Extra exercises

The extra exercises are designed to give you experience with all of the critical Visual Studio and C# skills. As a result, some of these exercises will take considerably longer than an hour to complete. In general, the more exercises you do and the more time you spend doing them, the more competent you will become.

Guidelines for doing the extra exercises

- Many of the exercises have you start from a project that contains one or more forms and some of the code for the application. Then, you supply any additional forms, controls, or code that's required.

- Do the exercise steps in sequence. That way, you will work from the most important tasks to the least important.

- If you are doing an exercise in class with a time limit set by your instructor, do as much as you can in the time limit.

- Feel free to copy and paste code from the book applications or exercises that you've already done.

- Use your book as a guide to coding.

Extra 2-1   Design a simple form

In this exercise, you'll design a form that lets the user enter a number grade and then displays the letter grade when the user clicks the Calculate button.

1. Start a new project named CalculateLetterGrade in the Extra Starts\Chapter 02 directory.

2. Add the labels, text boxes, and buttons to the form as shown above. Then, set the properties of these controls as follows:

Default name	Property	Setting
label1	Text	&Number grade:
	TextAlign	MiddleLeft
	TabIndex	0
label2	Text	Letter grade:
	TextAlign	MiddleLeft
textBox1	Name	txtNumberGrade
	TabIndex	1
textBox2	Name	txtLetterGrade
	ReadOnly	True
	TabStop	False
button1	Name	btnCalculate
	Text	&Calculate
	TabIndex	2
button2	Name	btnExit
	Text	E&xit
	TabIndex	3

3. Set the properties of the form as follows:

Default name	Property	Setting
Form1	Text	Calculate Letter Grade
	AcceptButton	btnCalculate
	CancelButton	btnExit
	StartPosition	CenterScreen

4. Use the Form Designer to adjust the size and position of the controls and the size of the form so they look as shown above.

5. Rename the form to frmCalculateGrade. When you're asked if you want to modify any references to the form, click the Yes button.

6. Save the project and all of its files.

Extra 3-1   Code and test the Calculate Letter Grade form

In this exercise, you'll add code to a Calculate Letter Grade form. Then, you'll test the project to be sure it works correctly.

1. Open the CalculateLetterGrade project in the Extra Starts\Chapter 03 directory.

2. Display the form in the Form Designer, and double-click the Calculate button to generate a Click event handler for it. Then, add this statement to the event handler to get the number grade the user enters:

```
decimal numberGrade = Convert.ToDecimal(txtNumberGrade.Text);
```

3. Add this statement to the event handler to declare and initialize the variable that will hold the letter grade:

```
string letterGrade = "";
```

Then, add this if-else statement to set the letter grade:

```
if (numberGrade >= 88)
{
    letterGrade = "A";
}
else if (numberGrade >= 80 && numberGrade <= 87)
{
    letterGrade = "B";
}
else if (numberGrade >= 68 && numberGrade <= 79)
{
    letterGrade = "C";
}
else if (numberGrade >= 60 && numberGrade <= 67)
{
    letterGrade = "D";
}
else
{
    letterGrade = "F";
}
```

4. Add this statement to display the letter grade in the Letter Grade text box:

```
txtLetterGrade.Text = letterGrade;
```

5. Add this statement to move the focus back to the Number Grade text box:

```
txtNumberGrade.Focus();
```

6. Return to the Form Designer, and then double-click the Exit button to generate a Click event handler for it. Then, add this statement to the event handler to close the form:
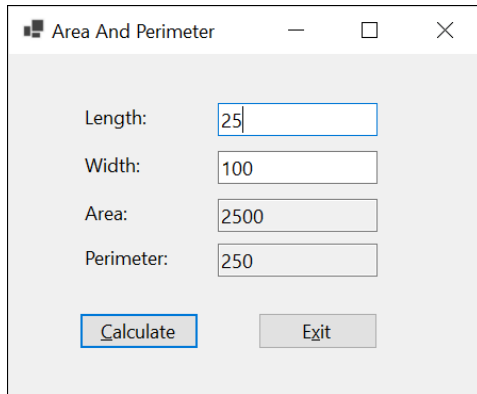
```
this.Close();
```

7. Run the application, enter a number between 0 and 100, and then click the Calculate button. A letter grade should be displayed and the focus should return to the Number Grade text box. Next, enter a different number and press the enter key to display the letter grade for that number. When you're done, press the Esc key to end the application.

Extra 4-1   Calculate area and perimeter

In this exercise, you'll create a form that accepts the length and width of a rectangle from the user and then calculates the area and perimeter of the rectangle.
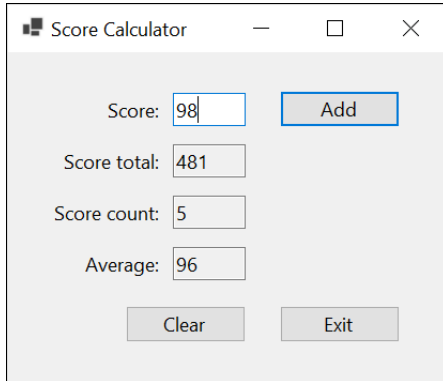

1. Start a new project named AreaAndPerimeter in the Extra Starts\Chapter 04 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Create an event handler for the Click event of the Calculate button. This event handler should get the values the user enters for the length and width, calculate and display the area (length x width) and perimeter (2 x length + 2 x width), and move the focus to the Length text box. It should provide for decimal entries, but you can assume that the user will enter valid decimal values.

4. Create an event handler for the Click event of the Exit button that closes the form.

5. Test the application to be sure it works correctly.

Extra 4-2   Accumulate test score data

In this exercise, you'll create a form that accepts one or more scores from the user. Each time a score is added, the score total, score count, and average score are calculated and displayed.

1. Start a new project named ScoreCalculator in the Extra Starts\Chapter 04 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Add button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Declare two class variables to store the score total and the score count.

4. Create an event handler for the Click event of the Add button. This event handler should get the score the user enters, calculate and display the score total, score count, and average score, and move the focus to the Score text box. It should provide for integer entries, but you can assume that the user will enter valid integer values.

5. Create an event handler for the Click event of the Clear button. This event handler should set the two class variables to zero, clear the text boxes on the form, and move the focus to the Score text box.

6. Create an event handler for the Click event of the Exit button that closes the form.

7. Test the application to be sure it works correctly.
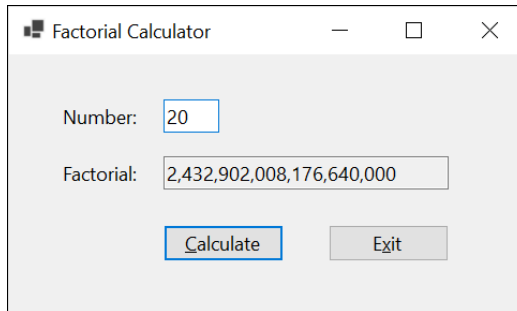
Extra 5-1   Calculate the factorial of a number

In this exercise, you'll create a form that accepts an integer from the user and then calculates the factorial of that integer.

The factorial of an integer is that integer multiplied by every positive integer less than itself. A factorial number is identified by an exclamation point following the number. Here's how you calculate the factorial of the numbers 1 through 5:

$1! = 1$	which equals 1
$2! = 2 * 1$	which equals 2
$3! = 3 * 2 * 1$	which equals 6
$4! = 4 * 3 * 2 * 1$	which equals 24
$5! = 5 * 4 * 3 * 2 * 1$	which equals 120
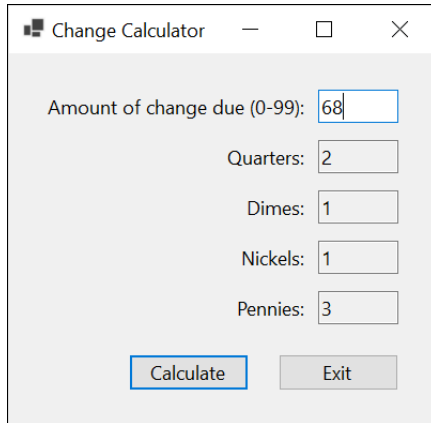
To be able to store the large integer values for the factorial, this application can't use the Int32 data type.

1. Start a new project named Factorial in the Extra Starts\Chapter 05 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Create an event handler for the Click event of the Calculate button. This event handler should get the number the user enters, calculate the factorial of that number, display the factorial with commas but no decimal places, and move the focus to the Number text box. It should return an accurate value for integers from 1 to 20. (The factorial of the number 20 is shown in the form above.)

4. Create an event handler for the Click event of the Exit button that closes the form.

5. Test the application to be sure it works correctly.

Extra 5-2   Calculate change

In this exercise, you'll develop a form that tells how many quarters, dimes, nickels, and pennies are needed to make change for any amount of change from 0 through 99 cents. One way to get the results is to use the division and modulus operators and to cast the result of each division to an integer.

1. Start a new project named ChangeCalculator in the Extra Starts\Chapter 05 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Create an event handler for the Click event of the Calculate button. Then, write the code for calculating and displaying the number of quarters, dimes, nickels, and pennies that are needed for the change amount the user enters. This code should provide for integer entries, but you can assume that the user will enter valid integer values.

4. Create an event handler for the Click event of the Exit button that closes the form.

5. Test the application to be sure it works correctly.

Extra 5-3   Calculate income tax

In this exercise, you'll use nested if statements and arithmetic expressions to calculate the federal income tax that is owed for a taxable income amount entered by the user.
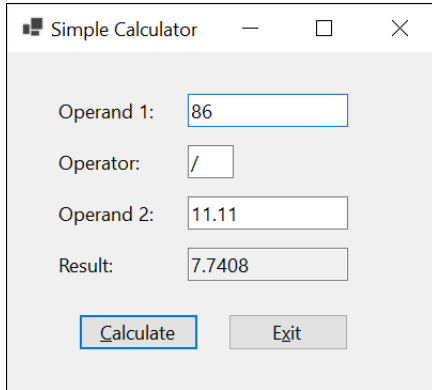
This is the 2020 table for the federal income tax on individuals that you should use for calculating the tax:

Taxable income		Income tax	
Over…	**But not over…**		**Of excess over…**
$0	$9,875	$0 plus 10%	$0
$9,876	$40,125	$987.50 plus 12%	$9,875
$40,126	$85,525	$4,617.50 plus 22%	$40,125
$85,526	$163,300	$14,605.50 plus 24%	$85,525
$163,301	$207,350	$33,271.50 plus 32%	$163,300
$207,351	$518,400	$47,367.50 plus 35%	$207,350
$518,401		$156,235 plus 37%	$518,400

1. Start a new project named TaxCalculator in the Extra Starts\Chapter 05 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Create an event handler for the Click event of the Exit button that closes the form.

4. Create an event handler for the Click event of the Calculate button. Then, write the code for calculating and displaying the tax owed for any amount within the first two brackets in the table above. This code should provide for decimal entries, but you can assume that the user will enter valid decimal values. To test this code, use income values of 8700 and 35350, which should display taxable amounts of 870 and 4044.50.

5. Add the code for the next tax bracket. Then, if you have the time, add the code for the remaining tax brackets.

Extra 6-1   Create a simple calculator

In this exercise, you'll create a form that accepts two operands and an operator from the user and then performs the requested operation.

1. Start a new project named SimpleCalculator in the Extra Starts\Chapter 06 directory.

2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate button should fire. When the user presses the Esc key, the Click event of the Exit button should fire.

3. Code a private method named Calculate() that performs the requested operation and returns a decimal value. This method should accept the following arguments:
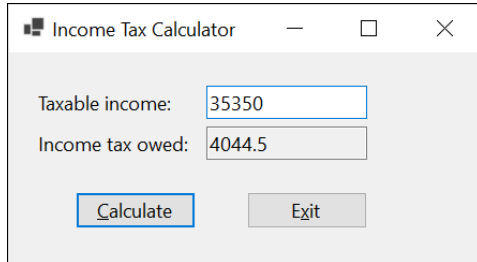
Argument	Description
`decimal operand1`	The value entered for the first operand.
`string operator1`	One of these four operators: +, -, *, or /.
`decimal operand2`	The value entered for the second operand.

4. Create an event handler for the Click event of the Calculate button. This event handler should get the two numbers and operand the user enters, call the Calculate() method to get the result of the calculation, display the result rounded to four decimal places, and move the focus to the Operand 1 text box.

5. Create an event handler for the Click event of the Exit button that closes the form.

6. Create an event handler that clears the Result text box if the user changes the text in any of the other text boxes.

7. Test the application to be sure it works correctly.

Extra 6-2   Add a method and an event handler to the income tax calculator

In this exercise, you'll add a method and another event handler to an income tax calculator application.

1. Open the TaxCalculator project in the Extra Starts\Chapter 06 directory and display the code for the form.

2. Code the declaration for a private method named CalculateTax() that receives the income amount and returns the tax amount.

3. Move the if-else statement in the btnCalculate_Click() event handler to the CalculateTax() method. Then, declare a variable for the tax at the beginning of this method, and return the tax at the end of the method.

4. Modify the statement in the btnCalculate_Click() event handler that declares the tax variable so it gets its value by calling the CalculateTax() method.

5. Create an event handler that clears the Income Tax Owed text box if the user changes the value in the Taxable Income text box.

6. Test the application to be sure it still works correctly. The income values of 8700 and 35350 should display taxable amounts of 870 and 4044.50.

Extra 7-1 Add exception handling to the simple calculator

In this exercise, you'll add exception handling to the Simple Calculator form.
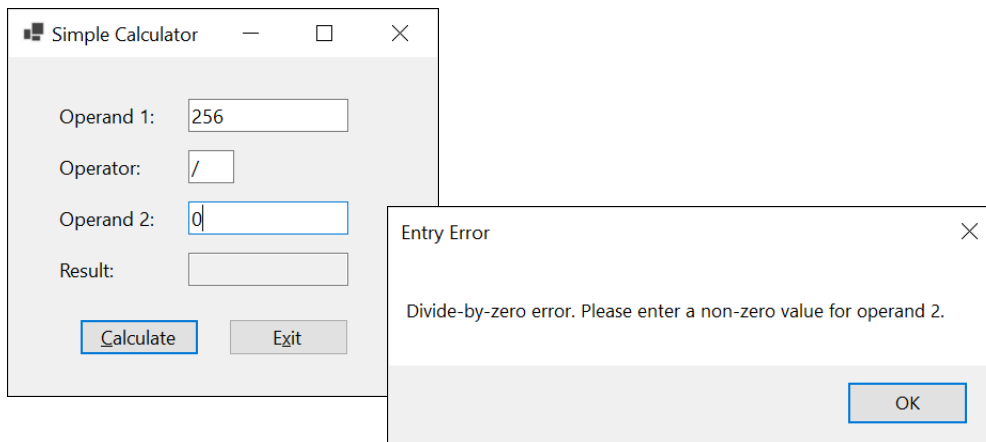
1. Open the SimpleCalculator project in the Extra Starts\Chapter 07\SimpleCalculatorExceptionHandling directory.

2. Add a try-catch statement in the btnCalculate_Click() event handler that will catch any exceptions that occur when the statements in that event handler are executed. If an exception occurs, display a dialog box with the error message, the type of error, and a stack trace. Test the application by entering a nonnumeric value for one of the operands.

3. Add three additional catch blocks to the try-catch statement that will catch a FormatException, an OverflowException, and a DivideByZeroException. These catch blocks should display a dialog box with an appropriate error message.

4. Test the application again by entering a nonnumeric value for one of the operands. Then, enter 0 for the second operand as shown above to see what happens.

Extra 7-2 Add data validation to the simple calculator

In this exercise, you'll add data validation to the Simple Calculator form.
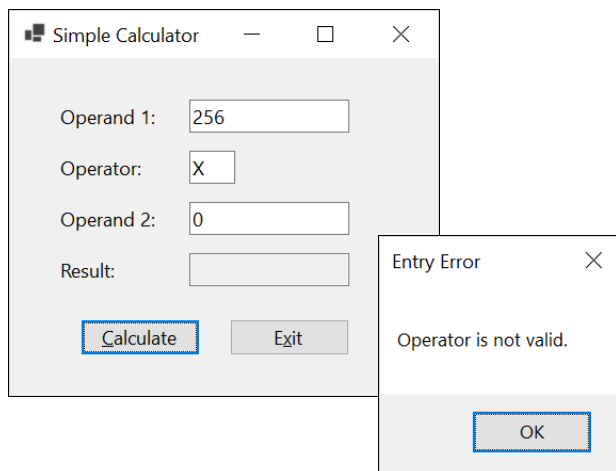
1. Open the SimpleCalculator project in the Extra Starts\Chapter 07\SimpleCalculatorValidation directory.

2. Code methods named IsPresent(), IsDecimal(), and IsWithinRange() that work like the methods described in chapter 7 of the book.

3. Code a method named IsOperator() that checks that the text box that's passed to it contains a value of +, -, *, or /.

4. Code a method named IsValidOperation() that checks for a divide by zero operation. This method will need to check the Operand 2 text box and the value of the Operator text box.

5. Code a method named IsValidData() that checks that the Operand 1 and Operand 2 text boxes contain a decimal value between 0 and 1,000,000, that the Operator text box contains a valid operator, and that the operation is valid.

6. Delete all of the catch blocks from the try-catch statement in the btnCalculate_Click() event handler except for the one that catches any exception. Then, add code to this event handler that performs the calculation and displays the result only if the values of the text boxes are valid.

7. Test the application to be sure that all the data is validated properly.

Extra 8-1   Display a test scores array

In this exercise, you'll enhance the Score Calculator form so it saves the scores the user enters in an array and then lets the user display the sorted scores in a dialog box.
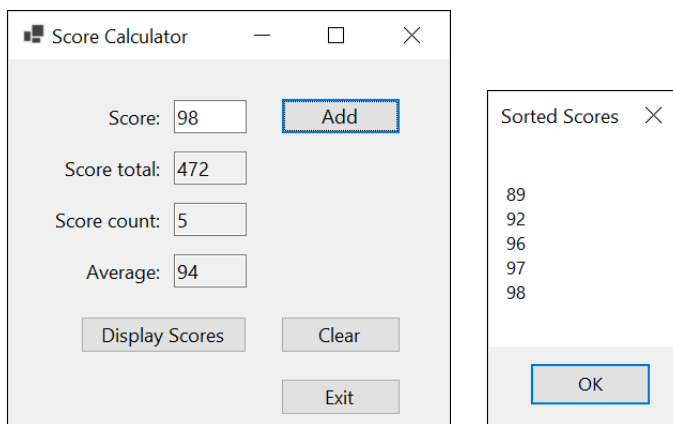
1. Open the ScoreCalculator project in the Extra Starts\Chapter 08\ScoreCalculatorArray directory.

2. Declare a class variable for an array that can hold up to 10 scores, and delete the class variable for the score total.

3. Modify the Click event handler for the Add button so it adds the score that's entered by the user to the next element in the array. To do that, you can use the score count variable to refer to the element. Then, convert the code that refers to the deleted total class variable to a local variable. Finally, use a foreach loop to add each score in the array to this local total variable.

4. Move the Clear button as shown above. Then, modify the Click event handler for this button so it removes any scores that have been added to the array. The easiest way to do that is to create a new array and assign it to the array variable. Also, remove the reference to the deleted total class variable.

5. Add a Display Scores button that sorts the scores in the array, displays the scores in a dialog box, and moves the focus to the Score text box. Be sure that only the elements that contain scores are displayed.

6. Test the application to be sure it works correctly. In particular, see what happens if you enter more than 10 scores.

Extra 8-2   Display a test scores list

In this exercise, you'll modify the Score Calculator form so it saves the scores the
user enters in a list and then lets the user display the sorted scores in a dialog box.

1. Open the ScoreCalculator project in the Extra Starts\Chapter
   08\ScoreCalculatorList directory.

2. Replace the declaration for the array variable with a declaration for a List<int>
   object, and delete the class variable for the score count.

3. Modify the Click event handler for the Add button so it adds the score that's
   entered by the user to the list. In addition, delete the statement that increments the
   score count variable you deleted.

4. Modify the Click event handler for the Clear button so it removes any scores that
   have been added to the list.

5. Modify the Click event handler for the Display Scores button so it sorts the
   scores in the list and then displays them in a dialog box.

6. Test the application to be sure it works correctly. In particular, see what happens
   if you enter more than 20 scores.

Extra 9-1   Calculate reservation totals

In this exercise, you'll add code that calculates the number of nights, total price, and average price for a reservation based on the arrival and departure dates the user enters.

Reservations	— □ ✕
Arrival date:	2/13/2021
Departure date:	2/16/2021
Number of nights:	3
Total price:	$390.00
Avg. price per night:	$130.00
Calculate	Exit

Open the project and implement the calculations

1. Open the Reservations project in the Extra Starts\Chapter 09 directory. Then, display the code for the form and notice that some of the methods are commented out so they don't return errors.

2. Add code to get the arrival and departure dates the user enters when the user clicks the Calculate button. Then, calculate the number of days between those dates, calculate the total price based on a price per night of $120, calculate the average price per night, and display the results.

3. Test the application to be sure it works correctly. At this point, the average price will be the same as the nightly price.

Enhance the way the form works

4. Add an event handler for the Load event of the form. This event handler should get the current date and three days after the current date and assign these dates to the Arrival Date and Departure Date text boxes as default values. Be sure to format the dates as shown above.

5. Modify the code so Friday and Saturday nights are charged at $150 and other nights are charged at $120. One way to do this is to use a while loop that checks the day for each date of the reservation.

6. Test the application to be sure that the default dates are displayed correctly and that the totals are calculated correctly.

Add code to validate the dates

7. Uncomment the IsDateTime() method and then add code to check that the arrival and departure dates are valid dates.

8. Uncomment the IsWithinRange() method and then add code to check that the arrival and departure dates are within a range that includes the minimum and maximum dates that are passed to it.

9. Uncomment the IsValidData() method and then add code that uses the IsPresent(), IsDateTime(), and IsWithinRange() methods to validate the arrival and departure dates. These dates should be in a range from the current date to five years after the current date.

10. Add code that uses the IsValidData() method to validate the arrival and departure dates. In addition, add code to check that the departure date is after the arrival date.

11. Test the application to be sure the dates are validated properly.

Extra 9-2   Work with strings

In this exercise, you'll add code that parses an email address and formats the city, state, and zip code portion of an address.

Open the project and add code to parse an email address

1. Open the StringHandling project in the Extra Starts\Chapter 09 directory.

2. Add code to parse the email address into two parts when the user clicks the Parse button: the user name before the @ sign and the domain name after the @ sign. Be to check that the email contains an @ sign before you parse it, and display an error message if it doesn't. Also, be sure to remove any leading or trailing spaces that the user enters. Display the results in a message box like the first one shown above.

3. Test the application with both valid and invalid email addresses to be sure it works correctly.

Add code to format an address

4. Add code to format the city, state, and zip code when the user clicks the Format button. To do that, create a string that contains the city, state, and zip code and then use the Insert() method to insert the appropriate characters. Be sure that the two-character state code is in uppercase. (You can assume that the user enters appropriate data in each text box.) Display the results in a message box like the second one shown above.

5. Test the application to be sure it formats the city, state, and zip code correctly.

Extra 10-1   Convert lengths

In this exercise, you'll add code to a form that converts the value the user enters based on the selected conversion type.

The application should handle the following conversions:

From	To	Conversion
Miles	Kilometers	1 mile = 1.6093 kilometers
Kilometers	Miles	1 kilometer = 0.6214 miles
Feet	Meters	1 foot = 0.3048 meters
Meters	Feet	1 meter = 3.2808 feet
Inches	Centimeters	1 inch = 2.54 centimeters
Centimeters	Inches	1 centimeter = 0.3937 inches

1. Open the LengthConversions project in the Extra Starts\Chapter 10 directory. Display the code for the form, and notice the rectangular array whose rows contain the value to be displayed in the combo box, the text for the labels that identify the two text boxes, and the multiplier for the conversion as shown above.

2. Set the DropDownStyle property of the combo box so the user must select an item from the list.

3. Add code to load the combo box with the first element in each row of the rectangular array, and display the first item in the combo box when the form is loaded.

4. Add code to change the labels for the text boxes, clear the calculated length, and move the focus to the entry text box when the user selects a different item from the combo box.

5. Test the application to be sure the conversions are displayed in the combo box, the first conversion is selected by default, and the labels change appropriately when a different conversion is selected.

6. Add code to calculate and display the converted length when the user clicks the Calculate button. To calculate the length, you can get the index for the selected conversion and then use that index to get the multiplier from the array. Test the application to be sure this works correctly.

7. Add code to check that the user enters a valid decimal value for the length. Then, test the application one more time to be sure the validation works correctly.

Extra 10-2   Process lunch orders

In this exercise, you'll complete a form that accepts a lunch order from the user and then calculates the order subtotal and total.

The application should provide for these main courses and add-ons:

Main course	Price	Add-on	Add-on price
Hamburger	6.95	Lettuce, tomato, and onions Ketchup, mustard, and mayo French fries	.75
Pizza	5.95	Pepperoni Sausage Olives	.50
Salad	4.95	Croutons Bacon bits Bread sticks	.25

1. Open the LunchOrder project in the Extra Starts\Chapter 10 directory.

2. Add three radio buttons to the Main Course group box, and set their properties so they appear as shown above. The Hamburger option should be selected by default.

3. Add a group box for the add-on items. Then, add three check boxes to this group box as shown above. None of the check boxes should be selected by default.

4. Code a method name ClearTotals() that clears the three text boxes and a method named ClearAddOns() that removes the check marks from the three check boxes.

5. Code an event handler that changes the text that's displayed for the Add-ons group box and the three check boxes when the user selects a different main course. This event handler should also remove the check marks from the add-ons and clear the order totals. Test the application to be sure this works correctly.

6. Code an event handler that calculates and displays the subtotal, tax, and order total when the user clicks the Place Order button. The subtotal is the cost of the main course plus the cost of the add-ons. The tax is 7.75% of the subtotal. And the order total is the subtotal plus the tax. Test the application to be sure this works correctly.

7. Code an event handler that clears the order totals when the user checks or unchecks an add-on. Then, test the application one more time.

Extra 10-3 Add a second form to an application

In this exercise, you'll add a second form to an Invoice Total application that lets the user change the sales tax percent.

Open the project and change the name of the existing form

1. Open the InvoiceTotal project in the Extra Starts\Chapter10 directory.

2. Change the name of the existing form to frmInvoiceTotal.

Create the Sales Tax form

3. Add another form named frmSalesTax to the project.

4. Add a label, text box, and two buttons to the new form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the OK button should fire. When the user presses the Esc key, the Click event of the Cancel button should fire.

5. Add code to get the sales tax, store it in the Tag property of the form, and set the DialogResult property of the form to OK when the user clicks the OK button.

Modify the code for the Invoice Total form

6. Change the SalesTax constant that's declared in the Invoice Total form so its value can be changed.

7. Add a Change Percent button to the Invoice Total form as shown above. Then, add code that displays the Sales Tax form and gets the result when the user clicks this button. If the user clicks the OK button on the Sales Tax form, this event handler should store the new sales tax percent in the sales tax variable and change the Tax label on the form so it displays the correct tax. Test the application to be sure this works correctly.

8. Add data validation to the Sales Tax form to check that the user enters a decimal value between 0 and 10 (noninclusive). To make that easier, you can copy the IsPresent(), IsDecimal(), and IsWithinRange() methods from the Invoice Total form. Test the application to be sure the validation works correctly.

Extra 12-1   Create and use a class

In this exercise, you'll add a class to an Inventory Maintenance application and then add code to the two forms that use this class.
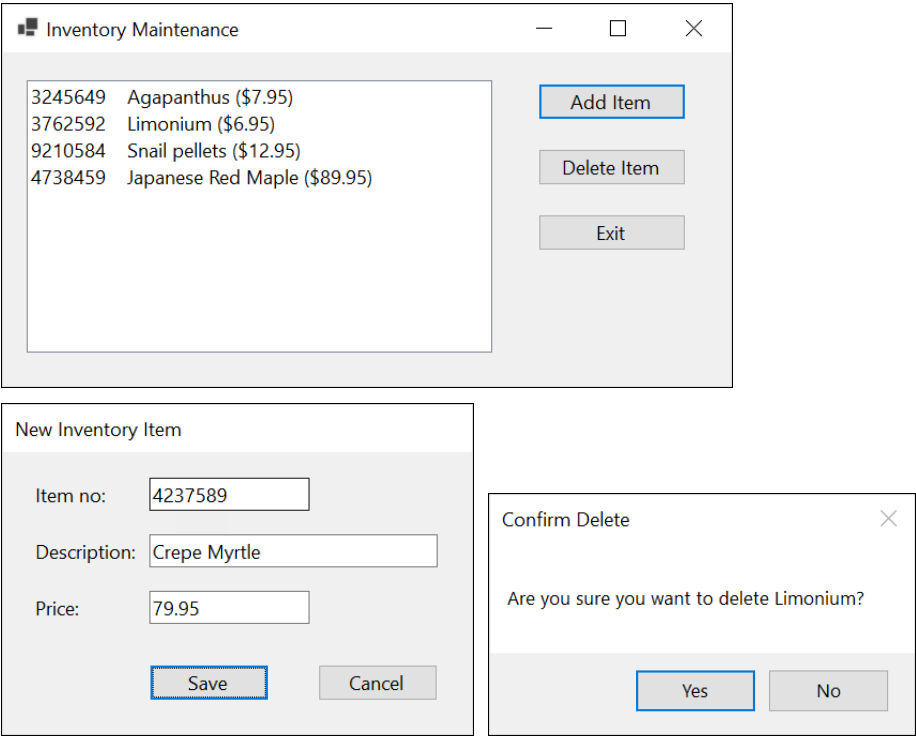
Open the project and add an InvItem class

1. Open the InventoryMaintenance project in the Extra Starts/Chapter 12 directory. Then, review the existing code for both of the forms so you get an idea of how this application should work.

2. Add a class named InvItem to this project, and add the properties, method, and constructors that are shown in the table below.

Property	Description
ItemNo	Gets or sets an int that contains the item's number.
Description	Gets or sets a string that contains the item's description.
Price	Gets or sets a decimal that contains the item's price.
Method	**Description**
GetDisplayText()	Returns a string that contains the item's number, description, and price formatted like this: 3245649 Agapanthus ($7.95). (The item number and description are separated by four spaces.)
Constructor	**Description**
()	Creates an InvItem object with default values.
(itemNo, description, price)	Creates an InvItem object with the specified values.

Add code to implement the New Item form

3. Display the code for the New Item form, and declare a class-level InvItem variable named invItem with an initial value of null.

4. Add a public method named GetNewItem() that displays the form as a dialog box and returns an InvItem object.

5. Add code to the btnSave_Click() event handler that creates a new InvItem object and closes the form if the data is valid.

Add code to implement the Inventory Maintenance form

6. Display the code for the Inventory Maintenance form, and declare a class-level List<InvItem> variable named invItems with an initial value of null.

7. Add a statement to the frmInvMaint_Load() event handler that uses the GetItems() method of the InvItemDB class to load the items list.

8. Add code to the FillItemListBox() method that adds the items in the list to the Items list box. Use the GetDisplayText() method of the InvItem class to format the item data.

9. Add code to the btnAdd_Click() event handler that creates a new instance of the New Item form and executes the GetNewItem() method of that form. If the InvItem object that's returned by this method is not null, this event handler should add the new item to the list, call the SaveItems() method of the InvItemDB class to save the list, and then refresh the Items list box. Test the application to be sure this event handler works.

10. Add code to the btnDelete_Click() event handler that removes the selected item from the list, calls the SaveItems() method of the InvItemDB class to save the list, and refreshes the Items list box. Be sure to confirm the delete operation. Then, test the application to be sure this event handler works.

Extra 13-1   Use indexers, delegates, events, and operators

In this exercise, you'll modify a class that stores a list of inventory items so it uses an indexer, a delegate, an event, and operators. Then, you'll modify the code for the Inventory Maintenance form so it uses these features.

Open the project and review the code

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 13 directory.

2. Review the code for the InvItemList class so you understand how it works. Then, review the code for the Inventory Maintenance form to see how it uses this class. Finally, run the application to see how it works.

Add an index to the InvItemList class

3. Delete the GetItemByIndex() method from the InvItemList class, and replace it with an indexer that receives an int value. This indexer should include both get and set accessors, and the get accessor should check that the value that's passed to it is a valid index. If the index isn't valid, the accessor should throw an ArgumentOutOfRangeException with a message that includes the index value.

4. Modify the Invoice Maintenance form to use this indexer instead of the GetItemByIndex() method. Then, test the application to be sure it still works.

Add overloaded operators to the InvItemList class

5. Add overloaded + and - operators to the InvItemList class that add and remove an inventory item from the inventory item list.

6. Modify the Inventory Maintenance form to use these operators instead of the Add() and Remove() methods. Then, test the application to be sure it still works.

Add a delegate and an event to the InvItemList class

7. Add a delegate named ChangeHandler to the InvItemList class. This delegate should specify a method with a void return type and an InvItemList parameter.

8. Add an event named Changed to the InvItemList class. This event should use the ChangeHandler delegate and should be raised any time the inventory item list changes.

9. Modify the Inventory Maintenance form to use the Changed event to save the inventory items and refresh the list box any time the list changes. To do that, you'll need to code an event handler that has the signature specified by the delegate, you'll need to wire the event to the event handler, and you'll need to remove any unnecessary code from the event handlers for the Save and Delete buttons. When you're done, test the application to be sure it still works.

Extra 14-1   Use inheritance

In this exercise, you'll add two classes to the Inventory Maintenance application that inherit the InvItem class. Then, you'll add code to the forms to provide for these new classes.
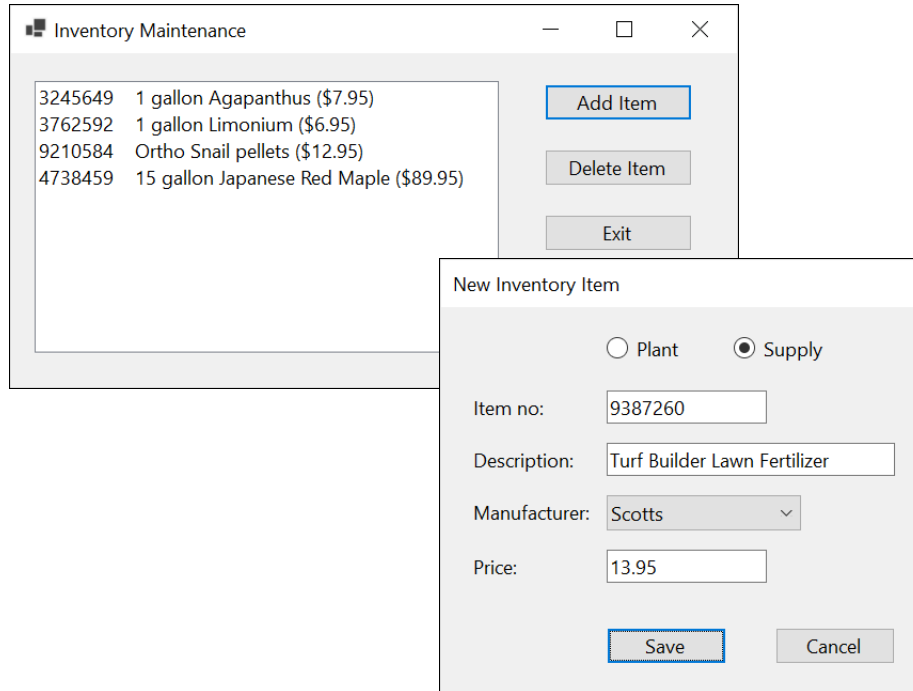
1. Open the InventoryMaintenance project in the Extra Starts\Chapter 14 directory. Then, review the code for the New Item form to see that the items in the combo box and the label for the combo box depend on which radio button is selected.

2. Display the InvItem class and modify the GetDisplayText() method so it's overridable.

3. Add a class named Plant that inherits the InvItem class. This new class should add a string property named Size. It should also provide a default constructor and a constructor that accepts four parameters (item number, description, price, and size) to initialize the class properties. This constructor should call the base class constructor to initialize the properties defined by that class. Finally, this class should override the GetDisplayText() method to add the size in front of the description, as in this example:

    3245649   1 gallon Agapanthus ($7.95)

4. Add another class named Supply that inherits the InvItem class and adds a string property named Manufacturer. Like the Plant class, the Supply class should provide a default constructor and a constructor that accepts four parameters, and it should override the GetDisplayText() method so the manufacturer is added in front of the description like this:

    9210584   Ortho Snail pellets ($12.95)

5. Modify the event handler for the Click event of the Save button on New Item form so it creates a new item of the appropriate type using the data entered by the user.

6. Test the application by adding at least one of each type of inventory item.

Extra 15-1   Create and use an interface

In this exercise, you'll create an IDisplayable interface and then use it in the InvItem class of the Inventory Maintenance application.

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 15\InventoryMaintIDisplayable directory.

2. Create a public interface named IDisplayable. Then, add the declaration for a method named GetDisplayText() with no parameters and a string return value.

3. Display the code for the InvItem class, find the GetDisplayText() method that this class contains, and comment out this method.

4. Modify the declaration for the class to indicate that it implements the IDisplayable interface. Then, generate a stub for the GetDisplayText() method that this interface defines.

5. Modify the declaration for the GetDisplayText() method so it can be overridden. Then, replace the generated code for this method with the code in the original GetDisplayText() method that you commented out in step 3.

6. Test the application to be sure it still works.

7. Display the code for the Inventory Maintenance form. Then, modify the event handler for the Click event of the Add button to include the following debugging code after the code that gets an InvItem object named invItem from the New Item form:

    ```
    Debug.WriteLine($"Item type: {invItem.GetType()}");
    Debug.WriteLine($"Item is InvItem: {invItem is InvItem}");
    Debug.WriteLine($"Item is IDisplayable: {invItem is IDisplayable}");
    ```

8. Run the application and add a new Plant item and a new Supply item. After adding each item, check the Output window and review the information that's displayed.

Extra 15-2   Implement the IEnumerable interface

In this exercise, you'll implement the IEnumerable interface in the list class of the Inventory Maintenance application.

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 15\InventoryMaintIEnumerable directory.

2. Display the code for the InvItemList class, and modify its declaration to indicate that it implements the IEnumerable interface. Then, generate a code stub for the only member of this interface, the GetEnumerator() method.

3. Replace the generated throw statement for the GetEnumerator() method with a foreach statement that returns each item that's stored in the list for the class. Be sure to precede the return keyword with the yield keyword.

4. Display the code for the Inventory Maintenance form. Then, modify the code for the FillItemListBox so it uses a foreach statement instead of a for statement.

5. Test the application to be sure it still works.

Extra 16-1   Add XML documentation to a class

In this exercise, you'll add documentation to the InvItem class of the Inventory Maintenance application.

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 16\InventoryMaintDocumentation directory.

2. Display the code for the InvItem class. Then, add some basic documentation for the class and its properties, methods, and constructors.

3. Display the code for the New Item form, and locate the btnSave_Click() event handler. Then, reenter the statement that creates the InvItem object up to the opening parenthesis. When you do, a screen tip should be displayed that lets you select from two constructors. Select the constructor that lets you enter values for the new item. The screen tip should include the documentation for the constructor, along with the documentation for the first parameter.

4. Enter a value for the first parameter, followed by a comma. Now, the screen tip should display the documentation for the second parameter. Finish entering the statement, then delete the statement you just added.

5. Display the code for the Inventory Maintenance form, and locate the FillItemListBox() method. Then, reenter the statement that adds an item to the list box up to the dot operator before the GetDisplayText() method. Highlight the GetDisplayText() method in the completion list that's displayed to see that the documentation you entered for this method is included in the screen tip. When you're done, delete the code you just entered.

6. Display the code for the InvItemDB class, and locate the SaveItems() method. Then, reenter the statements in the for loop that call the WriteElementString() method of the xmlOut object, and notice the documentation that's displayed when you select each property of the item object. Delete the statements you just entered.

Extra 16-2   Create and use a class library

In this exercise, you'll create a class library that includes InvItem and InvItemDB classes. Then, you'll use that class library with the Inventory Maintenance application.

Create the Class Library project

1. Create a new Class Library project named InventoryLibrary in the Extra Starts\Chapter 16\InventoryMaintLibrary directory.

2. Delete the empty Class1.cs class, and add the InvItem and InvItemDB classes from the Inventory Maintenance project in the same directory.

3. Change the namespace in both of the classes you just added so it's a namespace named Inventory nested within a namespace with your last name.

4. Change the solution configuration for the class library to Release, and then build the class library. When you're done, close the solution.

Modify the Inventory Maintenance application to use the library

5. Open the InventoryMaintenance project in the Extra Starts\Chapter 16\InventoryMaintLibrary directory.

6. Delete the InvItem.cs and InvItemDB.cs files, and then add a reference to the InventoryLibrary assembly you created in step 4.

7. Display the code for the Inventory Maintenance form, and add a using directive for the namespace you created in step 3. Do the same for the New Item form.

8. Test the application to be sure it still works.

Extra 17-1   Work with a text file

In this exercise, you'll add code to an Inventory Maintenance application that reads data from and writes data to a text file.

1.   Open the InventoryMaintenance project in the Extra Starts\Chapter 17\InventoryMaintText directory, and display the code for the InvItemDB class.

2.   Add code to the GetItems() method that creates a StreamReader object with a FileStream object for the InventoryItems.txt file that's included in the project. (The Path constant contains the path to this file.) The file should be opened if it exists or created if it doesn't exist, and it should be opened for reading only.

3.   Add code that reads each record of the text file, stores the fields, which are separated by pipe characters, in an InvItem object, and adds the object to the List<InvItem> object. Then, close the StreamReader object.

4.   Add code to the SaveItems() method that creates a StreamWriter object with a FileStream object for the InventoryItems.txt file. The file should be created if it doesn't exist or overwritten if it does exist, and it should be opened for writing only.

5.   Add code that writes each InvItem object in the List<InvItem> object to the text file, separating the fields with pipe characters. Then, close the StreamWriter object.

6.   Test the application to be sure it works correctly.

Extra 17-2   Work with a binary file

In this exercise, you'll add code to an Inventory Maintenance application that reads data from and writes data to a binary file.

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 17\InventoryMaintBinary directory, and display the code for the InvItemDB class.

2. Add code to the GetItems() method that creates a BinaryReader object with a FileStream object for the InventoryItems.dat file that's included in the project. (The Path constant contains the path to this file.) The file should be opened if it exists or created if it doesn't exist, and it should be opened for reading only.

3. Add code that reads each record of the binary file, stores the fields in an InvItem object, and adds the object to the List<InvItem> object. Then, close the BinaryReader object.

4. Add code to the SaveItems() method that creates a BinaryWriter object with a FileStream object for the InventoryItems.dat file. The file should be created if it doesn't exist or overwritten if it does exist, and it should be opened for writing only.

5. Add code that writes each InvItem object in the List<InvItem> object to the binary file. Then, close the BinaryWriter object.

6. Test the application to be sure it works correctly.

Extra 18-1   Use LINQ

In this exercise, you'll use LINQ to update the Inventory Maintenance application so it displays the inventory items in alphabetical order by description, and so it can filter the inventory items by price.

Inventory Maintenance

Show: All

```
3245649   Agapanthus ($7.95)          Add Item
4738459   Japanese Red Maple ($89.95)
3762592   Limonium ($6.95)            Delete Item
9210584   Snail pellets ($12.95)
```

Inventory Maintenance

Show: Under $10

```
3245649   Agapanthus ($7.95)          Add Item
3762592   Limonium ($6.95)
                                      Delete Item

                                        Exit
```
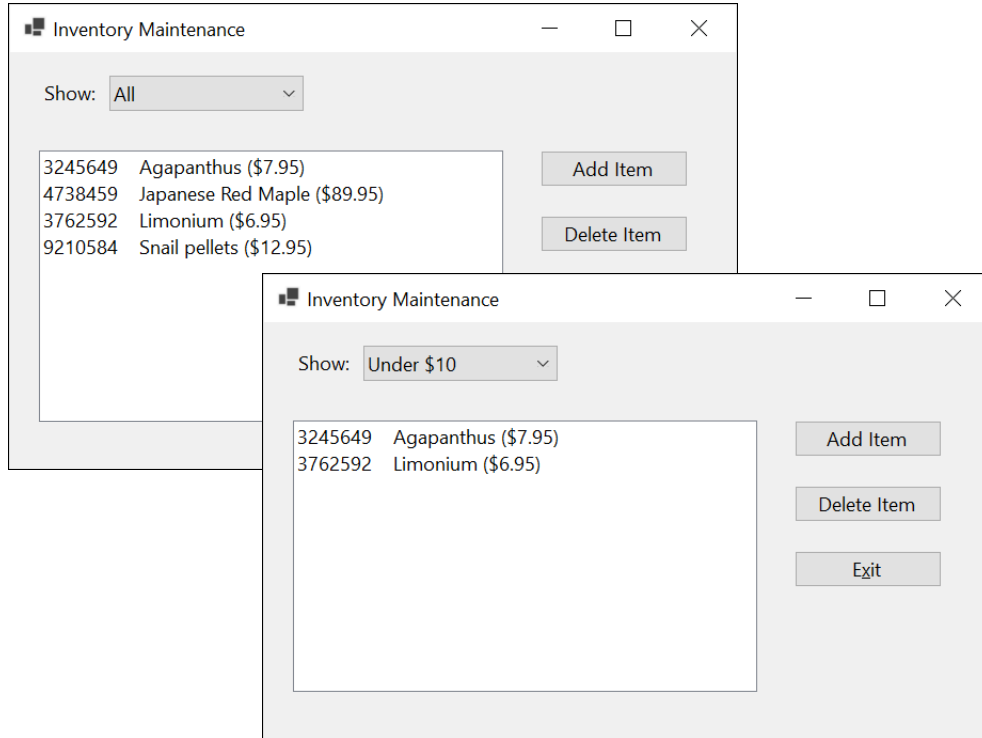
Review the starting code

1. Open the InventoryMaintenance project in the Extra Starts\Chapter 18 directory.

2. Display the code for the InvItemList class and note that it contains a readonly property named DisplayText.

3. Display the code for the Inventory Maintenance form and review the code that loads the items in the combo box and the code that executes when the selected item in the combo box is changed.

4. Run the application and view the items in the combo box. Notice that nothing happens when you change the selected item. Exit the application.

Add code that sorts and filters the inventory items

5. Find the FillItemListBox() method in the Inventory Maintenance form. It starts by storing the selected value of the combo box in a variable named filter and declaring a local collection of InvItem objects named filteredItems.

    Note: This local collection is of the IEnumerable<InvItem> type. That's because LINQ queries return IEnumerable<T> collections (see figure 18-2).

6. Code an if/else statement that uses LINQ to query the class-level collection named invItems based on the value of the filter variable. The LINQ queries

should also order the inventory items by Description. Assign the result of each LINQ query to the filteredItems collection.

Note: You can use method-based queries or query expressions here.

7. Update the foreach statement so it loops through the local filteredItems collection rather than the class-level invItems collection.

8. Test the application to be sure it displays the items in alphabetical order and filters the items by price when the selected item in the combo box changes.

9. Add the following new item to the inventory:

      ItemNo: 4372639   Description: Creeping Phlox   Price: 24.99

    Notice that the new item is sorted so it appears alphabetically in the display. Use the combo box to show items whose price is between $10 and $50, and see that the new item is displayed. When you're done, exit the application.

10. Open the InventoryItems.txt file and see that the items aren't in alphabetical order, and that the item you just added is the last item in the file.

Add code that uses LINQ to find the inventory item to delete

11. Run the application, select the item you added in step 9, and click Delete. Note that the confirmation message identifies the wrong item for deletion. Click Cancel and exit the application.

12. Display the btnDelete_Click() event handler in the Inventory Maintenance form and comment out the line of code that uses the selected index value of the ListView control to get the selected InvItem object from the invItems collection.

13. Add code that uses that selected index value to retrieve the display text of the selected item from the Items collection of the ListView control.

14. Use a LINQ query to get the selected InvItem object from the InvItems collection. The LINQ query should select the item whose DisplayText property is equal to the display text value you just retrieved.

    Note: To do this, you need to use the First() or FirstOrDefault() method. Because of that, you should code this as a method-based query.

15. Run the application and try to delete the item you added in step 9. This time, the confirm message should present the correct inventory item. Click OK to delete the item and exit the application.

Extra 20-1   Use Entity Framework Core

In this exercise, you'll use Entity Framework Core to create an application that lets you update the data in an OrderOptions table. This table contains a single row that stores the sales tax and shipping charges used by the application.

Create and modify the DB context and entity classes

1. Open the OrderOptionsMaintenance project in the Extra Starts\Chapter 20 directory. This project contains the Order Options Maintenance form, a Validator class, and code for validating the data the user enters.

2. Use the NuGet Package Manager to add the Microsoft.EntityFrameworkCore SqlServer and Tools packages, as shown in figure 20-2.

3. Use the Package Manager Console to generate DB context and entity classes for the MMABooks database, as shown in figure 20-3.

4. Update the generated DB context code so the connection string is retrieved from an application configuration file, as shown in figure 20-7.

Write the code to retrieve and update the options

5. Declare two class-level variables: one that stores an instance of the DB context and one for an OrderOption object that will store the results of the query that retrieves the data from the OrderOptions table.

6. In the Load event handler for the form, code a LINQ query to get the data from the OrderOptions table. Because this table contains a single row, you can use the Single() or First() methods. Assign the result to the OrderOption object that you defined in the last step. Then, assign the properties of the OrderOption object to the text boxes on the form.

7. In the btnSave_Click() event handler, assign the current values in the text boxes to the properties of the OrderOption object. Then, save the changes to the database and display a message indicating that the order options have been updated.

8. Test the application to be sure it works. Then, close the application.

Write code to handle database errors

9. In the btnSave_Click() event handler, move the existing code within a try clause and add a catch clause that handles the DbUpdateException as shown in figure 20-12.

10. Test the application to make sure it still works as it should.

Extra 21-1   Use ADO.NET

In this exercise, you'll use ADO.NET to create an application that lets you update the data in an OrderOptions table. This table contains a single row that stores the sales tax and shipping charges used by the application.

Write the code to retrieve and update the options

1. Open the OrderOptionsMaintenance project in the Extra Starts\Chapter 21 directory. This project contains the Order Options Maintenance form, a Validator class, code for validating the data the user enters, and a Models folder with an OrderOption class and two DB classes.

2. Use the NuGet Package Manager to add the Microsoft.Data.SqlClient package. You can use figure 20-2 as a guide.

3. In the MMABooksDB class, update the static ConnectionString property to get the connection string from the App.config file, as shown in figure 21-12, part 1.

4. In the OrderOptionsDB class, update the static GetOrderOptions() and UpdateOrderOptions() methods to retrieve and update the data from the OrderOptions table. You can use the code in figure 21-12, part 1 and 2, as a guide. Remember, though, that the OrderOptions table only contains one record. Because of that, you won't need to use a WHERE clause in your SELECT or UPDATE statement.

5. In the code for the maintenance form, declare a class-level variable for an OrderOption object that will store the data from the OrderOptions table.

6. In the Load event handler for the form, use the OrderOptionsDB class to get the data from the OrderOptions table and assign the result to the OrderOption object that you defined in the last step. Then, assign the properties of the OrderOption object to the text boxes on the form.

7. In the btnSave_Click() event handler, assign the current values in the text boxes to the properties of the OrderOption object. Then, use the OrderOptionsDB class to save the changes to the database. Finally, display a message indicating that the order options have been updated.

8. Test the application to be sure it works. Then, close the application.

Write code to handle database errors and concurrency conflicts

9. In the btnSave_Click() event handler, move the existing code within a try clause. Add a catch clause that handles the SqlException exception. Within that catch block, write code that displays the exception message to the user.

10. In the btnSave_Click() event handler, make sure the code that calls the UpdateOrderOptions() method only updates the form and displays a success message if that method returned a Boolean value of true. Then, update the code to display a concurrency message if that Boolean value is false. You can use figure 21-14, part 3, as a guide.

11. Test the application to make sure it still works as it should.

Extra 22-1   Create an application that uses a DataGridView control

In this exercise, you'll create an application that uses a DataGridView control to maintain the states in the States table of the MMABooks database.
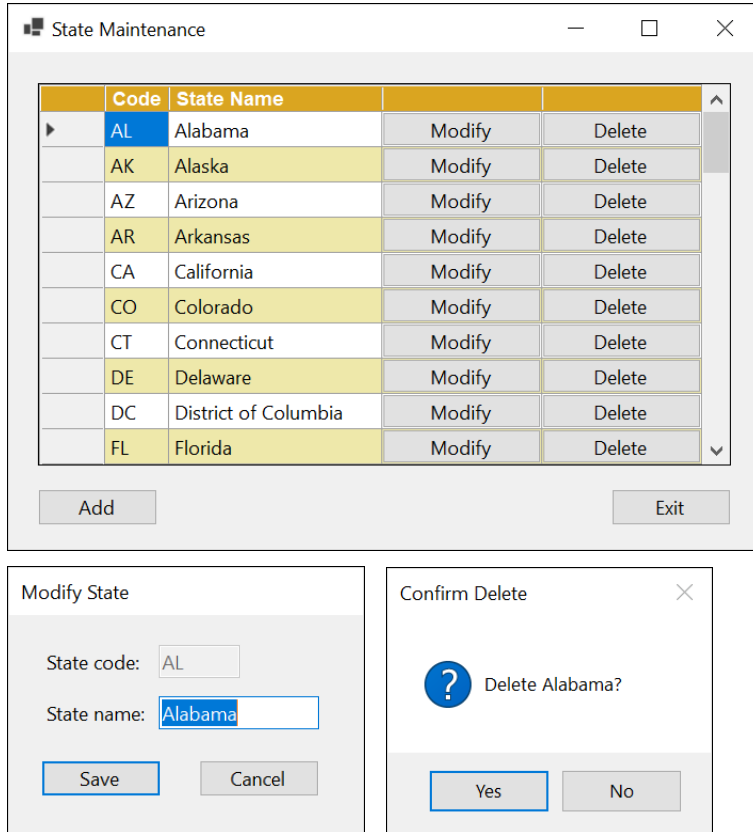
Open the project and create the grid

1. Open the StateMaintenance project in the Extra Starts\Chapter 22\StateMaintenance directory. This project start provides the EF Core DB context and entity classes, the Add/Modify form, the Validator class, and some of the code for the State Maintenance form.

2. Add a DataGridView control to the State Maintenance form. When the smart tag menu is displayed, set the options so the user can't add, edit, or delete data directly in the control.

3. In the DisplayStates() method, use a LINQ query to get the state code and state name for each state. Sort the states by name, and bind the DataGridView control to this list of states.

4. Format the columns so they're wide enough to display the data they contain. In addition, format the column headers and alternating rows any way you like.

5. Adjust the width of the grid so you can see all the columns as well as the vertical scrollbar. In addition, adjust the height of the grid so you can see 10 states at a time. You may have to run the application and adjust the width and height multiple times before you get this right.

Add two buttons to the grid

6. Declare a constant that contains the index value for the Modify button column that you'll add to the grid. Then, declare a constant that contains the index value for the Delete button column. Remember, indexes are zero based.

7. Add the button columns to the grid. To do that, use the Insert() method of the Columns collection, and pass in the correct constant for the index argument.

8. Re-adjust the width of the grid so you can see all the columns as well as the vertical scrollbar. Adjust the position of the buttons and the size of the form so it looks as shown above.

9. Generate an event handler for the CellClick event of the DataGridView control. This event handler should start by using the constant values to check if a cell that contains a Modify or Delete button was clicked. If it was, it should get the state code from the row that was clicked and then get the state with that code and store it in the selectedState class variable. Then, if a Modify button was clicked, it should call the ModifyState() method, and if a Delete button was clicked, it should call the DeleteState() method.

10. Test these changes to be sure they work. To test the delete operation, first add a new state and then delete that state.

Extra 22-2   Add paging to an application

In this exercise, you'll create an application that uses a data source and a DataGridView control to maintain the states in the States table of the MMABooks database.
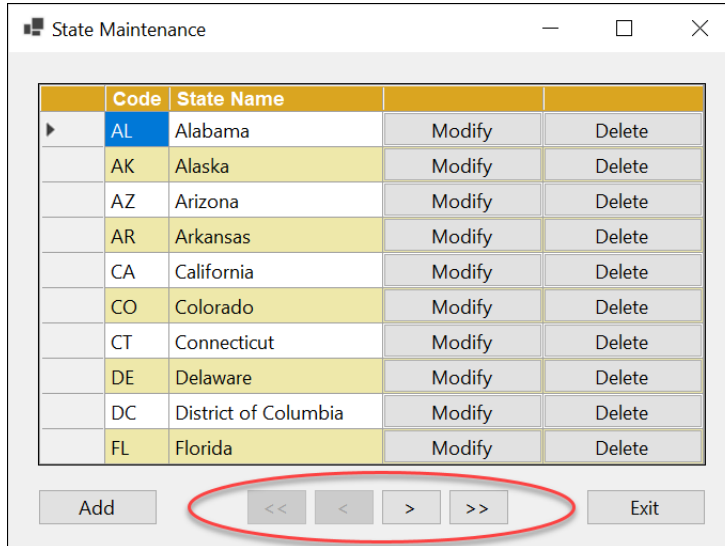
1. Open the StateMaintenance project in the Extra Starts\Chapter 22\StateMaintenancePaging directory.

2. Add four buttons between the Add and Exit buttons that will let the user move to the first, previous, next, and last page of customers.

3. Declare a constant with a value that indicates the maximum number of rows per page. Then, declare variables to store the total number of rows, the total number of pages, and the current page number.

4. Add code to the event handler for the Load event of the form that calculates and sets the total number of rows and the total number of pages. Then, set the current page number to the first page so this event handler displays the customer rows for the first page.

5. Modify the DisplayStates() method so it displays the state rows for the current page. To do that, this method can calculate the values for the skip and take variables needed by the LINQ query. If you need help, you can refer to the code in figure 22-11.

6. Make sure to size the combo box so it's tall enough and wide enough to display all rows for the page without needing to display a scroll bar.

7. Add a method named EnableDisableButtons() that enables or disables the appropriate buttons depending on the current page. Then, update the code in the DisplayStates() method so it calls the EnableDisableButtons() method.

8. Add an event handler for the Click event of the First button. Then, add code to this event handler that sets the current page to the first page and calls the DisplayStates() method.

9. Add event handlers for the remaining paging buttons, and add the code necessary to set the current page number and display that page of states. If you need help, you can refer to the code in figure 22-11.

10. Run the application to be sure the paging buttons work and are enabled and disabled as appropriate.

Extra 22-3   Create a Master/Detail form

In this exercise, you'll create a form that displays the states and the customers for a selected state.

1. Open the CustomersByState project in the Extra Starts\Chapter 22 directory.

2. Run the application and see that the states are displayed in a DataGridView control, with paging buttons to move forward and backward. Close the application by clicking the X in the upper right corner of the application.

3. Display the Customer By State form and add a second DataGridView control to display the customers in a selected state. When the smart tag menu is displayed, set the options so the user can't add, edit, or delete data.

4. In the code for the form, add a DisplayCustomers() method that accepts an index value of the int type. This method should use the index it receives to get the state row that was selected and then the state code from that row. Then, it should use a LINQ query to get the name, address, city, and zip code for each customer in the selected state and bind the Customers grid to that list of customers.

5. Add code to the DisplayStates() method that selects the first state in the States grid. Then, add code that calls the DisplayCustomers() method to display the customers for the first state in the Customers grid.

6. Generate an event handler for the RowHeaderMouseClick event of the States grid. Then, add code that displays the customers for the state that was clicked.