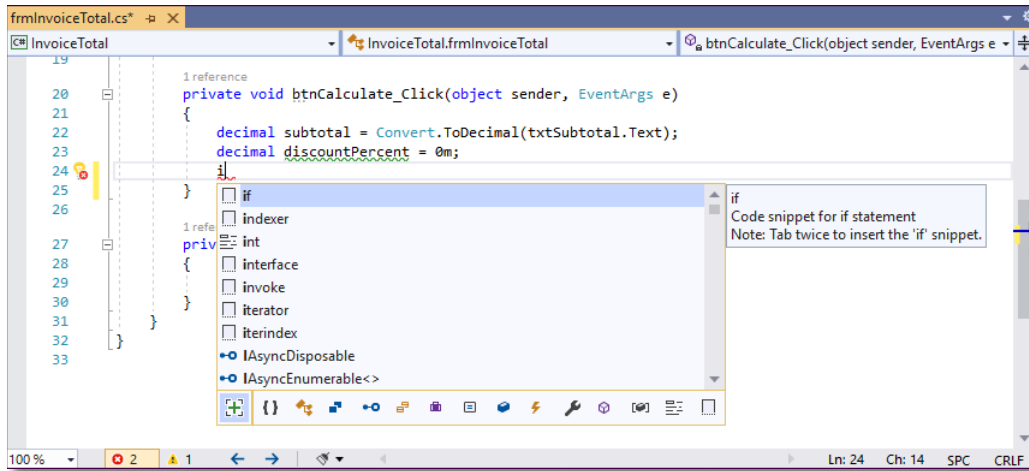
If you press the Tab key twice, Visual Studio inserts a snippet that contains the start of an if block as shown by the second screen. That way, you can enter a condition within the parentheses, and you can enter some statements between the braces.

Other times, you'll want to surround existing lines of code with a code snippet. In that case, you can select the code that you want to surround, right-click on that code, and select the Surround With command from the resulting menu. Then, you can select the appropriate snippet. For example, you might want to add an if block around one or more existing statements.

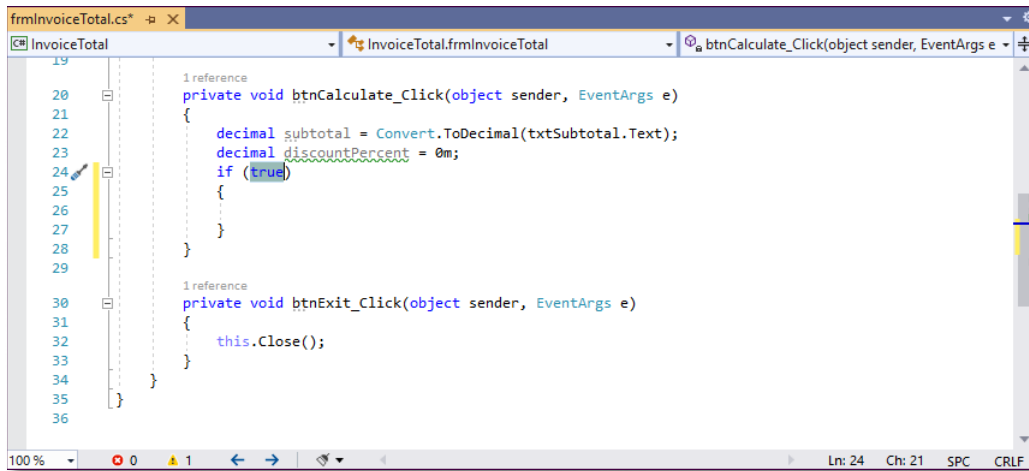
If you find that you like using code snippets, you should be aware that it's possible to add or remove snippets from the default list. To do that, you can choose the Code Snippets Manager command from the Tools menu. Then, you can use the resulting dialog box to remove code snippets that you don't use 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 completion list for the if keyword



The if snippet after it has been inserted



Description

- To insert a code snippet, select an item from a completion list that has a code snippet and then press the Tab key twice.
- To surround existing code with a code snippet, select the code, right-click on it, and select the Surround With command from the resulting menu. Then, select the appropriate snippet.
- 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 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 a name that you've used to identify a variable in your code 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's live code analysis feature makes it easy to refactor your code.

Figure 3-12 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 one occurrence of the subtotal variable was selected, the Rename dialog box was displayed, and the name of the variable was changed to total. 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. To do that, you begin by selecting the code you want to refactor. Then, you can click Ctrl + period (.) to display a light bulb with a menu of possible actions in the left margin. For example, you can use refactoring to extract methods as shown in chapter 6.

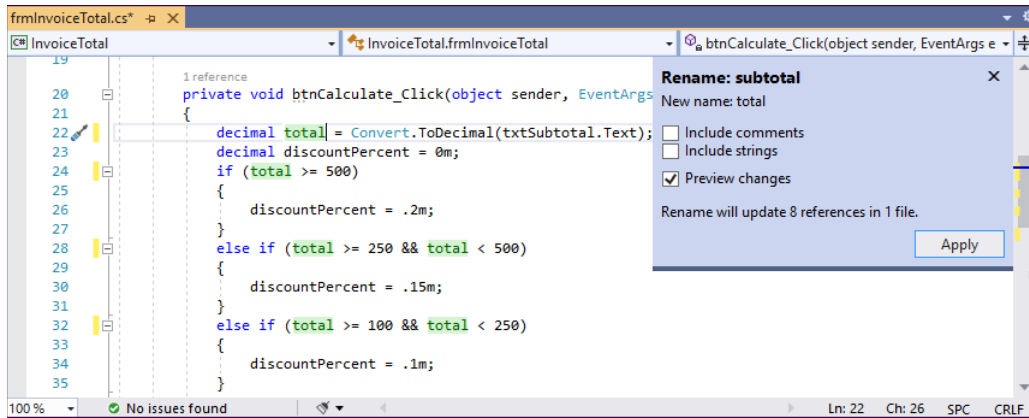
If you already have experience with another object-oriented language, these refactoring features should make sense to you. If not, don't worry. You'll learn more about these features as you progress through this book.

How to get help information

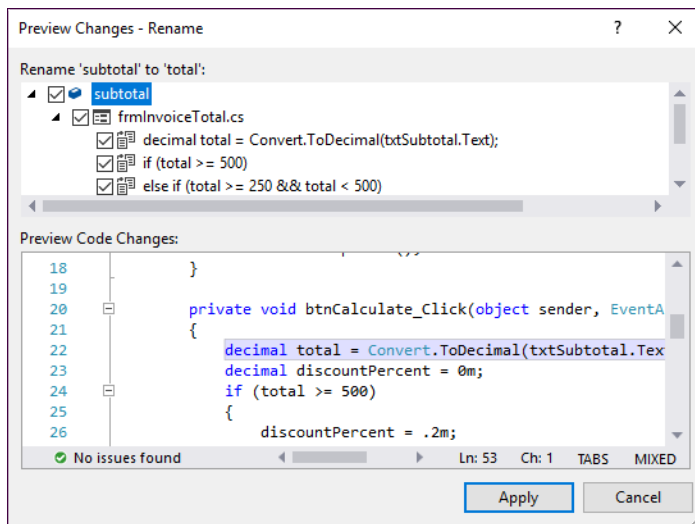
As you develop applications in C#, it's likely that you'll need some additional information about Visual Studio, the C# language, or the classes that are available from the .NET platform. The easiest way to get that information is to search the Internet. When you do, your searches will often lead to Microsoft's official online documentation, a great source of information about C# and .NET programming.

Because you use your web browser to access this help information, you can work with it just as you would any other web page. For example, to jump to a topic, you can click on the appropriate link. Or, to move back through previously displayed topics, you can use your browser's Back button. As a result, with a little practice, you shouldn't have much trouble getting help information.

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*. Refactoring is part of the live code analysis feature.
- To change a name that's used in your code, you can highlight the name, right-click on it, 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.
- 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.

Figure 3-12 How to refactor code

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.

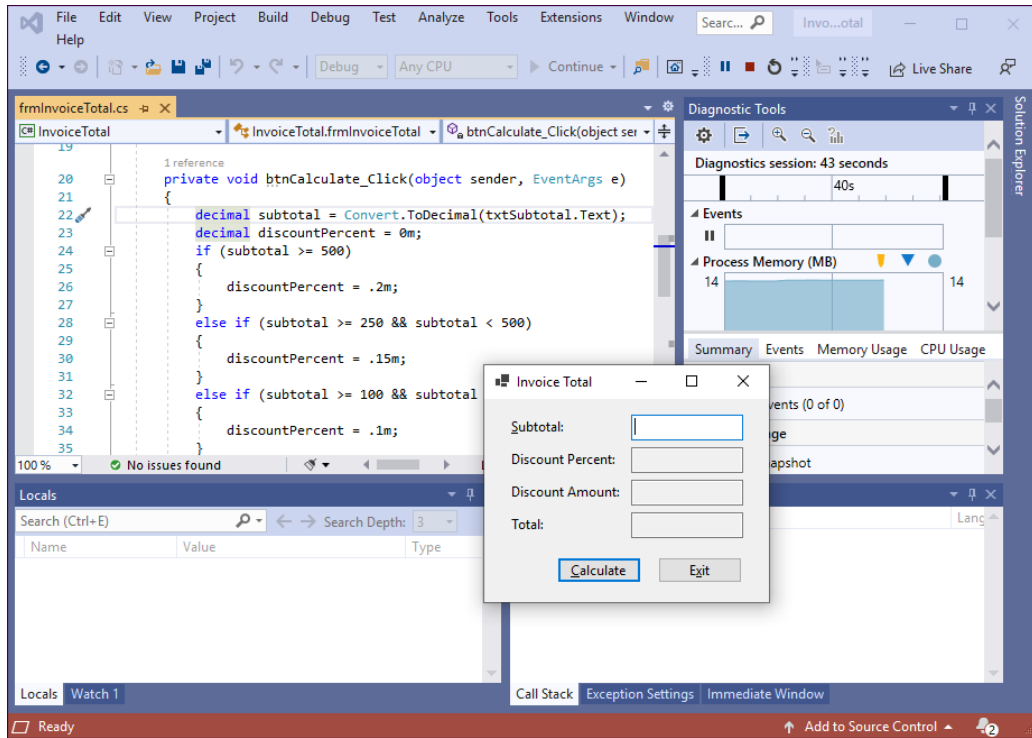
How to run a project

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 3-13. 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. When you do that, you should realize that the errors will still be listed in the Error List window and highlighted in the Code Editor even after you've corrected them. The errors aren't cleared until you build the project again.

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



Description

- To *run* a project, click the Start button in the Standard toolbar (the one with the green arrowhead and project name on it), 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.
- To locate the statement that contains the error, you can double-click on the error description in the Error List window. After you've corrected all the errors, run the project again to rebuild it and clear the errors.
- 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 3-13 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 3-14 provides an overview of the testing process for C# 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.

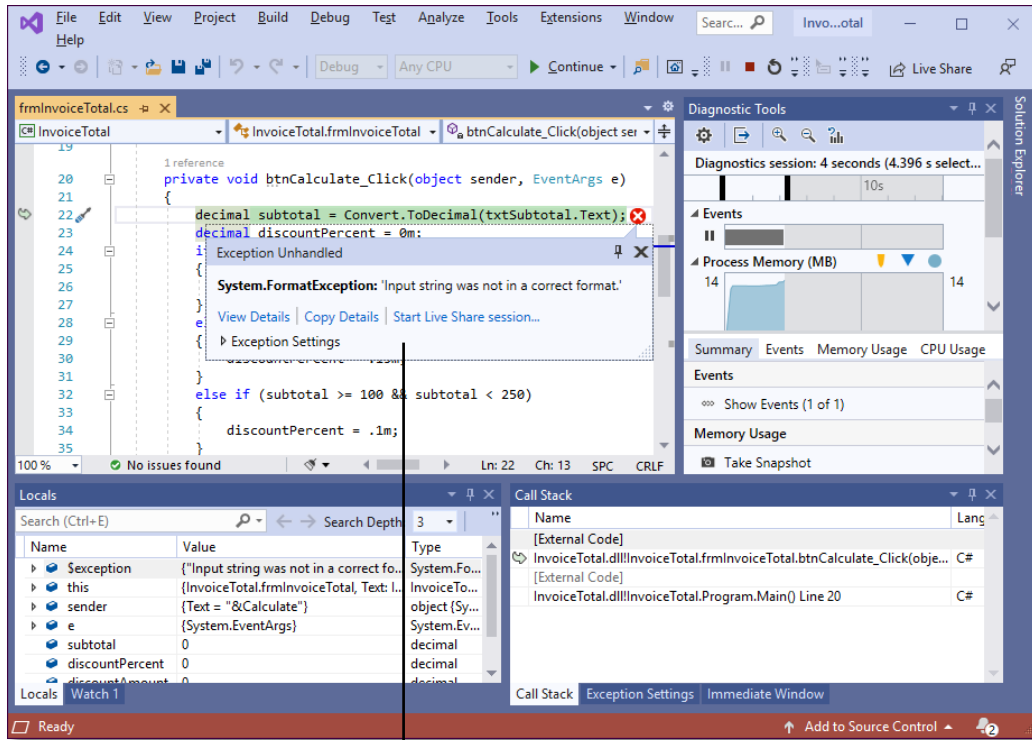
Finally, test the program to make sure that it properly handles invalid data entered by users. For example, type text information into text boxes that expect numeric data. Leave fields blank. Use negative numbers where they shouldn't be allowed. Remember that the goal of testing is to find all of the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

As you test your projects, you'll encounter *runtime errors*. These errors, also known as *exceptions*, occur when C# 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 Helper 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 Helper that's displayed when a runtime error occurs



Exception Helper 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 Helper window like the one above is displayed. At that point, you need to debug the application.

Figure 3-14 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 Helper window like the one shown in figure 3-14. 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.

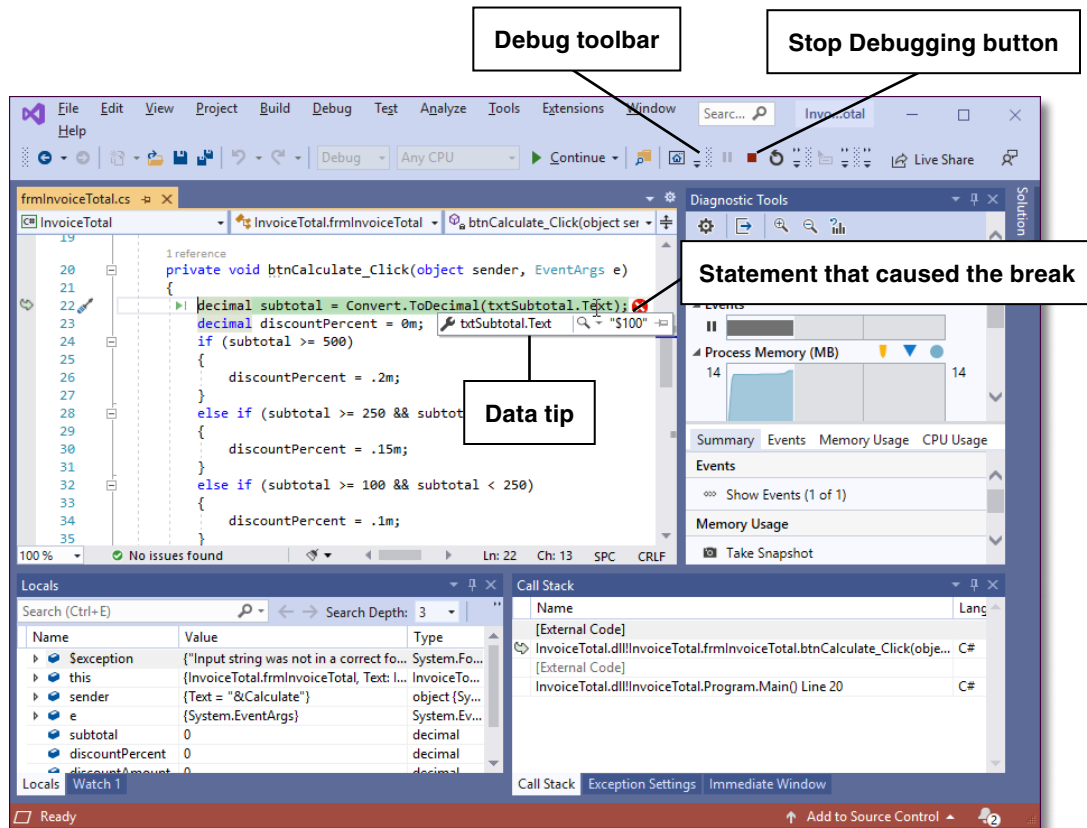
Often, you can figure out what caused the problem just by knowing what statement couldn't be executed or by reading the message displayed by the Exception Helper. But sometimes, it helps to find out what the current values in some of the variables or properties in the program are.

To do that, you can place the mouse pointer over a variable or property in the code to display a *data tip* as shown in figure 3-15. This tip displays the current value of the variable or property. You can do this with the Exception Helper 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 "\$100", which shows that the user didn't enter valid numeric data.

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 Debug 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 11, 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 Helper window.
- The Exception Helper window suggests what the error might be.
- If you close the Exception Helper 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 C# code.
- To exit break mode and end the application, click the Stop Debugging button in the Debug toolbar or press Shift+F5.
- You'll learn more about debugging in chapter 11.

Figure 3-15 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 C# 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. For starters, you need to learn the C# language. So in the next six chapters, you'll learn the essentials of the C# language. Then, in chapter 11, you'll learn some debugging techniques that can be used with more advanced code.

Terms

object-oriented programming	event-driven application	bookmark
object-oriented language	event handler	collapse
object	event wiring	expand
class	method declaration	code snippet
instance	method name	refactoring
instantiation	statement	inline renaming
property	block of code	build a project
method	syntax error	run a project
event	build error	test a project
member	live code analysis	runtime error
dot operator	comment	exception
dot	single-line comment	bug
static member	delimited comment	debug
	comment out a line	break mode
		data tip

Exercise 3-1 Code and test 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 correctly. You'll also experiment with debugging and review some help information.

Copy and open the Invoice Total application

1. Use Windows Explorer to copy the Invoice Total project that you created for chapter 2 from the C:\C#\Chapter 02 directory to the C:\C#\Chapter 03 directory.
2. Open the Invoice Total solution (InvoiceTotal.sln) that's now in the C:\C#\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 method declaration for the Click event of this object. Then, enter the code for this method 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 method 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 3-7. If any syntax errors are listed in this window, double-click on each error to move to the error in the Code Editor. Then, correct the error.

Test the application

6. Press F5 to build and run the project. If you corrected all the syntax errors in step 5, the build should succeed and the Invoice Total form should appear. If not, you'll need to correct the errors and press F5 again.
7. 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 methods 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 Helper as shown in figure 3-14.
9. Note the highlighted statement and read the message that's displayed in the Exception Helper. Then, move the mouse pointer over the variable and property in this statement to display their data tips. This shows that the code for this application needs to be enhanced so it checks for invalid data. You'll learn how to do that in chapter 7. For now, though, click the Stop Debugging button in the Debug toolbar to end the application.

Create a syntax error and see how it affects the IDE

10. When you return to the Code Editor, hide the Error List window by clicking on its Auto Hide button. Next, change the name of the Subtotal text box from txtSubtotal to txtSubTotal. This creates an error since the capitalization doesn't match the capitalization used by the Name property of the text box.
11. Try to run the application, and click No when Visual Studio tells you the build had errors and asks whether you want to continue with the last successful build. Then, double-click on the error in the Error List, correct the error, and run the application again to make sure the problem is fixed.

Use refactoring

12. Highlight the name of the subtotal variable, then right-click on the name and

select the Rename command from the shortcut menu that's displayed. When the Rename dialog box appears, enter the name `invoiceSubtotal` and notice that all occurrences of the variable are changed.

13. 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.

Generate and delete an event handler

14. Display the Form Designer for the Invoice Total form and double-click a blank area on the form. This should generate an event handler for the Load event of the form.
15. Delete the event handler for the Load event of the form. Then, run the application. When you do, you'll get a build error that indicates that the form does not contain a definition for this event handler.
16. Double-click on the error. This opens the Designer.cs file for the form and jumps to the statement that wires the event handler. Delete this statement to correct the error.
17. If you're curious, review the generated code that's stored in the Designer.cs file for this simple form. Then, click the minus sign to the left of the region named "Windows Form Designer generated code" to collapse this region.
18. Run the form to make sure it's working correctly. When you return to the Code Editor, close the Designer.cs file for the form.

Exit from Visual Studio

19. 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 C# programming skills

The easiest way is to let [Murach's C# \(7th Edition\)](#) 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 C# using Windows Forms applications
- Take advantage of all the time- and work-saving features of Visual Studio as you develop C# applications
- Build database applications using Entity Framework (EF) Core, ADO.NET, LINQ, and the DataGridView control
- 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 C# application
- Loan to your colleagues who will be asking you more and more questions about C# programming



Mike Murach, Publisher

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

When you buy directly from us, you must be satisfied. Try our books for 30 days or our eBooks for 14 days. They must outperform any competing book or course you've ever tried, or return your purchase for a prompt refund....no questions asked.

Thanks for your interest in Murach books!

A handwritten signature in black ink, appearing to read 'Mike'.