

# 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 .....	2
Extra 2-1 Design a simple form.....	3
Extra 3-1 Code and test the Calculate Letter Grade form .....	4
Extra 4-1 Calculate area and perimeter .....	6
Extra 4-2 Accumulate test score data .....	7
Extra 5-1 Calculate the factorial of a number .....	8
Extra 5-2 Calculate change.....	9
Extra 5-3 Calculate income tax .....	10
Extra 6-1 Create a simple calculator .....	11
Extra 6-2 Add a method and an event handler to the income tax calculator .	12
Extra 7-1 Add exception handling to the simple calculator .....	13
Extra 7-2 Add data validation to the simple calculator .....	14
Extra 8-1 Display a test scores array .....	15
Extra 8-2 Display a test scores list.....	16
Extra 9-1 Calculate reservation totals.....	17
Extra 9-2 Work with strings .....	19
Extra 10-1 Convert lengths .....	20
Extra 10-2 Process lunch orders.....	21
Extra 10-3 Add a second form to an Invoice Total application .....	22
Extra 12-1 Create and use an Inventory Item class.....	23
Extra 13-1 Modify a list class to use an indexer, a delegate, an event, and operators .....	25
Extra 14-1 Use inheritance with the Inventory Maintenance application .....	26
Extra 15-1 Create and use an interface .....	28
Extra 15-2 Implement the IEnumerable interface .....	29
Extra 16-1 Add XML documentation to a class.....	30
Extra 16-2 Create and use a class library.....	31
Extra 18-1 Create a State Maintenance application .....	32
Extra 18-2 Create a Product Maintenance application .....	33
Extra 19-1 Add parameterized queries to the Product Maintenance form.....	34
Extra 19-2 Create a Master/Detail form .....	35
Extra 20-1 Write the code for a Product Maintenance application.....	37
Extra 21-1 Work with a text file .....	40
Extra 21-2 Work with a binary file .....	41
Extra 22-1 Work with an XML file.....	42
Extra 23-1 Use LINQ to create an Invoice Line Items application .....	43
Extra 24-1 Use the Entity Framework to create an Order Options Maintenance application.....	45
Extra 25-1 Create an SDI application .....	46
Extra 25-2 Create an MDI application.....	47

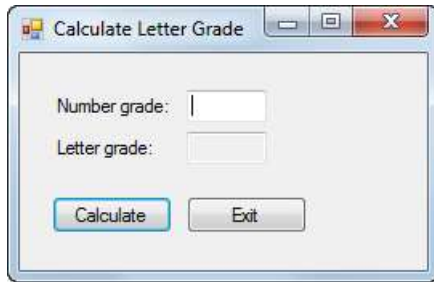
## Guidelines for doing the extra exercises

---

- 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 Exercises\Chapter 02\CalculateLetterGrade directory. Be sure to store the solution in its own 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. Now, 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 the Calculate Letter Grade form that you designed in extra exercise 2-1. Then, you'll build and test the project to be sure it works correctly.

1. Open the CalculateLetterGrade project in the Extra Exercises\Chapter 03\CalculateLetterGrade 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. Finally, 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();
```

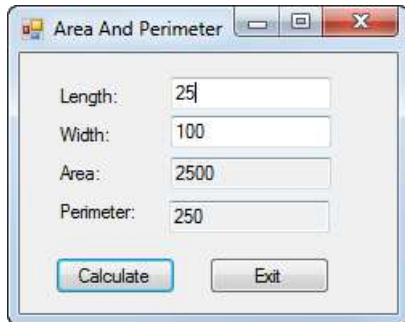
7. Run the application, enter a number between 0 and 100, and then click the Calculate button. A letter grade should be displayed and the focus should return to the Number Grade text box. Next, enter a different number and press the enter

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.

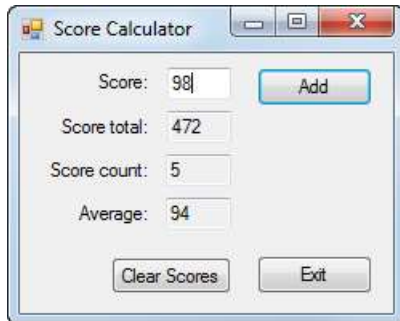


1. Start a new project named AreaAndPerimeter in the Extra Exercises\Chapter 04\AreaAndPerimeter directory.
2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event 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 ( $2 \times \text{length} + 2 \times \text{width}$ ), 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.

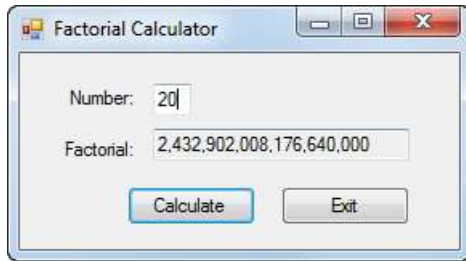


1. Start a new project named ScoreCalculator in the Extra Exercises\Chapter 04\ScoreCalculator directory.
2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event 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 Scores 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.

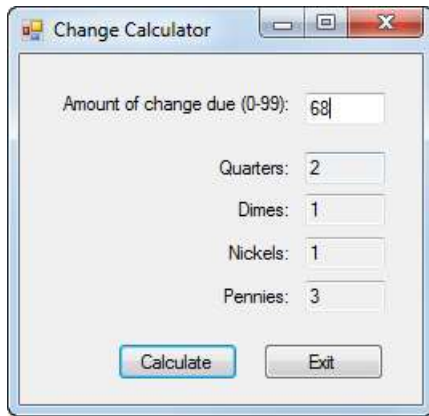
1. Start a new project named `Factorial` in the `Extra Exercises\Chapter 05\Factorial` directory.
2. Add labels, text boxes, and buttons to the default form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the Calculate 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

---

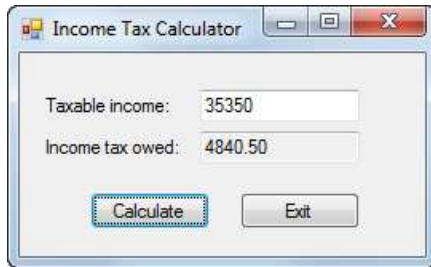
In this exercise, you'll develop a form that tells how many quarters, dimes, nickels, and pennies are needed to make change for any amount of change from 0 through 99 cents. One way to get the results is to use the division and modulus operators and to cast the result of each division to an integer.



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



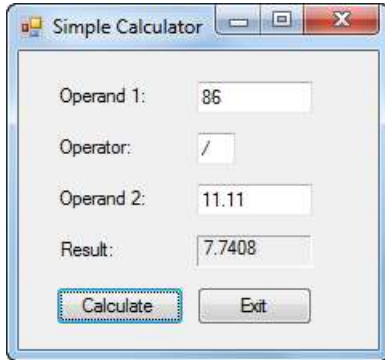
This is the 2015 table for the federal income tax on individuals that you should use for calculating the tax:

Taxable income		Income tax	
Over...	But not over...	Of excess over...	
\$0	\$9,225	\$0 plus 10%	\$0
\$9,225	\$37,450	\$922.50 plus 15%	\$9,225
\$37,450	\$90,750	\$5,156.25 plus 25%	\$37,450
\$90,750	\$189,300	\$18,481.25 plus 28%	\$90,750
\$189,300	\$411,500	\$46,075.25 plus 33%	\$189,300
\$411,500	\$413,200	\$119,401.25 plus 35%	\$411,500
\$413,200		\$119,996.25 plus 39.6%	\$413,200

1. Start a new project named TaxCalculator in the Extra Exercises\Chapter 05\TaxCalculator 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 4840.50.
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 Exercises\Chapter 06\SimpleCalculator 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:

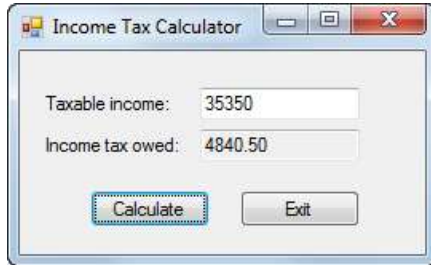
Argument	Description
<b>decimal operand1</b>	The value entered for the first operand.
<b>string operator1</b>	One of these four operators: +, -, *, or /.
<b>decimal operand2</b>	The value entered for the second operand.

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 the income tax calculator of extra exercise 5-3.

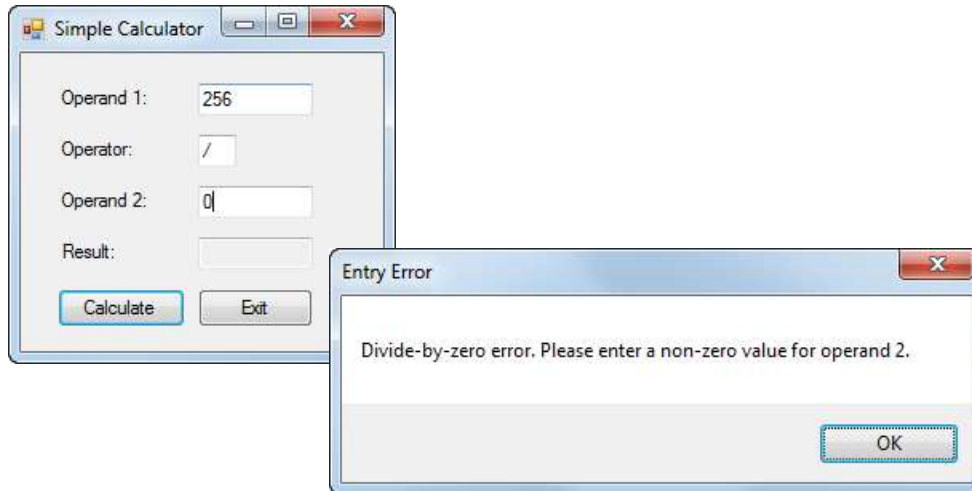


1. Open the TaxCalculator project in the Extra Exercises\Chapter 06\TaxCalculator directory and display the code for the form.
2. Code the declaration for a private method named CalculateTax that receives the income amount and returns the tax amount.
3. Move the if-else statement in the btnCalculate\_Click 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.

## Extra 7-1 Add exception handling to the simple calculator

---

In this exercise, you'll add exception handling to the Simple Calculator form of extra exercise 6-1.

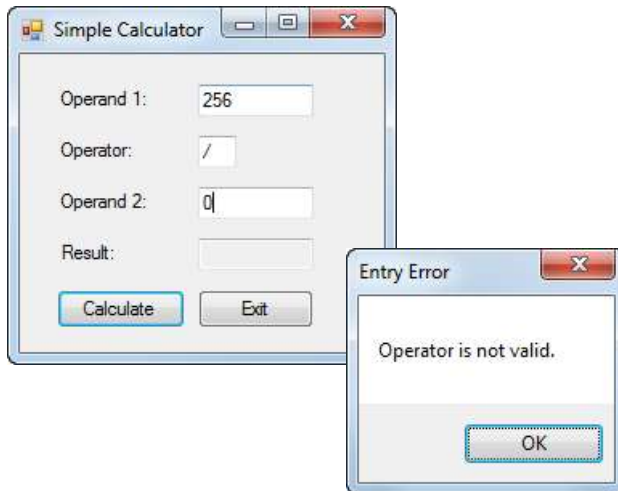


1. Open the SimpleCalculator project in the Extra Exercises\Chapter 07\SimpleCalculator With Exception Handling directory.
2. Add a try-catch statement in the btnCalculate\_Click event handler that will catch any exceptions that occur when the statements in that event handler are executed. If an exception 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 of extra exercise 7-1.

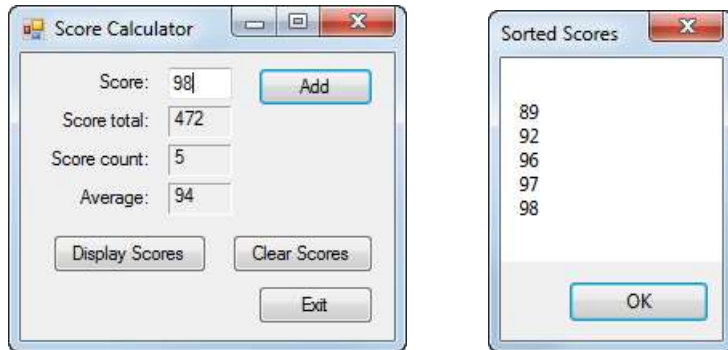


1. Open the SimpleCalculator project in the Extra Exercises\Chapter 07\SimpleCalculator With Data Validation directory.
2. Code methods named `IsPresent`, `IsDecimal`, and `IsWithinRange` that work like the methods described in chapter 7 of the book.
3. Code a method named `IsOperator` that checks that the text box that's passed to it contains a value of `+`, `-`, `*`, or `/`.
4. Code a method named `IsValidData` that checks that the Operand 1 and Operand 2 text boxes contain a decimal value between 0 and 1,000,000 (non-inclusive) and that the Operator text box contains a valid operator.
5. 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.
6. 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 of extra exercise 4-2 so it saves the scores the user enters in an array and then lets the user display the sorted scores in a dialog box.

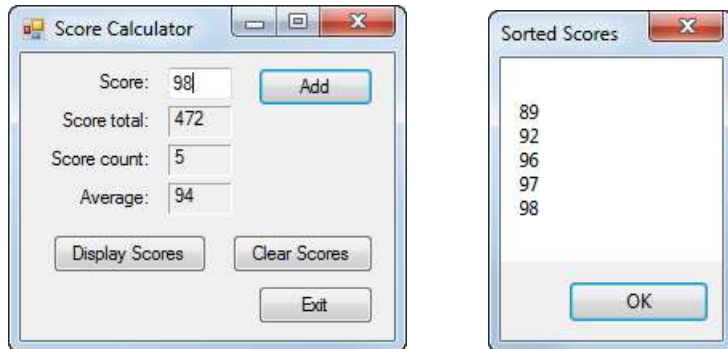


1. Open the ScoreCalculator project in the Extra Exercises\Chapter 08\ScoreCalculator With Array directory. This is the Score Calculator form from extra exercise 4-2 with data validation and exception handling added.
2. Declare a class variable for an array that can hold up to 20 scores.
3. Modify the 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.
4. Move the Clear Scores 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.
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.

## Extra 8-2 Display a test scores list

---

In this exercise, you'll modify the Score Calculator form of extra exercise 8-1 so the scores are stored in a list instead of an array.



1. Open the ScoreCalculator project in the Extra Exercises\Chapter 08\ScoreCalculator With List 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. Then, declare a local variable to store the count, and assign the Count property of the list to this variable.
4. Modify the Click event handler for the Clear Scores 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.



## 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.

### Open the project and implement the calculations

1. Open the Reservations project in the Extra Exercises\Chapter 09\Reservations 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

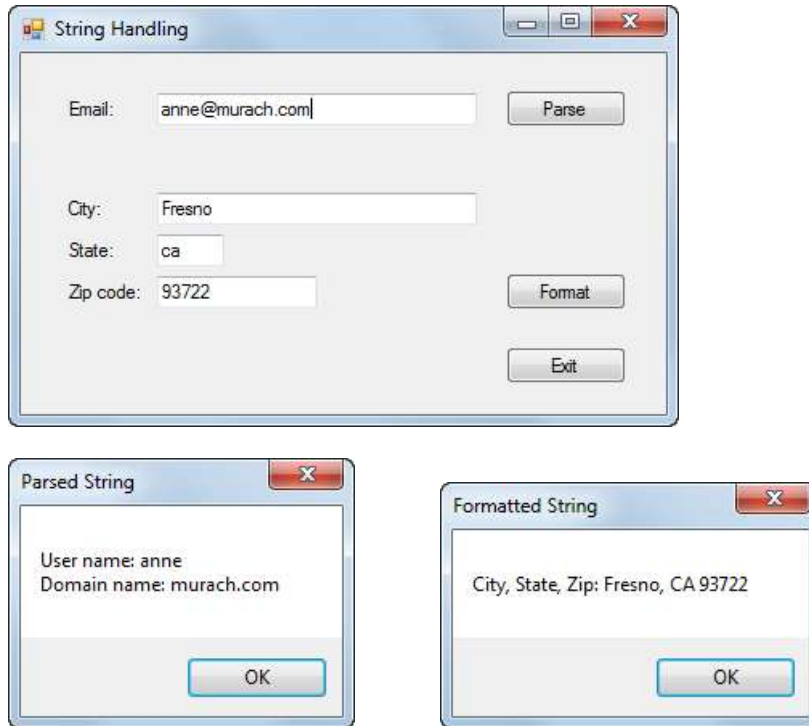
dates. These dates should be in a range from the current date to five years after the current date.

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

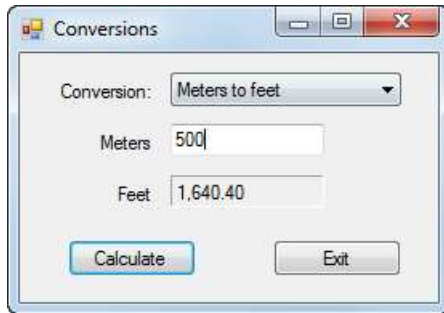
1. Open the StringHandling project in the Extra Exercises\Chapter 09\StringHandling directory.
2. Add code to parse the email address into two parts when the user clicks the Parse button: the user name before the @ sign and the domain name after the @ sign. Be to check that the email contains an @ sign 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 Exercises\Chapter 10\LengthConversions 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.

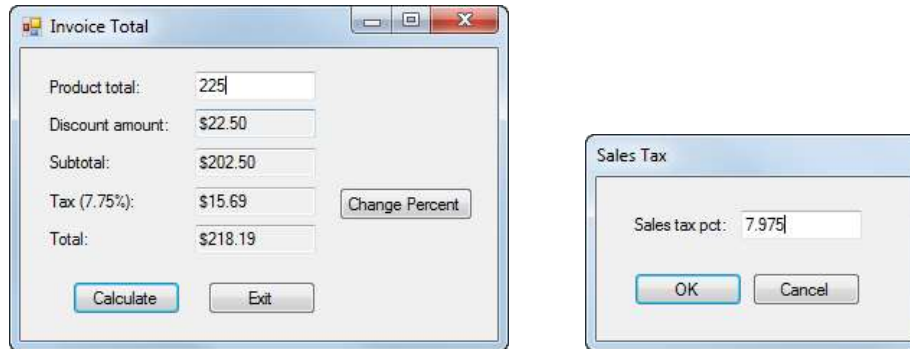
The application should provide for these main courses and add-ons:

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

1. Open the LunchOrder project in the Extra Exercises\Chapter 10\LunchOrder 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 Invoice Total 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 Exercises\Chapter10\InvoiceTotal directory.
2. Change the name of the existing form to frmInvoiceTotal.

### Create the Sales Tax form

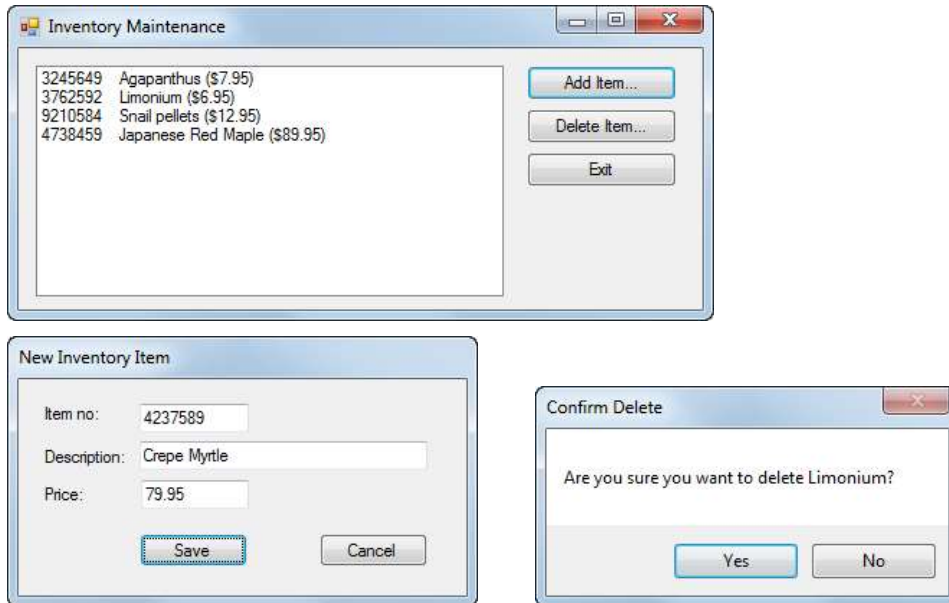
3. Add another form named frmSalesTax to the project.
4. Add a label, text box, and two buttons to the new form and set the properties of the form and its controls so they appear as shown above. When the user presses the Enter key, the Click event of the OK button should fire. When the user presses the 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 an Inventory Item 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 Exercises\InventoryMaintenance 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
<b>ItemNo</b>	Gets or sets an int that contains the item's number.
<b>Description</b>	Gets or sets a string that contains the item's description.
<b>Price</b>	Gets or sets a decimal that contains the item's price.
Method	Description
<b>GetDisplayText()</b>	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
<b>()</b>	Creates an InvItem object with default values.
<b>(itemNo, description, price)</b>	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 variable named `invItem` of type `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 variable named `invItems` of type `List<InvItem>` 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 Modify a list class to use an indexer, a delegate, an event, 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 Exercises\Chapter 13\InventoryMaintenance directory. This is an enhanced version of the Inventory Maintenance application from extra exercise 12-1 that uses a list class.
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 consists of 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

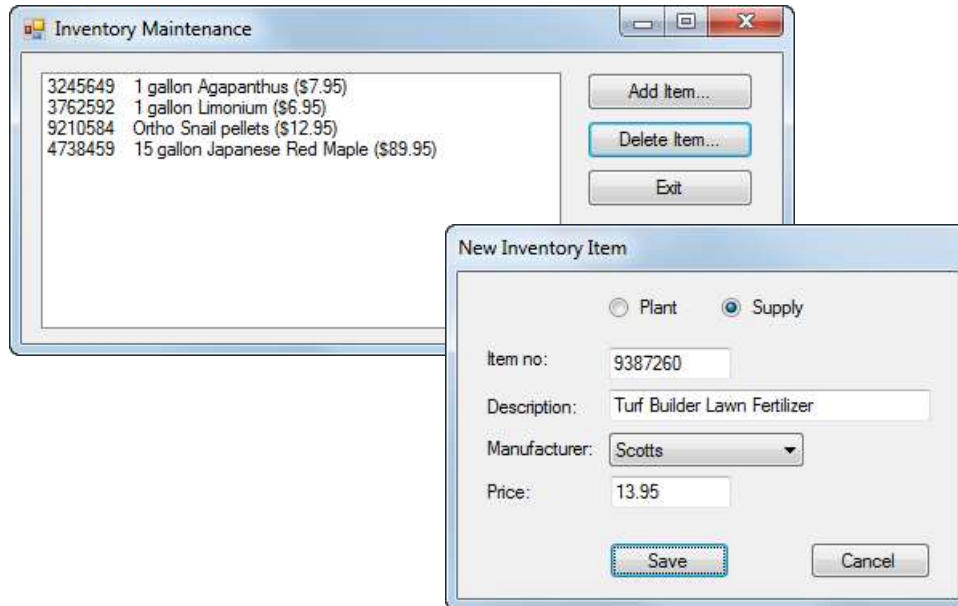
5. Add overloaded `+` and `-` operators to the `InvItemList` class that add and remove an inventory item from the inventory item list.
6. Modify the Inventory Maintenance form to use these operators instead of the `Add` and `Remove` methods. Then, test the application to be sure it still works.

### 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 with the Inventory Maintenance application

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 Exercises\Chapter 14\InventoryMaintenance` 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)

## **27** Extra exercises for *Murach's C# 2015*

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 from extra exercise 14-1.

1. Open the InventoryMaintenance project in the Extra Exercises\Chapter 15\InventoryMaint With IDisplayable 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, and copy the statement in the `GetDisplayText` method that this class contains. Then, delete 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 throw statement with the statement you copied from the original `GetDisplayText` method in step 3.
6. Test the application to be sure it still works.

## 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 from extra exercise 15-1.

1. Open the InventoryMaintenance project in the Extra Exercises\Chapter 15\InventoryMaint With IEnumerable 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 from extra exercise 12-1.

1. Open the InventoryMaintenance project in the Extra Exercises\Chapter 16\InventoryMaint With Documentation 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 the `InvItem` and `InvItemDB` classes of the Inventory Maintenance application from extra exercise 16-1. 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 Exercises\Chapter 16\InventoryMaint With Library` 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.

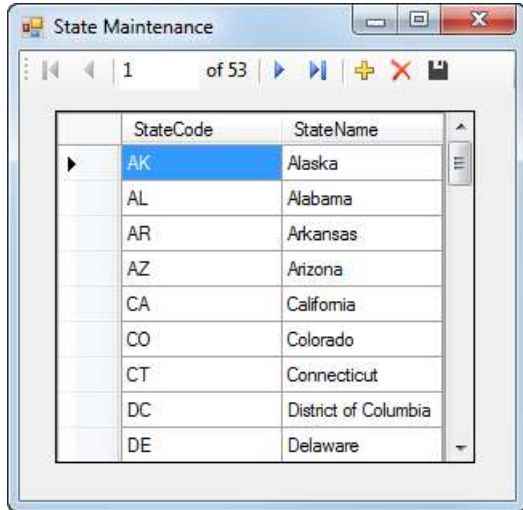
### Modify the Inventory Maintenance application to use the library

5. Open the `InventoryMaintenance` project in the `Extra Exercises\Chapter 16\InventoryMaint With Library` 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 18-1 Create a State Maintenance 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.



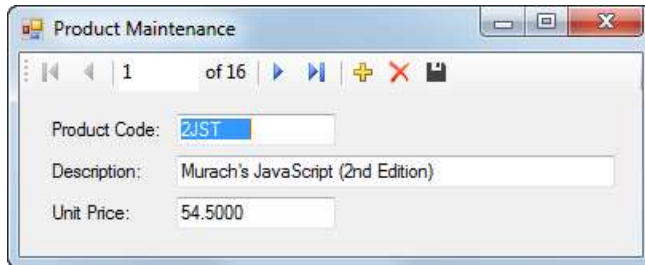
1. Start a new project named StateMaintenance in the Extra Exercises\Chapter 18\StateMaintenance directory.
2. Add the MMABooks.mdf file in the Extra Exercises\Database directory to the project. Then, use the Data Source Configuration Wizard to create a data source that includes the StateCode and StateName columns from the States table. When you're done, set the "Copy to Output Directory" property of the MMABooks.mdf file to "Copy if Newer".
3. Display the Data Sources window, and then drag the States table in the data source you just created onto the default form to add a DataGridView control that's bound to the table.
4. Set the properties of the form and control so they appear as shown above.
5. Test the application to be sure it works correctly.



## Extra 18-2 Create a Product Maintenance application

---

In this exercise, you'll create an application that uses a data source and text boxes to maintain the products in the Products table of the MMABooks database.

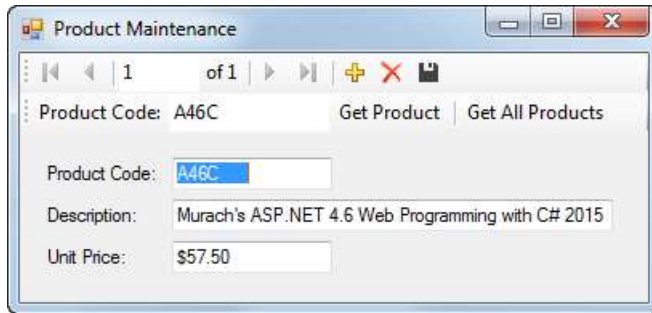


1. Start a new project named ProductMaintenance in the Extra Exercises\Chapter 18\ProductMaintenance directory.
2. Add the MMABooks.mdf file in the Extra Exercises\Database directory to the project. Then, use the Data Source Configuration Wizard to create a data source that includes the ProductCode, Description, and UnitPrice columns from the Products table. When you're done, set the "Copy to Output Directory" property of the MMABooks.mdf file to "Copy if Newer".
3. Display the Data Sources window, and then associate the columns in the Products table of the data source you just created with individual controls.
4. Drag the Products table onto the default form to add a text box for each column.
5. Set the properties of the form and controls so they appear as shown above.
6. Test the application to be sure it works correctly.

## Extra 19-1 Add a parameterized query to the Product Maintenance form

---

In this exercise, you'll add a Toolstrip to the Product Maintenance form of extra exercise 18-2 that provides for retrieving a product based on its code and for retrieving all products. In addition, you'll format the data in the Unit Price text box.



1. Open the ProductMaintenance project in the Extra Exercises\Chapter 19\ProductMaintenance directory. Then, increase the height of the form and move the controls on the form down to make room for another Toolstrip control.
2. Use the smart tag menu for one of the bound controls to add a parameterized query named FillByProductCode that will retrieve the row for a product with the specified product code. Add an if statement to the try block that was generated that checks the Count property of the binding source to be sure that the product is found. If it's not found, an error message should be displayed. Test the application to be sure this works.
3. Display the Items Collection editor for the Toolstrip control you just created. Then, modify the control so it appears as shown above.
4. Create an event handler for the Click event of the Get All Products button. Then, copy the statement in the Load event handler for the form that loads data into the Products table to the Click event handler, and delete the Load event handler so no data is loaded when the application starts. Test the application to see how this works.
5. Add a using directive for the System.Data.SqlClient namespace to the form. Then, add a try-catch statement to the Click event handler you just created that catches any SQL exceptions that occur. If an exception occurs, the catch block should display the exception number and message in a dialog box whose caption is the type of exception.
6. Format the Unit Price text box so the data it contains is displayed as currency with two decimal places. Then, test the application one more time.

## Extra 19-2 Create a Master/Detail form

In this exercise, you'll create a form that lets the user display an invoice, along with the line items for that invoice.

Product Code	Unit Price	Quantity	Item Total
A46V	\$57.50	1	\$57.50
DB2R	\$45.00	1	\$45.00
SQ12	\$57.50	1	\$57.50
VB15	\$56.50	1	\$56.50

### Create the project and add the database

1. Start a new project named InvoiceLineItems in the Extra Exercises\Chapter 19\InvoiceLineItems directory.
2. Add the MMABooks.mdf file in the Extra Exercises\Database directory to the project. Then, use the Data Source Configuration Wizard to create a data source that includes all of the columns except for CustomerID from the Invoices table and all of the columns from the InvoiceLineItems table. When you're done, set the "Copy to Output Directory" property of the MMABooks.mdf file to "Copy if Newer".

### Add the invoice information and toolbar

3. Drag the columns in the Invoices table onto the form as text boxes, and format them as shown above.
4. Delete the binding navigator toolbar, and delete the statement in the Load event handler for the form that fills the Invoices table.
5. Create a parameterized query named FillByInvoiceID that gets the invoice data for a specific invoice ID, and modify the Toolstrip control that's generated so it appears as shown above. Simplify the code that was generated for the Click event handler of the button on the Toolstrip control.
6. Test the application to be sure that the invoice data is displayed when you enter an invoice ID and click the Get Invoice button.

**Add the line item information**

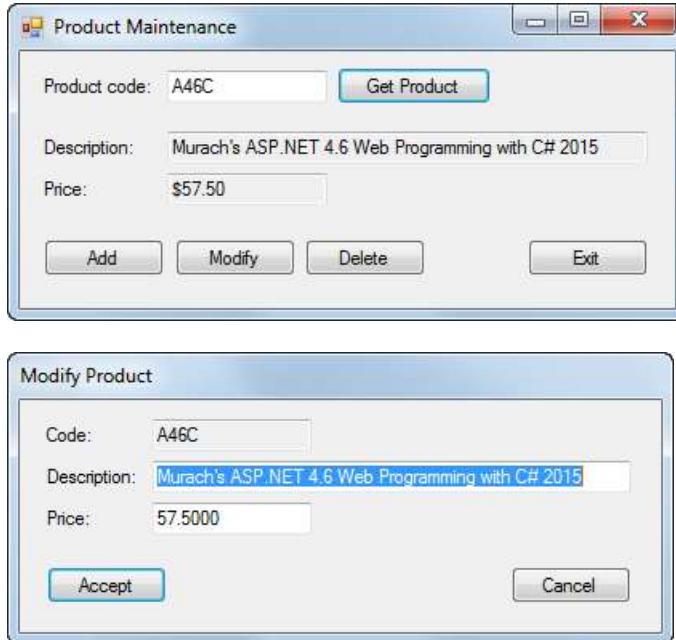
7. Drag the InvoiceLineItems table that's subordinate to the Invoices table onto the form as a DataGridView control. Then, disable adding, editing, and deleting for the DataGridView control, delete the InvoiceID column, and edit the remaining columns so they appear as shown above.
8. Create a parameterized query for the DataGridView control named FillByInvoiceID that gets the line items for a specific invoice ID, and delete the ToolStrip and event handler that are generated.
9. Modify the code in the event handler for the Click event of the FillByInvoiceIDToolStripButton so it gets the line items for the selected invoice. Be sure to check that an invoice with the invoice ID that was entered was found before you get the line items. If an invoice isn't found, display an appropriate error message.
10. Delete the Load event handler for the form so the line items aren't retrieved when the application starts. Then, test the application to be sure that the invoice data and the line item data for an invoice is displayed when you enter an invoice ID and click the Get Invoice button.

**Complete the application**

11. Add a catch block to the try-catch statement in the Click event handler to catch the exception that's thrown if the invoice ID isn't an integer. Then, add a using directive for the System.Data.SqlClient namespace, and add another catch block that catches a SqlException.
12. Make sure that all the properties for the form and control are set correctly and then test it again.

## Extra 20-1 Write the code for a Product Maintenance application

In this exercise, you'll use ADO.NET to write the data access code for an application that lets the user add, modify, and delete products.



### Open the project and add a class that gets a connection to the database

1. Open the ProductMaintenance project in the Extra Exercises\Chapter 20\ProductMaintenance directory. This project contains the two forms for the application, the Product and Validator classes, and the MMABooks.mdf database file.
2. Add a public class named MMABooksDB to the project. Then, add a static method named GetConnection that creates an SqlConnection object for the MMABooks database and then returns that connection. For this to work, you'll need to add a using directive for the System.Data.SqlClient namespace at the beginning of the class.

### Write the code to retrieve a product

3. Add another public class named ProductDB to the project, and add using directives for the System.Data and System.Data.SqlClient namespaces to this class.
4. Add a static method named GetProduct to the ProductDB class. This method should receive the product code of the product to be retrieved, and it should return a Product object for that product. If a product with the product code isn't found, this method should return null. Place the code that works with the database in the try block of a try-catch statement, include a catch block that catches and then throws any SQLException that occurs, and include a finally block that closes the connection.

5. Display the code for the Product Maintenance form, and add a statement to the `GetProduct` method that calls the `GetProduct` method of the `ProductDB` class to get the product with the product code the user enters. Assign the product that's returned to the product variable.
6. Test the application to be sure that the data for a product is displayed when you enter a product code and click the Get Product button.

### **Write the code to update a product**

7. Add a static method named `UpdateProduct` to the `ProductDB` class. This method should receive two `Product` objects. The first one should contain the original product data and should be used to provide for optimistic concurrency. The second one should contain the new product data and should be used to update the product row. This method should also return a Boolean value that indicates if the update was successful. Like the `GetProduct` method, this method should include a try-catch statement with a catch block that catches and then throws any `SqlException` that occurs, and a finally block that closes the connection.
8. Display the code for the Add/Modify Product form, and locate the event handler for the Click event of the Accept button. Add code to this event handler that calls the `UpdateProduct` method of the `ProductDB` class. If this method returns a true value, the event handler should assign the `newProduct` object to the product object and then set the `DialogResult` property of the form to `DialogResult.OK`. Otherwise, it should display an error message indicating that another user has updated or deleted the product, and then set the `DialogResult` property to `DialogResult.Retry`.
9. Test the application to be sure the modify operation works correctly.

### **Write the code to add a product**

10. Add a static method named `AddProduct` to the `ProductDB` class. This method should receive a `Product` object with the data for the new product, and it should return a Boolean value that indicates if the add operation was successful. Be sure to include a try-catch statement with a catch block that catches and throws an `SqlException` and a finally block that closes the connection.
11. Display the code for the Add/Modify Product form, and add code to the event handler for the Click event of the Accept button that calls the `AddProduct` method of the `ProductDB` class. If this method returns a true value, the event handler should set the `DialogResult` property of the form to `DialogResult.OK`. Otherwise, it should display an error message indicating that another product with the same code already exists, and then set the `DialogResult` property to `DialogResult.Retry`.
12. Test the application to be sure the add operation works correctly.

### **Write the code to delete a product**

13. Add a static method named `DeleteProduct` to the `ProductDB` class. This method should receive a `Product` object with the data for the product to be deleted, and it should return a Boolean value that indicates if the delete operation was successful. The `Product` object should be used to provide for optimistic concurrency. Include a try-catch statement like the ones in the other methods of this class.

14. Display the code for the Product Maintenance form, and add code to the event handler for the Click event of the Delete button that calls the DeleteProduct method of the ProductDB class. If this method returns a true value, the event handler should call the ClearControls method of the form. Otherwise, it should display a message indicating that another user has updated or deleted the product. Then, it should call the GetProduct method of the form to determine if the product has been deleted. If it hasn't, the event handler should call the DisplayProduct method of the form. Otherwise, it should call the ClearControls method of the form.
15. Test the application to be sure the delete operation works correctly. Note, however, that you won't be able to delete a product if it's associated with one or more rows in the InvoiceLineItems table.

## Extra 21-1 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 Exercises\Chapter 21\InventoryMaint With Text File 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 21-2 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 Exercises\Chapter 21\InventoryMaint With Binary File 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 text 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 22-1 Work with an XML file

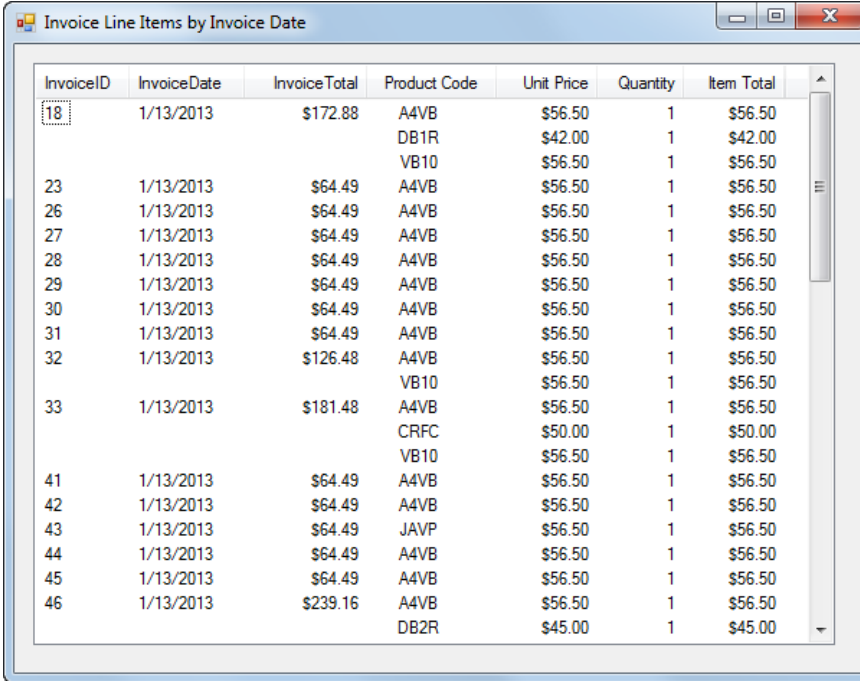
---

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

1. Open the InventoryMaintenance project in the Extra Exercises\Chapter 21\InventoryMaint With XML File directory, and display the code for the `InvItemDB` class.
2. Add code to the `GetItems` method that creates an `XmlReaderSettings` object that ignores white space and comments. Then, create an `XmlReader` object for the `InventoryItems.xml` file that's included in the project and that uses the reader settings. (The `Path` constant contains the path to the XML file.)
3. Add code that reads each `Item` element, stores the contents of the child elements in an `InvItem` object, and adds the object to the `List<InvItem>` object. Be sure to check that the file contains at least one `Item` element before the loop that processes these elements is executed. Then, close the `XmlReader` object.
4. Add code to the `SaveItems` method that creates an `XmlWriterSettings` object that indents each element four spaces. Then, create an `XmlWriter` object for the `InventoryItems.xml` file that uses the writer settings.
5. Add code that writes an XML declaration line and a start tag for the `Items` element. Then, add code that creates an `Item` element for each `InvItem` object in the `List<InvItem>` object. Each property of the `InvItem` object should be saved as a child element with the same name as the property.
6. Add code that writes the end tag for the `Items` element and then closes the `XmlWriter` object.
7. Test the application to be sure it works correctly.

## Extra 23-1 Use LINQ to create an Invoice Line Items application

In this exercise, you'll use LINQ to join the data in two `List<>` objects and then display that data in a `ListView` control.



InvoiceID	InvoiceDate	InvoiceTotal	Product Code	Unit Price	Quantity	Item Total
18	1/13/2013	\$172.88	A4VB	\$56.50	1	\$56.50
			DB1R	\$42.00	1	\$42.00
			VB10	\$56.50	1	\$56.50
23	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
26	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
27	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
28	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
29	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
30	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
31	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
32	1/13/2013	\$126.48	A4VB	\$56.50	1	\$56.50
			VB10	\$56.50	1	\$56.50
33	1/13/2013	\$181.48	A4VB	\$56.50	1	\$56.50
			CRFC	\$50.00	1	\$50.00
			VB10	\$56.50	1	\$56.50
41	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
42	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
43	1/13/2013	\$64.49	JAVP	\$56.50	1	\$56.50
44	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
45	1/13/2013	\$64.49	A4VB	\$56.50	1	\$56.50
46	1/13/2013	\$239.16	A4VB	\$56.50	1	\$56.50
			DB2R	\$45.00	1	\$45.00

### Design the form

1. Open the `InvoiceLineItems` project in the `Extra Exercises\Chapter 23\InvoiceLineItems` directory. This project contains the Invoice Line Items form, along with the business and database classes and database files needed by the application.
2. Add a `ListView` control to the form, and set the `View` property of this control to `Details`.
3. Use the smart tag menu for the `ListView` control to display the `ColumnHeader Collection Editor`. Then, define the column headers for this control so they appear like the first and the last four shown above.

### Add code to display the line item data

4. Add an event handler for the `Load` event of the form. Then, use the `GetLineItems` method in the `LineItemDB` class to get a `List<LineItem>` object, and store this list in a variable.
5. Define a query expression that returns all the line items from the line item list. The `select` clause for this query expression should select entire line items.
6. Use a `foreach` statement to execute the query and load the results into the `ListView` control.
7. Test the application to be sure it displays the line items correctly.

### **Enhance the application to include invoice data**

8. Add two more columns to the ListView control for displaying the invoice date and invoice total. (To get these columns to display before the last four columns, you'll need to set their DisplayIndex properties to 1 and 2.) Then, add a statement to the Load event handler of the form that uses the GetInvoices method of the InvoiceDB class to get a List<Invoice> object, and store the list in a variable.
9. Modify the query expression so it joins the data in the invoice list with the data in the line item list, so it sorts the results by invoice date, and so only the fields that are needed by the form are returned by the query.
10. Modify the foreach statement so it adds the invoice ID, invoice date, and invoice total to the ListView control.
11. Test the application to be sure it displays the invoice data correctly.

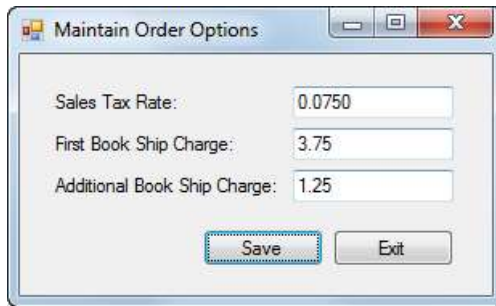
### **Enhance the application so it doesn't repeat invoice information**

12. Declare a variable outside the foreach statement to hold an invoice ID, and initialize this variable to 0. Then, add code within the foreach statement that checks if the invoice ID for the current item is equal to the invoice ID in the variable you just declared. If they aren't equal, the invoice ID, invoice date, and invoice total should be added to the List View control, and the invoice ID variable should be set to the value of the invoice ID for the current item. Otherwise, spaces should be added to the ListView control for these fields.
13. Test this change to be sure it works correctly.

## Extra 24-1 Use the Entity Framework to create an Order Options Maintenance application

---

In this exercise, you'll use the Entity Framework to create an application that lets you update the data in a table of order options. The OrderOptions table is one of the tables in a database named MMABooksEF. This database is identical to the MMABooks database that was used in the book applications, the book exercises, and these extra exercises except that a primary key has been added to the OrderOptions table so you can update it using EF.



### Open the project and create the Entity Data Model

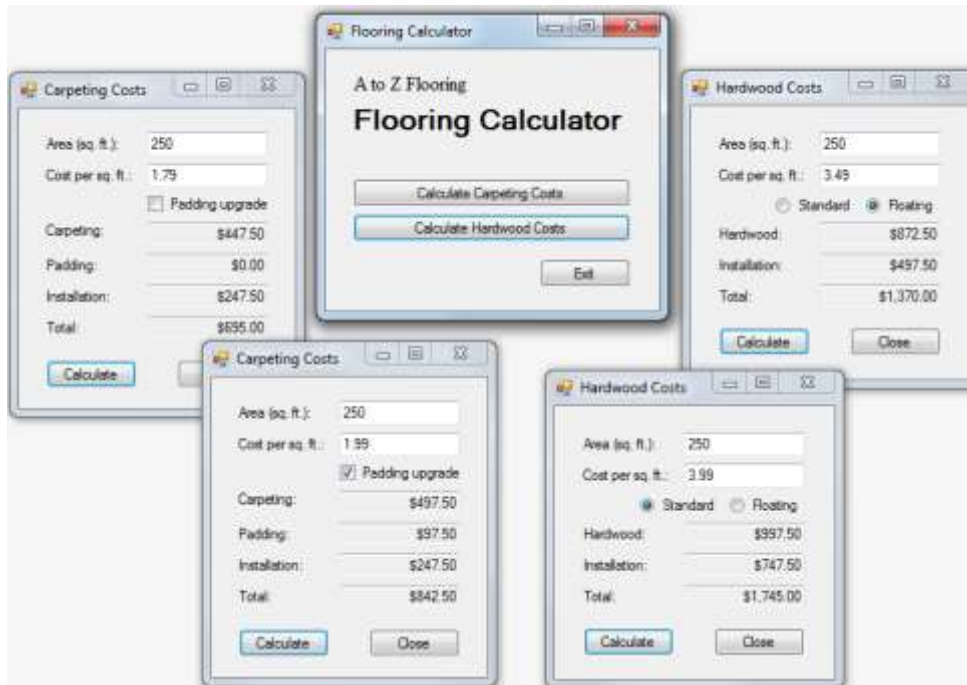
1. Open the OrderOptionsMaintenance project in the Extra Exercises\Chapter 24\OrderOptionsMaintenance directory. This project contains the Order Options Maintenance form, along with the MMABooksEF database that contains the OrderOptions table, a Validator class, and code for validating the data the user enters and catching exceptions.
2. Use the Entity Data Model Wizard to create an Entity Data Model that contains just the OrderOptions table in the MMABooksEF database.

### Write the code to retrieve and update the options

3. Declare two class-level variables: one that stores an instance of the object context for the Entity Data Model and one for an OrderOption object that will be used to store the results of the query that retrieves the data from the OrderOptions table.
4. In the Load event handler for the form, code a query expression that uses LINQ to Entities to get the data from the OrderOptions table. Because this table contains a single row, you should use the Single method on the query. Assign the result to the OrderOption object that you defined in step 2. Then, assign the properties of the OrderOption object to the text boxes on the form.
5. Within the try block in the Click event handler for the Save button, 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.
6. Test the application to be sure it works.

## Extra 25-1 Create an SDI application

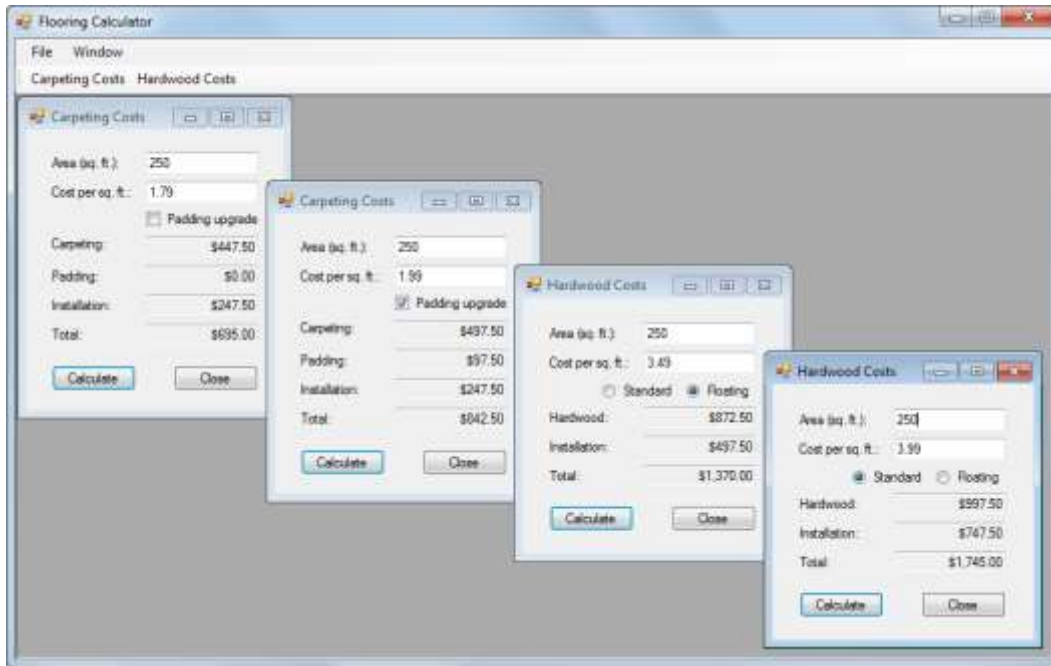
In this exercise, you'll create an application with a single-document interface that consists of a startup form and two additional forms.



1. Open the CalculateFlooringCosts project in the Extra Exercises\Chapter 24\CalculateFlooringCostsSDI directory. Then, review the code for the Carpeting Costs and Hardwood Costs forms so you understand how they work.
2. Create a startup form with three buttons. The first one should create and display a new instance of the Carpeting Costs form each time you click it, the second one should create and display a new instance of the Hardwood Costs form each time you click it, and the third one should exit from the application. Then, modify the program.cs file so the startup form is displayed when the application starts.
3. Add two labels to the startup form and set their properties so they appear as shown above. The first label uses a 12-point Times New Roman font, and the second label uses an 18-point Microsoft Sans Serif bold font.
4. Change the Exit buttons on the Carpeting Costs and Hardwood Costs forms to Close buttons.
5. Test the application to be sure it works correctly.

## Extra 25-2 Create an MDI application

In this exercise, you'll create an application with a multiple-document interface that consists of a parent form and two child forms.



### Open the project and add a parent form

1. Open the CalculateFlooringCosts project in the Extra Exercises\Chapter 24\CalculateFlooringCostsMDI directory. Then, review the code for the Carpeting Costs and Hardwood Costs forms so you understand how they work.
2. Add a form to the project, and set the `IsMdiContainer` property of this form to true to identify it as the parent form. Resize the form so it's large enough to hold several child forms, and modify the `program.cs` file so this form is displayed when the application starts.

### Add the File menu to the parent form

3. Add a `MenuStrip` control to the parent form. Then, use the Menu Designer to add a File menu to the parent form. This menu should include four menu items that display a Carpeting Costs form, display a Hardwood Costs form, close the active child form, and exit from the application. Give each menu item an appropriate name, include access keys for the menu and menu items, and include a separator bar between the Close and Exit items.
4. Add an event handler for the Click event of each item in the File menu. The event handlers for the New Carpeting Costs and New Hardwood Costs items should create a new instance of the appropriate form, set the parent form to the current form, and display the form. The event handler for the Close item should close the active child form, if there is one. And the event handler for the Exit item should exit from the application.
5. Test the application to be sure that the items in the File menu work as expected.

### **Add the Window menu to the parent form**

6. Add a Window menu to the right of the File menu. Then, add three items to this menu that will let the user arrange the forms in a cascaded, vertical, or horizontal layout.
7. Give each item in the Window menu an appropriate name, and then add an event handler for the Click event of each item that arranges the child forms appropriately.
8. Set the `MdiWindowListItem` property of the menu strip so the Window menu will display a list of the open child forms.
9. Test the application to be sure that the items in the Window menu work as expected.

### **Add a toolbar to the parent form**

10. Add a `ToolStrip` control to the parent form. Then, add two buttons to this toolbar with the text “Carpeting Costs” and “Hardwood Costs” on them. Be sure to assign appropriate names to these buttons.
11. Add an event handler for the Click event of each button that uses the Click event handler for the associated menu to display the form.
12. Test the application to be sure that the toolbar buttons work.

### **Add tooltips and context-sensitive help to the child forms**

13. Add a `ToolTip` control to each child form. Then, add a tooltip for each text box, check box, and radio button on these forms. For the check box and radio buttons, you should display the cost associated with selecting that control.
14. Add a `HelpProvider` control to each child form. Then, add context-sensitive help for the Area text box on each form that explains how to calculate the area (length in feet x width in feet).
15. Add context-sensitive help for each form that describes what the form does.
16. Test the application to be sure that the tooltips and context-sensitive help are displayed.