CHAPTER

# 4

# Decisions and Conditions

**at the completion of this chapter, you will be able to . . .**

1. Use `if` statements to control the flow of logic.
2. Understand and use nested `if` statements.
3. Read and create action diagrams that illustrate the logic in a selection process.
4. Evaluate Boolean expressions using the relational or comparison operators.
5. Combine expressions using logical operators && (and), || (or), and ! (not).
6. Test the Checked property of radio buttons and check boxes.
7. Perform validation on numeric fields using `if` statements.
8. Use a `switch` structure for multiple decisions.
9. Use one event handler to respond to the events for multiple controls.
10. Call an event handler from other methods.
11. Create message boxes with multiple buttons and choose alternate actions based on the user response.
12. Debug projects using breakpoints, stepping program execution, and displaying intermediate results.

In this chapter, you will learn to write applications that can take one action or another, based on a condition. For example, you may need to keep track of sales separately for different classes of employees, different sections of the country, or different departments. You also will learn alternate techniques for checking the validity of input data and how to display multiple buttons in a message box and take different actions depending on the user response.

## if Statements

A powerful capability of the computer is its ability to make decisions and to take alternate courses of action based on the outcome.

A decision made by the computer is formed as a question: Is a given condition true or false? If it is true, do one thing; if it is false, do something else.

| | |
|---|---|
| if *the sun is shining* | (condition) |
| *go to the beach* | (action to take if condition is true) |
| else | |
| *go to class* | (action to take if condition is false) |
| (See Figure 4.1.) | |

or

| | |
|---|---|
| if *you don't succeed* | (condition) |
| *try, try again* | (action) |
| (See Figure 4.2.) | |

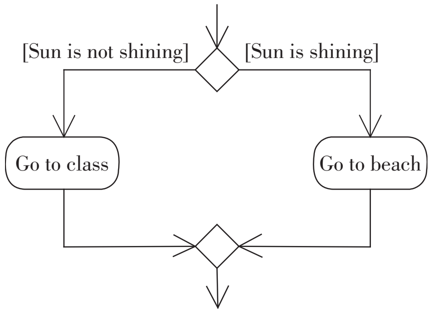


**Figure 4.1**

*The logic of an* `if`/`else` *statement in Unified Modeling Language (UML) activity diagram form.*

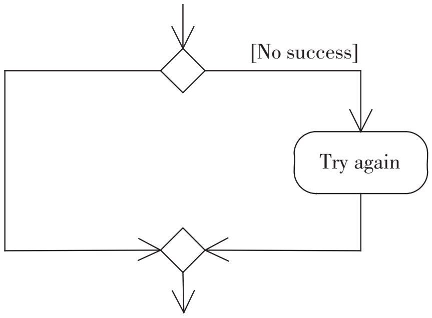


**Figure 4.2**

*The logic of an* `if` *statement without an* `else` *action in UML activity diagram form.*

Notice in the second example that no action is specified if the condition is not true.

In an **`if` statement,** when the condition is *true,* only the statement following the `if` is executed. When the condition is *false,* only the statement following the `else` clause, if present, is executed. You can use a block (braces) to include multiple statements in the `if` or `else` portion of the statement. Although not required, it is considered good programming practice to always use braces, even when you have only one statement for the `if` or `else`.

### if Statement—General Form

General Form

```
if (condition)
{
    // Statement(s)
}
[else
{
    // Statements(s)
}]
```

Only the first statement following an `if` or an `else` is considered a part of the statement unless you create a block of statements using braces. Notice that the `if` and `else` statements do not have semicolons. If you accidentally place a semicolon after the `if` or `else`, that terminates the statement and any statement(s) following will execute unconditionally.

The statements under the `if` and `else` clauses are indented for readability and clarity.

### if Statement—Examples

When the number of units in unitsDecimal is less than 32, select the radio button for Freshman; otherwise, make sure the radio button is deselected (see Figure 4.3). Remember that when a radio button is selected, the Checked property has a Boolean value of *true*.

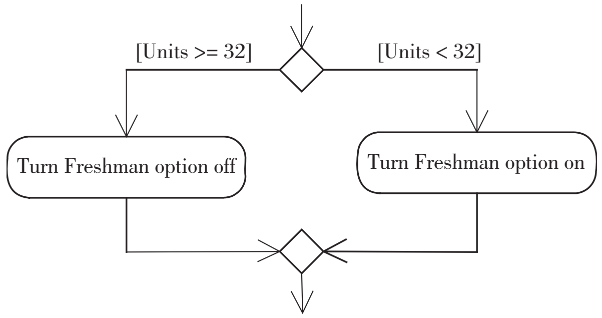
Examples

```
unitsDecimal = decimal.Parse(unitsTextBox.Text);
if (unitsDecimal < 32m)
{
    freshmanRadioButton.Checked = true;
}
else
{
    freshmanRadioButton.Checked = false;
}
```

When you type an `if` statement and press Enter, the editor places the insertion point on a blank line, indented from the `if`. If you have multiple statements on either the `if` or the `else`, you must place those statements in braces; otherwise, only the first statement belongs to the `if`. The editor properly places the

Figure 4.3

*The `if` statement logic in UML activity diagram form. If the number of units is fewer than 32, the Freshman radio button will be selected; otherwise the Freshman radio button will be deselected.*



braces directly under the `if` or `else` when you type the brace at the position of the insertion point on the blank line.

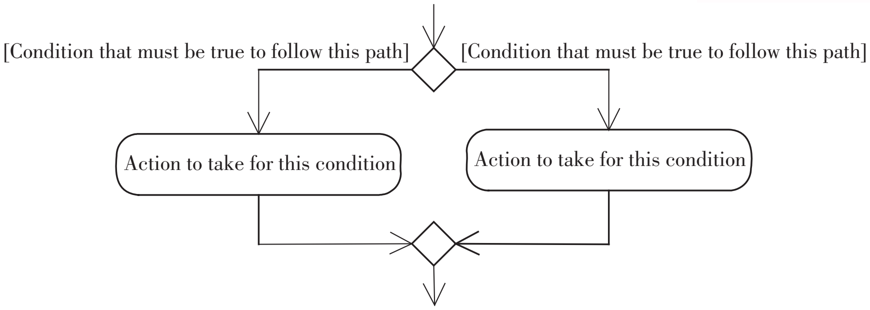
## Charting if Statements

A Unified Modeling Language (UML) activity diagram is a useful tool for showing the logic of an `if` statement. It has been said that one picture is worth a thousand words. Many programmers find that a diagram helps them organize their thoughts and design projects more quickly.

The UML specification includes several types of diagrams. The activity diagram is a visual planning tool for decisions and actions for an entire application or a single method. The diamond-shape symbol (called a *decision symbol*) represents a condition. The branches from the decision symbol indicate which path to take for different results of the decision (Figure 4.4).

Figure 4.4

*The UML activity diagram symbols used for program decisions and activities.*


[Condition that must be true to follow this path]
[Condition that must be true to follow this path]
Action to take for this condition
Action to take for this condition

# Boolean Expressions

The test in an `if` statement is a **Boolean expression**, which evaluates as *true* or *false*. To form Boolean expressions, also referred to as ***conditions***, you use **relational operators** (Table 4.1), also called ***comparison operators***. The comparison is evaluated and the result is either *true* or *false*.

Boolean expressions can be formed with numeric variables and constants, string variables and constants, object properties, and arithmetic expressions.

Table 4.1

The Relational Operators

| Symbol | Relation tested | Examples |
|---|---|---|
| > | greater than | `decimal.Parse(amountTextBox.Text) > limitDecimal`<br>`correctInteger > 75` |
| < | less than | `int.Parse(salesTextBox.Text) < 10000`<br>`saleDecimal < limitDecimal` |
| == | equal to | `passwordTextBox.Text == "101"`<br>`nameTextBox.Text == nameString` |
| != | not equal to | `freshmanRadioButton.Checked != true`<br>`nameTextBox.Text != ""` |
| >= | greater than or equal to | `int.Parse(quantityTextBox.Text) >= 500` |
| <= | less than or equal to | `countInteger <= maximumInteger` |

However, it is important to note that comparisons must be made on like types; that is, strings can be compared only to other strings, and numeric values can be compared only to other numeric values, whether a variable, constant, property, or arithmetic expression.

## Comparing Numeric Variables and Constants

When numeric values are involved in a test, an algebraic comparison is made; that is, the sign of the number is taken into account. Therefore, negative 20 is less than 10, and negative 2 is less than negative 1.

In a comparison, the double equal sign is used to test for equality. An equal sign (=) means replacement in an assignment statement. For example, the Boolean expression in the statement

✓ TIP

If you accidentally use an = (assignment) instead of an == (equal to) operator in a comparison, the compiler generates an error. ■

```
if (decimal.Parse(priceTextBox.Text) == maximumDecimal)
```

means "Is the current numeric value stored in priceTextBox.Text equal to the value stored in maximumDecimal?"

### Sample Comparisons

| alphaInteger | bravoInteger | charlieInteger |
|---|---|---|
| 5 | 4 | –5 |

| Boolean expression | Evaluates |
|---|---|
| `alphaInteger == bravoInteger` | *false* |
| `charlieInteger < 0` | *true* |
| `bravoInteger > alphaInteger` | *false* |
| `charlieInteger <= bravoInteger` | *true* |
| `alphaInteger >= 5` | *true* |
| `alphaInteger != charlieInteger` | *true* |

## Comparing Character Data

You can use the relational operators to compare character data stored in the char data type. As you recall, char variables hold only a single character. The determination of which character is less than another is based on the code used to store characters internally in the computer. C# stores characters using the 16-bit Unicode, which is designed to hold characters in any language, such as Japanese and Chinese. However, the Latin alphabet, numerals, and punctuation have the same values in Unicode as they do in the **ANSI code,** which is the most common coding method used for microcomputers. ANSI has an established order (called the *collating sequence*) for all letters, numbers, and special characters. Note in Table 4.2 that the uppercase letters are lower than the lowercase letters, and all numeric digits are less than all letters. Some special symbols are lower than the numbers and some are higher, and the blank space is lower than the rest of the characters shown.

You can compare a char variable to a literal, another char variable, an escape sequence, or the code number. You can enclose char literals in single or double quotes, which then compares to the value of the literal. Without the quote, you are comparing to the code number (from the code chart).

**TIP**

**A**lthough the editor helps you indent properly as you enter an `if` statement, when you modify code, the indentation can get messed up. Take the time to fix any indentation errors—you will save yourself debugging time and the indentation helps to visualize the intended logic. You can use *Edit / Advanced / Format Document* or *Format Selection* to reformat automatically. ■

```
(sexChar == 'F')    // Compare to the uppercase letter 'F'.
(codeChar != '\0') // Compare to a null character (character 0).
(codeChar == '9')  // Compare to the digit 9.
(codeChar == 9)    // Compare to the ANSI code 9 (the Tab character).
```

**Selected ANSI Codes**

**Table 4.2**

| Code | Value | Code | Value | Code | Value |
|---|---|---|---|---|---|
| 0 | Null | 38 | & | 60 | < |
| 8 | Backspace | 39 | ' | 61 | = |
| 9 | Tab | 40 | ( | 62 | > |
| 10 | Linefeed | 41 | ) | 63 | ? |
| 12 | Formfeed | 42 | * | 64 | @ |
| 13 | Carriage return | 43 | + | 65–90 | A–Z |
| 27 | Escape | 44 | , | 91 | [ |
| 32 | Space | 45 | - | 92 | \ |
| 33 | ! | 46 | . | 93 | ] |
| 34 | " | 47 | / | 94 | ^ |
| 35 | # | 48–57 | 0–9 | 95 | _ |
| 36 | $ | 58 | : | 96 | ` |
| 37 | % | 59 | ; | 97–122 | a–z |

### Examples

```
char upperCaseChar = 'C';
char lowerCaseChar = 'c';
```

The expression `(lowerCaseChar > upperCaseChar)` is *true* because the ANSI code for `'C'` is 67, while the code for `'c'` is 99.

```
char questionMarkChar = '?';
char exclamationChar = '!';
```

The expression `(questionMarkChar < exclamationChar)` is *false* because the ANSI code for `'?'` is 63 while the ANSI code for `'!'` is 33.

## Comparing Strings

You can use the equal to (==) and not equal to (!=) relational operators for comparing strings. The comparison begins with the leftmost character and proceeds one character at a time from left to right. As soon as a character in one string is not equal to the corresponding character in the second string, the comparison is terminated and the Boolean expression returns *false*.

### Example

```
string name1String = "Joan";
string name2String = "John";
```

The expression

```
if (name1String == name2String)
```

evaluates to *false*. The a in Joan is lower ranking than the h in John. However, when you need to compare for *less than* or *greater than,* such as arranging strings alphabetically, you cannot use the relational operators, but you *can* use the `CompareTo` method.

### The CompareTo Method

Use the `CompareTo` method to determine *less than* or *greater than*. The `CompareTo` method returns an integer with one of three possible values.

```
aString.CompareTo(bString)
```

If the two strings are equal, the method returns zero; when aString is greater than bString, a positive number returns. A negative number is returned when bString is greater than aString. To use this method effectively, you can set up a condition using the return value and a relational operator.

```
if (aString.CompareTo(bString) == 0) // Are the strings equal?
if (aString.CompareTo(bString) != 0) // Are the strings different?
if (aString.CompareTo(bString) > 0)  // Is aString greater than bString?
```

### Example

```
string word1String = "Hope";
string word2String = "Hopeless";
// Compare the strings.
if (word1String.CompareTo(word2String) < 0)
    // Display a message -- What will it be?
```

Will the result of the preceding comparison test be *true* or *false*? Before reading any further, stop and figure it out.

Ready? When one string is shorter than the other, the comparison proceeds as if the shorter string is padded with blanks to the right of the string, and the blank space is compared to a character in the longer string. So in the `CompareTo` method, `"Hope"` is less than `"Hopeless"`, the method returns a negative number, and the condition is *true*.

### Example

```
string car1String = "300ZX";
string car2String = "Porsche";
```

The expression

```
if (car1String.CompareTo(car2String) > 0)
```

evaluates *false*. When the number 3 is compared to the letter P, the 3 is lower—all numbers are lower than all letters. The `CompareTo` method returns a negative number and the comparison is *false*.

## Feedback 4.1

| count1Integer | count2Integer | count3Integer | word1TextBox.Text | word2TextBox.Text |
|---|---|---|---|---|
| 5 | 5 | –5 | “Bit” | “bit” |

Determine which Boolean expressions will evaluate *true* and which ones will evaluate *false*.

1. `count1Integer >= count2Integer`
2. `count3Integer < 0`
3. `count3Integer < count2Integer`
4. `count1Integer != count2Integer`
5. `count1Integer + 2 > count2Integer + 2`
6. `word1TextBox.Text == word2TextBox.Text`
7. `word1TextBox.Text != ""`
8. `word1TextBox.Text == "bit"`
9. `"2" != "Two"`
10. `'$' <= '?'`
11. `word1TextBox.Text.CompareTo(word2TextBox.Text) > 0`

## Testing for True or False

You can use shortcuts when testing for *true* or *false*. C# evaluates the expression in an `if` statement. If the condition is a Boolean variable or property, it holds the values *true* or *false*.

Example

```
if (blueRadioButton.Checked == true)
```

is equivalent to

```
if (blueRadioButton.Checked)
```

## Comparing Uppercase and Lowercase Characters

When comparing strings, the case of the characters is important. An uppercase *Y* is not equal to a lowercase *y*. Because the user may type a name or word in uppercase, in lowercase, or as a combination of cases, we must check user input for all possibilities. The best way is to use the **`ToUpper`** and **`ToLower`** **methods** of the string class, which return the uppercase or lowercase equivalent of a string, respectively.

### The ToUpper and ToLower Methods—General Form

General Form

```
TextString.ToUpper()
TextString.ToLower()
```

### The ToUpper and ToLower Methods—Examples

Examples

| nameTextBox.Text Value | nameTextBox.Text.ToUpper() | nameTextBox.Text.ToLower() |
|---|---|---|
| Richard | RICHARD | richard |
| PROGRAMMING | PROGRAMMING | programming |
| Robert Jones | ROBERT JONES | robert jones |
| hello | HELLO | hello |

An example of a Boolean expression using the `ToUpper` method follows.

```
if (nameTextBox.Text.ToUpper() == "PROGRAMMING")
{
    // Do something.
}
```

Note that when you convert nameTextBox.Text to uppercase, you must compare it to an uppercase literal ("PROGRAMMING") if you want it to evaluate as *true*.

## Compound Boolean Expressions

You can use **compound Boolean expressions** to test more than one condition. Create compound expressions by joining conditions with **logical operators**, which compare each expression and return a boolean result. The logical operators are `||` (or), `&&` (and), and `!` (not).

| Logical operator | Meaning | Example | Explanation |
|---|---|---|---|
| `||` (or) | If one expression or both expressions are *true*, the entire expression is *true*. | `int.Parse(numberLabel.Text) == 1 || int.Parse(numberLabel.Text) == 2` | Evaluates *true* when numberLabel.Text is either "1" or "2". |
| `&&` (and) | Both expressions must be *true* for the entire expression to be *true*. | `int.Parse(numberTextBox.Text) > 0 && int.Parse(numberTextBox.Text) < 10` | Evaluates *true* when numberTextBox.Text is "1", "2", "3", "4", "5", "6", "7", "8", or "9". |
| `!` (not) | Reverses the Boolean expression so that a *true* expression will evaluate *false* and vice versa. | `! foundBool` | Evaluates *true* when the Boolean value is *false*. |

### Compound Boolean Expression Examples

Examples

```
if (maleRadioButton.Checked && int.Parse(ageTextBox.Text) < 21)
{
    minorMaleCountInteger++;
}
if (juniorRadioButton.Checked || seniorRadioButton.Checked)
{
    upperClassmanInteger++;
}
```

The first example requires that both the radio button test and the age test be *true* for the count to be incremented. In the second example, only one of the conditions must be true.

One caution when using compound Boolean expressions: Each side of the logical operator must be a complete expression. For example,

```
countInteger > 10 || < 0
```

is incorrect. Instead, it must be

```
countInteger > 10 || countInteger < 0
```

### Combining Logical Operators

You can create compound Boolean expressions that combine multiple && and || conditions. When you have both an && and an ||, the && is evaluated before the ||. However, you can change the order of evaluation by using parentheses; any expression inside parentheses will be evaluated first.

For example, will the following condition evaluate *true* or *false*? Try it with various values for saleDecimal, discountRadioButton, and stateTextBox.Text.

```
if (saleDecimal > 1000.0m || discountRadioButton.Checked &&
  stateTextBox.Text.ToUpper() != "CA" )
{
    // Code here to calculate the discount.
}
```

| saleDecimal | discountRadioButton.Checked | stateTextBox.Text.ToUpper | Evaluates |
|---|---|---|---|
| 1500.0 | *false* | CA | *true* |
| 1000.0 | *true* | OH | *true* |
| 1000.0 | *true* | CA | *false* |
| 1500.0 | *true* | NY | *true* |
| 1000.0 | *false* | CA | *false* |

## Short-Circuit Operations

When evaluating a compound Boolean expression, sometimes the second expression is never evaluated. If a compound expression has an || and the first expression evaluates *true*, there is no reason to evaluate the second expression. For example, if you have the condition

```
totalInteger < 0 || totalInteger > 10
```

and totalInteger = –1 (negative 1), as soon as the first expression is tested, the entire expression is deemed *true* and the comparison stops, called ***short circuiting*** the operation. Likewise, if you have a compound expression with an && and the first expression evaluates *false*, there is no reason to evaluate the second expression.

```
countInteger >= 0 && countInteger <= 10
```

If countInteger = –1, the first expression is evaluated *false* and the comparison stops.

Most of the time you don't care whether it evaluates both expressions or not. But sometimes the result may not be what you expected. For example, consider the condition

```
amountInteger > 0 && balanceDecimal++ < limitDecimal
```

When amountInteger is less than or equal to 0, the first condition is *false* and there is no need to evaluate the second expression. This may have some surprising results: if you were performing an increment in the second expression, the increment may never occur.

If you need to have both comparisons made, you can avoid short circuiting by using a single comparison operator, rather than the double operator.

<table>
<tr><th>And—Short circuit</th><th>And—Regular<br>(forces comparison to occur)</th><th>Or—Short circuit</th><th>Or—Regular<br>(forces comparison to occur)</th></tr>
<tr><td>&&</td><td>&</td><td>||</td><td>|</td></tr>
</table>

The following condition forces the second condition to be tested, even when the first condition is *false*, and so guarantees that the increment operator is processed.

```
amountInteger > 0 & balanceDecimal++ < limitDecimal
```

## Nested if Statements

In many situations, another `if` statement is one of the statements to be executed when a condition tests *true* or *false*. An `if` statement that contains additional `if` statements is said to be a ***nested*** ***`if`*** statement. The following example shows a nested `if` statement in which the second `if` occurs in the *true* result of the first `if` (Figure 4.5).

```
if (tempInteger > 32)
{
    if (tempInteger > 80)
    {
        commentLabel.Text =  "Hot";
    }
    else
    {
        commentLabel.Text = "Moderate";
    }
}
else
{
    commentLabel.Text = "Freezing";
{
```

To nest `if` statements in the `else` portion, you may use either of the following approaches; however, your code is simpler if you use the second method (using `else if`).

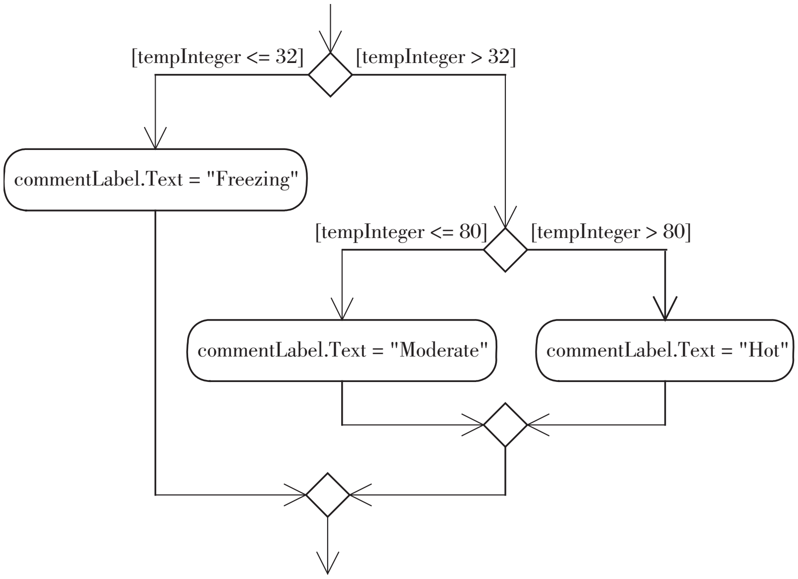

[tempInteger <= 32]
[tempInteger > 32]
commentLabel.Text = "Freezing"
[tempInteger <= 80]
[tempInteger > 80]
commentLabel.Text = "Moderate"
commentLabel.Text = "Hot"

**Figure 4.5**

*Diagramming a nested* `if` *statement.*

```
if (tempInteger <= 32)
{
    commentLabel.Text = "Freezing";
}
else
{
    if (tempInteger > 80)
    {
        commentLabel.Text = "Hot";
    }
    else
    {
        commentLabel.Text = "Moderate";
    }
}
```

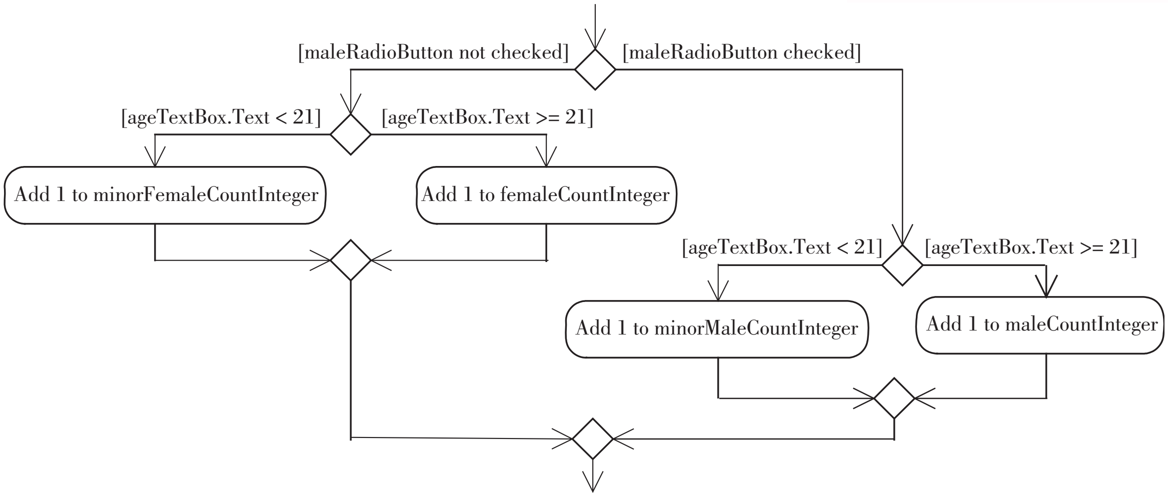
This code has the same logic but uses an `else if`:

```
if (tempInteger <= 32)
{
    commentLabel.Text = "Freezing";
}
else if (tempInteger > 80)
{
    commentLabel.Text = "Hot";
}
else
{
    commentLabel.Text = "Moderate";
}
```

You can nest `ifs` in both the *true* block and the `else` block. In fact, you may continue to nest `ifs` within `ifs` (Figure 4.6). However, projects become very difficult to follow (and may not perform as intended) when `if`s become too deeply nested.

**Figure 4.6**

*A diagram of a nested `if` statement with `if`s nested on both sides of the original `if`.*


[maleRadioButton not checked]
[maleRadioButton checked]
[ageTextBox.Text < 21]
[ageTextBox.Text >= 21]
Add 1 to minorFemaleCountInteger
Add 1 to femaleCountInteger
[ageTextBox.Text < 21]
[ageTextBox.Text >= 21]
Add 1 to minorMaleCountInteger
Add 1 to maleCountInteger

```
if (maleRadioButton.Checked)
{
    if (int.Parse(ageTextBox.Text) < 21)
    {
        minorMaleCountInteger++;
    }
    else
    {
        maleCountInteger++;
    }
}
else
{
    if (int.Parse(ageTextBox.Text) < 21)
    {
        minorFemaleCountInteger++;
    }
    else
    {
        femaleCountInteger++;
    }
}
```

## Coding an else if

When you need to write another `if` in the `else` portion of the original `if` statement, you can choose the spacing. You can place the second (nested) `if` on a line by itself or on the same line as the `else`. You can choose to write the preceding nested `if` statement this way:

```
if (maleRadioButton.Checked)
{
    if (int.Parse(ageTextBox.Text) < 21)
    {
        minorMaleCountInteger++;
    }
    else
    {
        maleCountInteger++;
    }
}
else if (int.Parse(ageTextBox.Text) < 21)
{
    minorFemaleCountInteger++;
}
else
{
    femaleCountInteger++;
}
```

Notice that the indentation changes when you write `else if` on one line, and the logic of the entire nested `if` may not be quite as clear. Most programmers code `else if` on one line when they must test multiple conditions.

```
if (twelveOunceRadioButton.Checked)
{
    itemPriceDecimal = 3m;
}
else if (sixteenOunceRadioButton.Checked)
{
    itemPriceDecimal = 3.5m;
}
else if (twentyOunceRadioButton.Checked)
{
    itemPriceDecimal = 4m;
else
{
    MessageBox.Show("Select the desired size", "Required Entry");
}
```

## Feedback 4.2

Assume that `frogsInteger = 10`, `toadsInteger = 5`, and `polliwogsInteger = 6`. What will be displayed for each of the following statements?

```
1. if (frogsInteger > polliwogsInteger)
   {
       frogsRadioButton.Checked = true;
   }
   else
   {
       polliwogsRadioButton.Checked = true;
   }
```

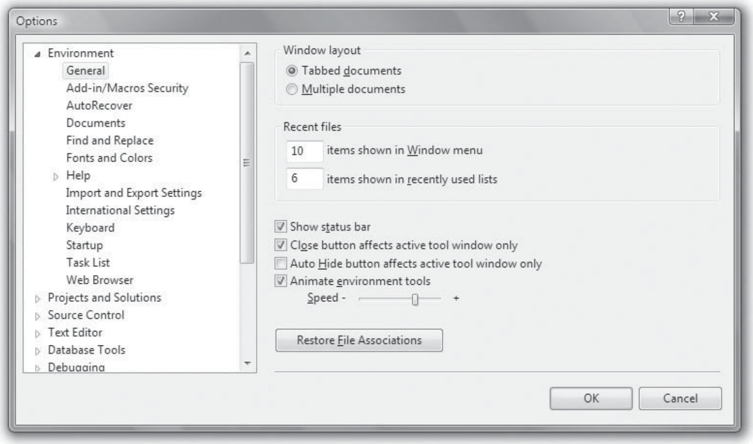
2.
```
if (frogsInteger > toadsInteger + polliwogsInteger)
{
    resultLabel.Text = "It's the frogs";
}
else
{
    resultLabel.Text = "It's the toads and the polliwogs";
}
```
3.
```
if (polliwogsInteger > toadsInteger && frogsInteger != 0
    || toadsInteger == 0)
{
    resultLabel.Text = "It's true";
}
else
{
    resultLabel.Text = "It's false";
}
```
4. Write the statements necessary to compare the numeric values stored in applesTextBox.Text and orangesTextBox.Text. Display in mostTextBox.Text which has more, the apples or the oranges, or a tie.
5. Write the statements that will test the current value of balanceDecimal. When balanceDecimal is greater than zero, the check box for Funds Available, called fundsCheckBox, should be selected, balanceDecimal is set back to zero, and countInteger is incremented by one. When balanceDecimal is zero or less, fundsCheckBox should not be selected (do not change the value of balanceDecimal or increment the counter).

# Using if Statements with Radio Buttons and Check Boxes

In Chapter 2 you used the CheckedChanged event for radio buttons and check boxes to carry out the desired action. Now that you can use `if` statements, you should not take action in the CheckedChanged event handlers for these controls. Instead, use `if` statements to determine which options are selected.

To conform to good programming practice and make your programs consistent with standard Windows applications, place your code in the Click event handler of buttons, such as an *OK* button or an *Apply* button. For example, refer to the Visual Studio *Options* dialog box (Figure 4.7); no action will occur when you click on a radio button or check box. Instead, when you click on the *OK* button, VS checks to see which options are selected.

In an application such as the radio button project in Chapter 2 (refer to Figure 2.23), you could modify the code for the button to include code similar to the following:



Window layout

● Tabbed documents
○ Multiple documents

Recent files

10 items shown in Window menu
6 items shown in recently used lists

☑ Show status bar
☑ Close button affects active tool window only
□ Auto Hide button affects active tool window only
☑ Animate environment tools
Speed - +

**Figure 4.7**

*The Visual Studio* Options *dialog box. When the user clicks* OK*, the program checks the state of all radio buttons and check boxes.*

```
if (beigeRadioButton.Checked)
{
    this.BackColor = Color.Beige;
}
else if (blueRadioButton.Checked)
{
    this.BackColor = Color.Blue;
}
else if (yellowRadioButton.Checked)
{
    this.BackColor = Color.Yellow;
}
else if (grayRadioButton.Checked)
{
    this.BackColor = Color.Gray;
}
```

### Additional Examples

```
if (fastShipCheckBox.Checked)
  totalDecimal += fastShipRateDecimal;

if (giftWrapCheckBox.Checked)
  totalDecimal += wrapAmountDecimal;
```

## A “Simple Sample”

Test your understanding of the use of the `if` statement by coding some short examples.

### Test the Value of a Check Box

Create a small project that contains a check box, a label, and a button. Name the button testButton, the check box testCheckBox, and the label messageLabel. In the Click event handler for testButton, check the value of the check box. If the check box is currently checked, display “Check box is checked” in messageLabel.

```
private void testButton_Click(object sender, EventArgs e)
{
    // Test the value of the check box.
    if (testCheckBox.Checked)
    {
        messageLabel.Text = "Check box is checked.";
    }
}
```

Test your project. When it works, add an `else` to the code that displays “Check box is not checked”.

### Test the State of Radio Buttons

Remove the check box from the previous project and replace it with two radio buttons, named freshmanRadioButton and sophomoreRadioButton and labeled “< 30 units” and “>= 30 units”. Now change the `if` statement to display “Freshman” or “Sophomore” in the label.

```
if (freshmanRadioButton.Checked)
{
    messageLabel.Text = "Freshman";
}
else
{
    messageLabel.Text = "Sophomore";
}
```

Can you modify the sample to work for Freshman, Sophomore, Junior, and Senior? In the sections that follow, you will see code for testing multiple radio buttons and check boxes.

## Checking the State of a Radio Button Group

Nested `if` statements work very well for determining which button of a radio button group is selected. Recall that in any group of radio buttons, only one button can be selected. Assume that your form has a group of radio buttons for Freshman, Sophomore, Junior, and Senior. In a calculation method, you want to add 1 to one of four counter variables, depending on which radio button is selected:

```
if (freshmanRadioButton.Checked)
{
    freshmanCountInteger++;
}
else if (sophomoreRadioButton.Checked)
{
    sophomoreCountInteger++;
}
```

```
else if (juniorRadioButton.Checked)
{
    juniorCountInteger++;
}
else if (seniorRadioButton.Checked)
{
    seniorCountInteger++;
}
```

Note that, in most situations, the final test is unnecessary. You should be able to just code an `else` and add to seniorRadioButton if the first three expressions are *false*. You might prefer to code the expression to make the statement more clear, or if no radio button is set initially, or if the program sets all radio buttons to false.

### Clearing and Testing Radio Buttons

You can clear all radio buttons in a group by setting the Checked property of each to *false*. However, this leads to a lot of programming if you have a long list of radio buttons. *Remember*: When you set one button to *true*, all of the others are automatically set to *false*. You can use this fact to easily clear a set of radio buttons. Add an extra button to the group and set its Visible property to *false*. When you want to clear the visible buttons, set the Checked property of the invisible radio button to *true*.

```
noColorRadioButton.Checked = true;
```

If you are using this technique, you should set the initial value of the invisible radio button to Checked = *true* in the Form Designer.

**Testing whether the User Has Selected an Option** When you use an invisible radio button, you can verify in code that the user has made a selection by testing the Checked property of the button:

```
if (noColorRadioButton.Checked)
{
    MessageBox.Show("Please select a color.", "Required Entry");
}
else
{
    // A selection was made; take any appropriate action.
}
```

## Checking the State of Multiple Check Boxes

Although nested `if` statements work very well for groups of radio buttons, the same is not true for a series of check boxes. Recall that if you have a series of check boxes, any number of the boxes may be selected. In this situation, assume that you have check boxes for Discount, Taxable, and Delivery. You will need separate `if` statements for each condition.

```
if (discountCheckBox.Checked)
{
    // Calculate the discount.
}
if (taxableCheckBox.Checked)
{
    // Calculate the tax.
}
if (deliveryCheckBox.Checked)
{
    // Calculate the delivery charges.
}
```

# Enhancing Message Boxes

In Chapter 3 you learned to display a message box to the user. Now it's time to add such features as controlling the format of the message, displaying multiple buttons, checking which button the user clicks, and performing alternate actions depending on the user's selection.

## Displaying the Message String

The message string you display in a message box may be a string literal enclosed in quotes or a string variable. You also may want to concatenate several items, for example, combining a literal with a value from a variable. It's usually a good idea to create a variable for the message and format the message before calling the `Show` method; if nothing else, it makes your code easier to read and follow.

### Combining Values into a Message String

You can concatenate a literal such as "Total Sales: " with the value from a variable. You may need to include an extra space inside the literal to make sure that the value is separated from the literal.

```
string messageString = "Total Sales: " + totalDecimalSales.ToString("C");
MessageBox.Show(messageString, "Sales Summary", MessageBoxButtons.OK);
```

### Creating Multiple Lines of Output

If your message is too long for one line, the message wraps to a second line. But if you would like to control the line length and position of the split, you can insert a **NewLine (`\n`) character** into the string message. You can concatenate this constant into a message string to set up multiple lines.

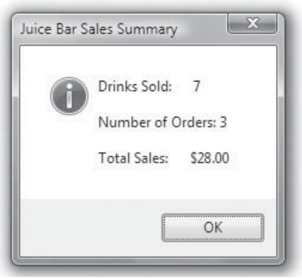
In this example, a second line is added to the MessageBox from the previous example.

**TIP**

Specify only the message for a "quick and dirty" message box for debugging purposes. It will display an *OK* button and an empty title bar: `MessageBox.Show("I'm here.");`. ■

```
string formattedTotalString = totalSalesDecimal.ToString("N");
string formattedAvgString = averageSaleDecimal.ToString("N");
string messageString = "Total Sales: " + formattedTotalString + "\n"
  + "Average Sale: " + formattedAvgString;
MessageBox.Show(messageString, "Sales Summary", MessageBoxButtons.OK);
```

You can combine multiple NewLine constants to achieve double spacing and create multiple message lines (Figure 4.8).



Juice Bar Sales Summary

Drinks Sold: 7

Number of Orders: 3

Total Sales: $28.00

**Figure 4.8**

*A message box with multiple lines of output, created by concatenating two NewLine characters at the end of each line.*

```
// Concatenate the text for the message.
string summaryString = "Drinks Sold: " + drinksInteger.ToString() + "\n\n" +
                       "Number of Orders: " + ordersInteger.ToString() + "\n\n" +
                       "Total Sales: " + totalSalesDecimal.ToString("C");

// Display the message box.
MessageBox.Show(summaryString, "Juice Bar Sales Summary", MessageBoxButtons.OK,
  MessageBoxIcon.Information);
```

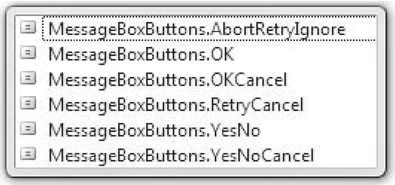
### Using the Character Escape Sequences

You can use several other constants from the character escape sequence (\) list, in addition to the `\n` constant.

| Escape sequence | Description |
|---|---|
| `\'` | Includes a single quote in a character literal. |
| `\"` | Includes a double quote in a string literal. |
| `\\` | Includes a backslash in a string literal. |
| `\n` | New line. |
| `\r` | Carriage return. |
| `\b` | Backspace character. |
| `\f` | Formfeed character (not useful in Microsoft Windows). |
| `\t` | Horizontal tab. |

## Displaying Multiple Buttons

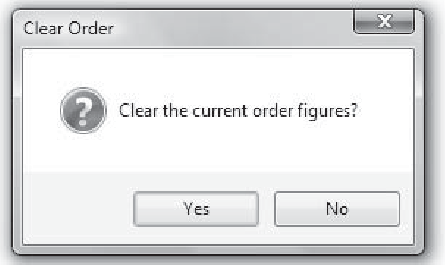
You can choose the buttons to display in the message box using the MessageBoxButtons constants (Figure 4.9). Figure 4.10 shows a MessageBox with two buttons using the `MessageBoxButtons.YesNo` constant. The `Show` method



MessageBoxButtons.AbortRetryIgnore
MessageBoxButtons.OK
MessageBoxButtons.OKCancel
MessageBoxButtons.RetryCancel
MessageBoxButtons.YesNo
MessageBoxButtons.YesNoCancel

**Figure 4.9**

*Choose the button(s) to display from the MessageBoxButtons constants.*



Clear Order

Clear the current order figures?

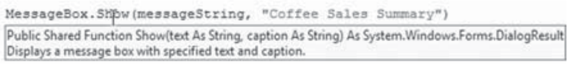
**Figure 4.10**

*Display* Yes *and* No *buttons on a message box using* `MessageBoxButtons.YesNo`.

returns a **DialogResult object** that you can check to see which button the user clicked.

## Determining the Return Type of a Method

How do you know that the `Show` method returns an object of the DialogResult class? An easy way is to point to the `Show` keyword and pause; the popup displays the type of the return value (Figure 4.11).



```
MessageBox.Show(messageString, "Coffee Sales Summary")
```

Public Shared Function Show(text As String, caption As String) As System.Windows.Forms.DialogResult
Displays a message box with specified text and caption.

**Figure 4.11**

*Pause the mouse pointer over the* `Show` *keyword and IntelliSense pops up with an argument list and the method's return type.*

## Declaring an Object Variable for the Method Return

To capture the information about the outcome of the `Show` method, you must declare a variable that can hold an instance of the DialogResult type.

```
DialogResult whichButtonDialogResult;
```

Then you assign the return value of the `Show` method to the new variable.

```
string messageString = "Clear the current order figures?";
whichButtonDialogResult = MessageBox.Show(messageString, "Clear Order",
  MessageBoxButtons.YesNo, MessageBoxIcon.Question);
```

The next step is to check the value of the return, comparing to the DialogResult constants, such as `Yes`, `No`, `OK`, `Retry`, `Abort`, and `Cancel`.

```
if (whichButtonDialogResult == DialogResult.Yes)
{
    // Code to clear the order.
}
```

### Specifying a Default Button and Options

Two additional signatures for the `MessageBox.Show` method are as follows:

General Form

```
MessageBox.Show(TextMessage, TitlebarText, MessageBoxButtons, MessageBoxIcons,
  MessageBoxDefaultButton);
MessageBox.Show(TextMessage, TitlebarText, MessageBoxButtons, MessageBoxIcons,
  MessageBoxDefaultButton, MessageBoxOptions);
```

When you display multiple buttons, you may want one of the buttons to be the default (the Accept button). For example, to make the second button (the *No* button) the default, use this statement:

```
string messageString = "Clear the current order figures?";
whichButtonDialogResult = MessageBox.Show(messageString, "Clear Order",
  MessageBoxButtons.YesNo, MessageBoxIcon.Question,
  MessageBoxDefaultButton.Button2);
```

You can right-align the text in the message box by setting the MessageBoxOptions argument:

```
string messageString = "Clear the current order figures?";
whichButtonDialogResult = MessageBox.Show(messageString, "Clear Order",
  MessageBoxButtons.YesNo, MessageBoxIcon.Question,
  MessageBoxDefaultButton.Button2, MessageBoxOptions.RightAlign);
```

## Input Validation

Careful programmers check the values entered into text boxes before beginning the calculations. Validation is a form of self-protection; it is better to reject bad data than to spend hours (and sometimes days) trying to find an error, only to discover that the problem was caused by a "user error." Finding and correcting the error early can often keep the program from producing erroneous results or halting with a run-time error.

Checking to verify that appropriate values have been provided as input is called ***validation***. The validation may include making sure that the input is numeric, checking for specific values, checking a range of values, or making sure that required items are entered.

In Chapter 3 you learned to use `try/catch` blocks to trap for nonnumeric values. This chapter presents some additional validation techniques using `if` statements.

*Note*: Chapter 14 covers some advanced validation techniques using the Validating event and error providers.

## Checking for a Range of Values

Data validation may include checking the reasonableness of a value. Assume you are using a text box to input the number of hours worked in a day. Even with overtime, the company does not allow more than 10 work hours in a single day. You could check the input for reasonableness with this code:

```
if (int.Parse(hoursTextBox.Text) > 10)
{
    MessageBox.Show("Too many hours.", "Invalid Data", MessageBoxButtons.OK);
}
```

## Checking for a Required Field

Sometimes you need to be certain that a value has been entered into a text box before proceeding. You can compare a text box value to an empty string literal.

```
if (nameTextBox.Text != "")
{
    // Good data -- Perform some action.
}
else
{
    MessageBox.Show("Required entry.", "Invalid Data", MessageBoxButtons.OK);
}
```

Many programmers prefer to reverse the test in an `if` statement, so that invalid entries are caught in the *true* portion of the test. This practice can be easier to read and understand. The preceding validation could be written as follows, which does not require the use of *not* (`!`).

```
if (nameTextBox.Text == "")
{
    MessageBox.Show("Required entry.", "Invalid Data", MessageBoxButtons.OK);
}
else
{
    // Good data -- Perform some action.
}
```

By checking separately for blank or nonnumeric data, you can display a better message to the user. Make sure to check for blanks first, since a blank field will throw an exception with a parsing method. For example, if you reverse the order of the if and try blocks in the following example, blanks in quantityTextBox will always trigger the nonnumeric message in the catch block.

```
if (quantityTextBox.Text != "") // Not blank.
{
    try
        quantityDecimal = decimal.Parse(quantityTextBox.Text);
    catch             // Nonnumeric data.
    {
        messageString = "Nonnumeric data entered for quantity.";
        MessageBox.Show(messageString, "Data Entry Error");
    }
}
else                 // Missing data.
{
    messageString = "The quantity is required.";
    MessageBox.Show(messageString, "Data entry error");
}
```

## Performing Multiple Validations

When you need to validate several input fields, how many message boxes do you want to display for the user? Assume that the user has neglected to fill five text boxes or make a required selection and clicked on *Calculate*. You can avoid displaying multiple message boxes in a row by using a nested if statement. This way you check the second value only if the first one passes, and you can exit the processing if a problem is found with a single field.

```
if (nameTextBox.Text != "")
{
    try
    {
        unitsDecimal = decimal.Parse(unitsTextBox.Text);
        if (freshmanRadioButton.Checked || sophomoreRadioButton.Checked
          || juniorRadioButton.Checked || seniorRadioButton.Checked)
        {
          // Data valid -- Do calculations or processing here.
        }
        else
        {
            MessageBox.Show("Please select a grade level.",
              "Data Entry Error", MessageBoxButtons.OK);
        }
    }
    catch(FormatException)
    {
        MessageBox.Show("Enter number of units.", "Data Entry Error",
          MessageBoxButtons.OK);
        unitsTextBox.Focus();
    }
}
```

```
else
{
    MessageBox.Show ("Please enter a name", "Data Entry Error",
      MessageBoxButtons.OK);
    nameTextBox.Focus();
}
```

# The switch Statement

Earlier you used the `if` statement for testing conditions and making decisions. Whenever you want to test a single variable for multiple values, the **`switch` statement** provides a flexible and powerful solution. Any decisions that you can code with a `switch` statement also can be coded with nested `if` statements, but usually the `switch` statement is simpler and more clear.

### The switch Statement—General Form

General Form

```
switch (expression)
{
    case testValue1:
        [statement(s);
        break;]
    [case testValue2:
        statement(s);
        break;]
    .
    .
    .
    [default:]
        statement(s);
        break;]
}
```

The expression in a `switch` statement is usually a variable or property that you wish to test. The test values are the values that you want to match; they may be numeric or string constants or variables, and must match the data type of the expression you are testing.

There is no limit to the number of case blocks that you can include, and no limit to the number of statements that can follow a `case` statement. After all of the statements for a `case` statement, you must place a `break` statement. The `break` causes the `switch` to terminate.

### The switch Statement—Examples

Examples

```
switch (listIndexInteger)
{
    case 0:
        // Code to handle selection of zero.
        break;
    case 1:
        // Code to handle selection of one.
        break;
    default:
        // Code to handle no selection made or no match.
        break;
}
switch (scoreInteger/10)
{
    case 10:
    case 9:
        messageLabel1.Text = "Excellent Score";
        messageLabel2.Text = "Give yourself a pat on the back.";
        break;
    case 8:
        messageLabel1.Text = "Very Good";
        messageLabel2.Text = "You should be proud.";
        break;
    case 7:
        messageLabel1.Text = "Satisfactory Score";
        messageLabel2.Text = "You should have a nice warm feeling.";
        break;
    default:
        messageLabel1.Text = "Your score shows room for improvement.";
        messageLabel2.Text = "";
        break;
}
```

Notice in the second example above that `case 10:` has no code. In this case, the code beneath `Case 9:` executes for both 10 and 9.

When you want to test for a string value, you must include quotation marks around the literals.

### Example

```
switch (teamNameTextBox.Text)
{
    case "Tigers":
        // Code for Tigers.
        break;
    case "Leopards":
        // Code for Leopards.
        break;
    case "Cougars":
    case "Panthers":
        // Code for Cougars and Panthers.
        break;
    default:
        // Code for any nonmatch.
        break;
}
```

Note that, in the previous example, the capitalization must also match exactly. A better solution would be

```
switch (teamNameTextBox.Text.ToUpper())
{
    case "TIGERS":
        // Code for Tigers.
        break;
    case "LEOPARDS":
        // Code for Leopards.
        break;
    case "COUGARS":
    case "PANTHERS":
        // Code for Cougars and Panthers.
        break;
    default:
        // Code for any nonmatch.
        break;
}
```

Although the `default` clause is optional, generally you will want to include it in `switch` statements. The statements you code beneath `default` execute only if none of the other `case` expressions is matched. This clause provides checking for any invalid or unforeseen values of the expression being tested. If the `default` clause is omitted and none of the `case` conditions is *true*, the program continues execution at the statement following the closing brace for the `switch` block.

The final `break` statement in the last `case` or `default` statement seems unnecessary, but the compiler generates an error message if you leave it out.

If more than one `case` value is matched by the expression, only the statements in the *first* matched `case` clause execute.

## Feedback 4.3

1. Convert the following `if` statement to a `switch` statement:

```
if (codeString == "A")
{
    outputLabel.Text = "Excellent";
}
else if (codeString == "B")
{
    outputLabel.Text = "Good";
}
else if (codeString == "C" || codeString == "D")
{
    outputLabel.Text = "Satisfactory";
}
else
{
    outputLabel.Text = "Not Satisfactory";
}
```

2. Rewrite the `switch` statement from question 1 to handle upper- or lowercase values for codeString.
3. Write the `switch` statement to convert this `if` statement to a `switch` statement:

```
if (countInteger == 0)
{
    MessageBox.Show("Invalid value");
}
else
{
    averageDecimal = sumDecimal / countInteger;
    MessageBox.Show("The average is: " + averageDecimal.ToString());
}
```

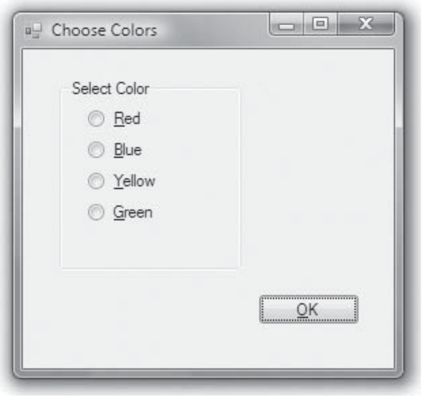
## Sharing an Event Handler

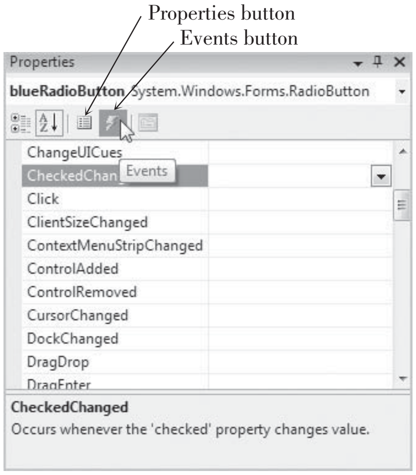
A very handy feature of C# is the ability to share an event-handling method for the events of multiple controls. For example, assume that you have a group of four radio buttons to allow the user to choose a color (Figure 4.12). Each of the radio buttons must have its own name and will ordinarily have its own event-handling method. You can code the event-handling method for one of the radio buttons and then connect the other radio buttons to the same method.

To share an event-handling method, first double-click on one of the radio buttons; the editor creates the event handler for that button. Then select another radio button and click on the *Events* button in the Properties window. You will see a list of the available events for the selected control (Figure 4.13).

**Figure 4.12**

*The four radio buttons allow the user to choose the color.*





CheckedChanged
Occurs whenever the 'checked' property changes value.

**Figure 4.13**

*Select the Events button in the Properties window to display the list of events for the selected control. You can set the event handler to an existing method.*

**TIP**

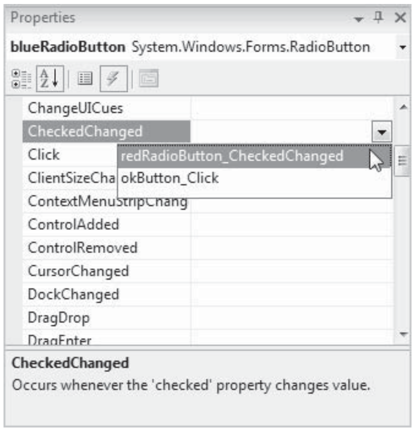
When you share an event handler, rename the method to more clearly reflect its purpose, such as radioButtons_CheckedChanged instead of blueRadioButton_CheckedChanged. Right-click on the method name in the Code Editor and select `Refactor / Rename` from the context menu to rename the method everywhere it occurs. ■

You can set the handler for the CheckedChanged event for a control to an existing method by selecting the method from the drop-down list (Figure 4.14).

After you have selected a single event-handling method for multiple controls, this method will execute when the user selects *any* of the radio buttons.

A good, professional technique is to set up a class-level variable to hold the selection that the user makes. Then, in the *OK* button's event-handling method, you can take action based on which of the buttons was selected.

The key to using the shared event handler is the `sender` argument that is passed to the `CheckedChanged` method. The sender is defined as a generic



CheckedChanged
Occurs whenever the 'checked' property changes value.

**Figure 4.14**

*Select the* `redRadioButton_CheckedChanged` *method to handle the CheckedChanged event for blueRadioButton.*

object. To use its Name property, you must cast sender to a RadioButton type. Then you can use its Name property in a switch statement to determine which radio button the user selected.

```
// Declare class-level variable.
Color selectedColor;
```

You can declare a class-level variable as a Color data type, assign the chosen color in the shared event handler, and then apply the color in the *OK* button's click event handler. In this example, we used *Refactor / Rename* to rename the shared event handler from redRadioButton_CheckedChanged to radioButtons_CheckedChanged, to make the name more generic.

```
public partial class ChangeColorForm : Form
{
    // Declare class-level variable.
    Color selectedColor;

    public ChangeColorForm()
    {
        InitializeComponent();
    }

    private void radioButtons_CheckedChanged(object sender, EventArgs e)
    {
        // Set the selected color to match the radio button.
        // Handles all four radio buttons.

        // Cast the sender argument to a RadioButton data type.
        RadioButton selectedRadioButton = (RadioButton)sender;
        switch (selectedRadioButton.Name)
        {
            case "redRadioButton":
                selectedColor = Color.Red;
                break;
            case "blueRadioButton":
                selectedColor = Color.Blue;
                break;
            case "yellowRadioButton":
                selectedColor = Color.Yellow;
                break;
            case "greenRadioButton":
                selectedColor = Color.Green;
                break;
        }
    }

    private void okButton_Click(object sender, EventArgs e)
    {
        // Change the color based on the selected radio button.

        this.BackColor = selectedColor;
    }
}
```

## Calling Event Handlers

If you wish to perform a set of instructions in more than one location, you should never duplicate the code. Write the instructions once, in an event-handling method, and "call" the method from another method. When you **call** a method, the entire method is executed and then execution returns to the statement following the call.

### The Call Statement—General Form

General Form

```
MethodName();
```

You must include the parentheses; if the method that you are calling requires arguments, then place the arguments within the parentheses; otherwise, leave them empty. Note that all method calls in this chapter *do* require arguments. Chapter 5 covers more options including writing additional methods, with and without arguments.

### The Call Statement—Example

Example

```
clearButton_Click(sender, e);
```

Notice the arguments for the `call` statement. You are passing the same two arguments that were passed to the calling method. If you examine any of the editor-generated event-handling method headers, you can see that every event handler requires these two arguments, which can be used to track the object that generated the event.

```
private void newOrderButton_Click(object sender, System.EventArgs e)
{
    // . . .
    // Call the clearButton_Click event-handling method.
    clearButton_Click(sender, e);
}
```

In the programming example that follows, you will accumulate individual items for one customer. When that customer's order is complete, you need to clear the entire order and begin an order for the next customer. Refer to the form in Figure 4.15; notice the two buttons: *Clear for Next Item* and *New Order*. The button for next item clears the text boxes on the screen. The button for a new order must clear the screen text boxes and clear the subtotal fields. Rather than repeat the instructions to clear the individual screen text boxes, we can call the event handler for clearButton_Click from the newOrderButton_Click method.

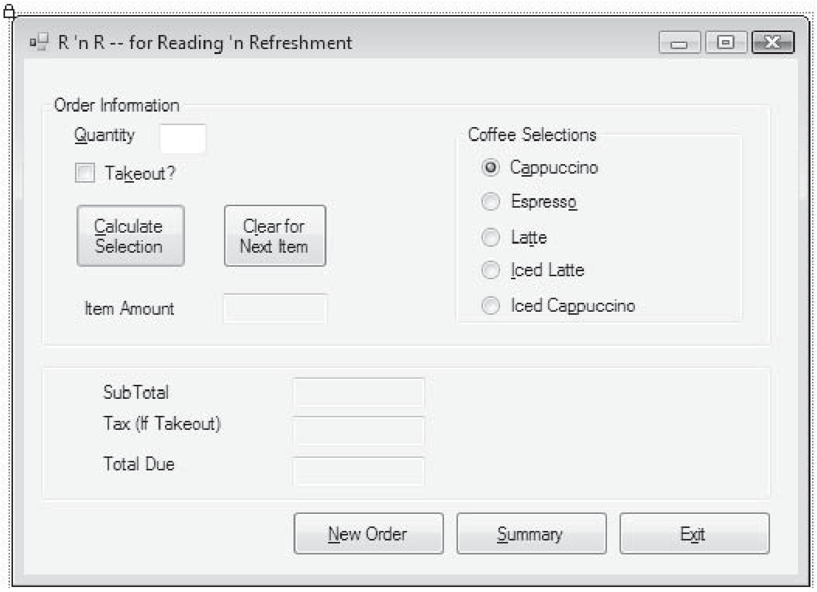


**Figure 4.15**

*A form with buttons that perform overlapping functions. The* New Order *button must include the same tasks as* Clear for Next Item.

```
private void newOrderButton_Click(object sender, System.EventArgs e)
{
    // Clear the current order and add to totals.

    DialogResult    responseDialogResult;
    string          messageString;

    // Confirm clear of current order.
    messageString = "Clear the current order figures?";
    responseDialogResult = MessageBox.Show(messageString, "Clear Order",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question,
        MessageBoxDefaultButton.Button2);

    if (responseDialogResult == DialogResult.Yes)  // User said Yes.
    {
        clearButton_Click(sender, e); // Clear the current order fields.
        // Continue with statements to clear the subtotals.
    }
}
```

In the newOrderButton_Click handler, all the instructions in clearButton_Click are executed. Then execution returns to the next statement following the call.

## Your Hands-On Programming Example

Create a project for the juice bar at Look Sharp Fitness Center. The application must calculate the amount due for individual orders and maintain accumulated totals for a summary. Have option buttons for the size (12, 16, and 20 ounce) and options for the drink selections. Two juices are available and three smoothie flavors. Have a check box for the three extra additive choices.

The prices for the drinks are 3.00, 3.50, and 4.00, depending on size. Each additive is an additional 50 cents. Display the price of the drink as the options are selected. Only add it to the order when the *Add to Order* button is selected.

The buttons are *Add to Order* for each drink selection, *Order Complete* to display the amount due for the order, *Summary Report* to display the summary information, and *Exit*.

The summary information should contain the number of drinks served, the number of orders, and the total dollar amount. Display the summary data in a message box.

When the user clicks the *Add to Order* button, the price of the drink should be added to the order totals and the screen fields should be reset to their default values, which are as follows: No selection for size or extras, *Fruit* selected for Juices and *1* for quantity, and the item price text box should be empty.

For the Size group of radio buttons, include an extra invisible radio button with its Checked property set to *true* initially. You can test the invisible radio button to make sure that the user has selected a size. Set the radio buttons to *true* when you need to clear the others.

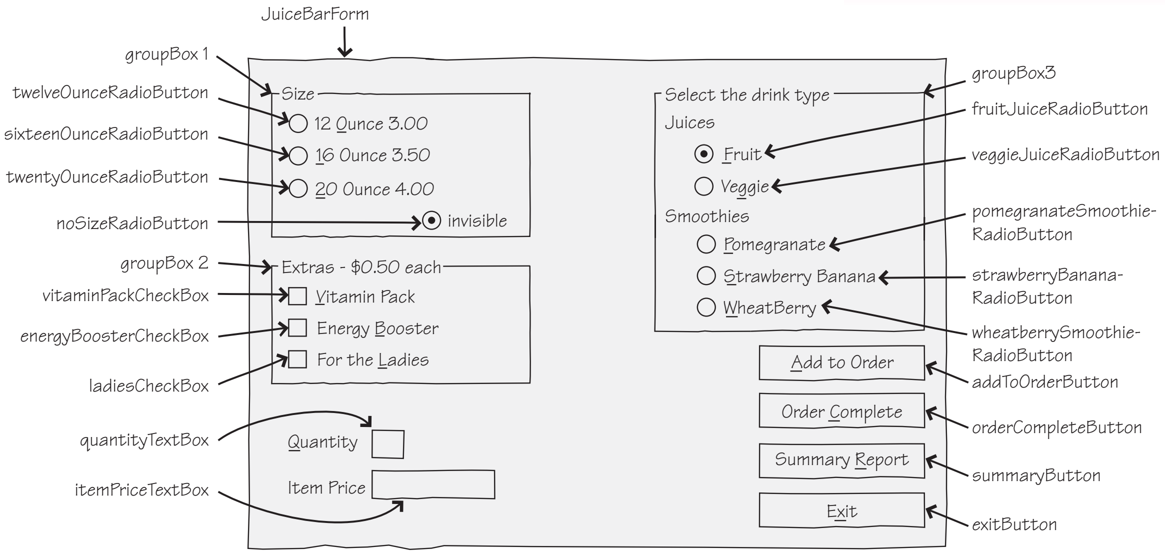
When the user clicks the *Order Complete* button, make sure that the last drink order was added into the total. You will know that an unsaved order is on the screen if the *Item Price* text box has a value (it is cleared when the order is calculated). If the user wants to add the last drink order into the total, call the event handler for the *Add to Order* button, rather than write the code a second time.

## Planning the Project

Sketch a form (Figure 4.16), which your users sign as meeting their needs.

**Figure 4.16**

*The planning sketch of the form for the hands-on programming exercise example.*



Plan the Objects and Properties. Plan the property settings for the form and each of the controls.

| Object | Property | Setting |
|---|---|---|
| JuiceBarForm | Name<br>Text<br>AcceptButton<br>CancelButton | JuiceBarForm<br>Juice Bar Orders<br>addToOrderButton<br>exitButton |
| groupBox1 | Text | Size |
| twelveOunceRadioButton | Name<br>Text | twelveOunceRadioButton<br>12 &Ounce |
| sixteenOunceRadioButton | Name<br>Text | sixteenOunceRadioButton<br>&16 Ounce |
| twentyOunceRadioButton | Name<br>Text | twentyOunceRadioButton<br>&20 Ounce |
| noSizeRadioButton | Name<br>Checked<br>Visible | noSizeRadioButton<br>true<br>false |
| groupBox2 | Text | Extras - $0.50 each |
| vitaminPackCheckBox | Name<br>Text | vitaminPackCheckBox<br>&Vitamin Pack |
| energyBoosterCheckBox | Name<br>Text | energyBoosterCheckBox<br>Energy &Booster |
| ladiesCheckBox | Name<br>Text | ladiesCheckBox<br>For the &Ladies |
| groupBox3 | Text | Select the drink type |
| label1 | Text | Juices |
| fruitJuiceRadioButton | Name<br>Text<br>Checked | fruitJuiceRadioButton<br>&Fruit<br>true |
| veggieJuiceRadioButton | Name<br>Text | veggieJuiceRadioButton<br>Ve&ggie |
| label2 | Text | Smoothies |
| pomegranateSmoothieRadioButton | Name<br>Text | pomegranateSmoothieRadioButton<br>&Pomegranate |
| strawberryBananaRadioButton | Name<br>Text | strawberryBananaRadioButton<br>&Strawberry Banana |
| wheatberrySmoothieRadioButton | Name<br>Text | wheatberrySmoothieRadioButton<br>&WheatBerry |
| label3 | Text | &Quantity |

| Object | Property | Setting |
|---|---|---|
| quantityTextBox | Name<br>Text | quantityTextBox<br>(blank) |
| label4 | Text | Item Price |
| itemPriceTextBox | Name<br>Text<br>ReadOnly<br>TabStop | itemPriceTextBox<br>(blank)<br>true<br>false |
| addToOrderButton | Name<br>Text | addToOrderButton<br>&Add to Order |
| orderCompleteButton | Name<br>Text<br>Enabled | orderCompleteButton<br>Order &Complete<br>false |
| summaryButton | Name<br>Text<br>Enabled | summaryButton<br>Summary &Report<br>false |
| exitButton | Name<br>Text | exitButton<br>E&xit |

Plan the Event Handlers. You need to plan the actions for five event handlers for the buttons.

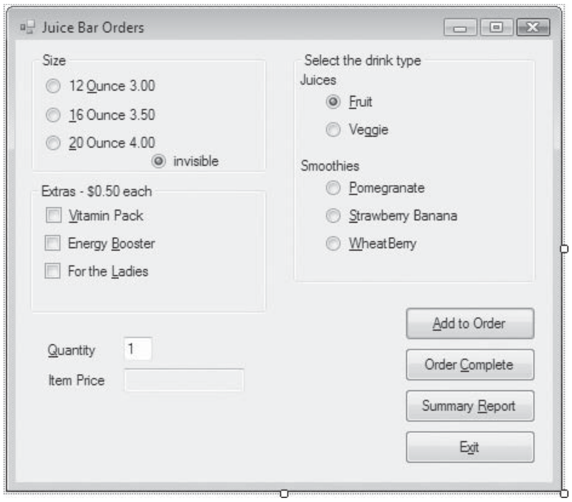
| Object | Method | Action |
|---|---|---|
| addToOrderButton | Click | Check if size is selected.<br>Validate for blank or nonnumeric amount.<br>Multiply price by quantity.<br>Add to the number of drinks.<br>Enable the Order Complete button.<br>Reset controls to default values |
| orderCompleteButton | Click | If last item not cleared from screen<br>  Ask user whether to add it.<br>  If yes<br>    Call addToOrderButton_Click.<br>Display the price of the order.<br>Add to the number of orders and totalSales.<br>Enable the Summary and Order Complete buttons.<br>Clear the order amount |
| summaryButton | Click | Display the summary totals in a message box. |
| exitButton | Click | Terminate the project. |
| Size radio buttons | CheckedChanged | Clear the Extras variable.<br>Find price of selected size. |

Write the Project Follow the sketch in Figure 4.16 to create the form. Figure 4.17 shows the completed form.

- Set the properties of each object as you have planned.
- Write the code. Working from the pseudocode, write each event-handling method.
- When you complete the code, use a variety of data to thoroughly test the project.

**Figure 4.17**

*The form for the hands-on programming exercise.*



## The Project Coding Solution

```
/*
 * Program Name:    Ch04HandsOn
 * Programmer:      Bradley/Millspaugh
 * Date:            June 2009
 *
 * Description:     This project calculates the amount due
 *                  based on the customer selection
 *                  and accumulates summary data for the day.
 *
 *
 */

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

```
namespace Ch04HandsOn
{
    public partial class JuiceBarForm : Form
    {
        // Declare class variables.
        private decimal itemPriceDecimal,
                        totalOrderDecimal,
                        totalSalesDecimal;
        private int     drinksInteger,
                        ordersInteger;

        public JuiceBarForm()
        {
            InitializeComponent();
        }
        private void addToOrderButton_Click(object sender, EventArgs e)
        {
            // Add the current item price and quantity to the order.

            if (noSizeRadioButton.Checked)
            {
                MessageBox.Show("You must select a drink size.",
                  "Missing required entry");
            }
            else
            {
                try
                {
                    int quantityInteger = int.Parse(quantityTextBox.Text);
                    if (quantityInteger != 0)
                    {
                        drinksInteger += quantityInteger;
                        totalOrderDecimal += itemPriceDecimal * quantityInteger;
                        orderCompleteButton.Enabled = true;

                        // Reset defaults for next item.
                        noSizeRadioButton.Checked = true;
                        fruitJuiceRadioButton.Checked = true;
                        vitaminPackCheckBox.Checked = false;
                        energyBoosterCheckBox.Checked = false;
                        ladiesCheckBox.Checked = false;
                        itemPriceTextBox.Clear();
                        quantityTextBox.Text = "1";
                    }
                    else
                    {
                        MessageBox.Show("Please enter a quantity.",
                          "Missing Required Entry");
                    }
                }
                catch (FormatException)
                {
                    MessageBox.Show("Invalid quantity.", "Data Entry Error");
                    quantityTextBox.Focus();
                    quantityTextBox.SelectAll();
                }
            }
        }
```

```
private void orderCompleteButton_Click(object sender, EventArgs e)
{
    // Order is complete, add to summary and clear order.
    // Check if the last item was added to the total.
    if (itemPriceTextBox.Text != "")
    {
        DialogResult responseDialogResult;
        string messageString = "Current Item not recorded. Add to order?";
        responseDialogResult = MessageBox.Show(messageString,
            "Verify Last Drink Purchase",
            MessageBoxButtons.YesNo,
            MessageBoxIcon.Question);
        if (responseDialogResult == DialogResult.Yes)
        {
            addToOrderButton_Click(sender, e);
        }
    }

    // Display amount due.
    string dueString = "Amount Due " + totalOrderDecimal.ToString("C");
    MessageBox.Show(dueString, "Order Complete");

    // Add to summary totals.
    ordersInteger++;
    totalSalesDecimal += totalOrderDecimal;

    // Reset buttons and total for new order.
    summaryButton.Enabled = true;
    orderCompleteButton.Enabled = false;
    totalOrderDecimal = 0m;
}

private void summaryButton_Click(object sender, EventArgs e)
{
    // Display the summary information in a message box.

    string summaryString = "Drinks Sold:      "
            + drinksInteger.ToString()
            + "\n\n" + "Number of Orders: "
            + ordersInteger.ToString()
            + "\n\n" + "Total Sales:      "
            + totalSalesDecimal.ToString("C");
    MessageBox.Show(summaryString, "Juice Bar Sales Summary",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}

private void exitButton_Click(object sender, EventArgs e)
{
    // End the application.

    this.Close();
}

private void twelveOunceRadioButton_CheckedChanged(object sender,
    EventArgs e)
{
    // Calculate and display the price for the selected item.
    // Handles all check boxes and radio buttons.
    int extrasInteger = 0;
```

```
            if (twelveOunceRadioButton.Checked)
            {
                itemPriceDecimal = 3m;
            }
            else if (sixteenOunceRadioButton.Checked)
            {
                itemPriceDecimal = 3.5m;
            }
            else if (twentyOunceRadioButton.Checked)
            {
                itemPriceDecimal = 4m;
            }
            extrasInteger = 0;
            if (vitaminPackCheckBox.Checked)
            {
                extrasInteger++;
            }
            if (energyBoosterCheckBox.Checked)
            {
                extrasInteger++;
            }
            if (ladiesCheckBox.Checked)
            {
                extrasInteger++;
            }
            itemPriceDecimal += extrasInteger * .5m; // 50 cents for each extra.
            itemPriceTextBox.Text = itemPriceDecimal.ToString("C");
        }
    }
}
```

## Debugging C# Projects

One of the advantages of programming in the Visual Studio environment is the availability of debugging tools. You can use these tools to help find and eliminate logic and run-time errors. The debugging tools also can help you to follow the logic of existing projects to better understand how they work.

Sometimes it's helpful to know the result of a Boolean expression, the value of a variable or property, or the sequence of execution of your program. You can follow program logic in Debug mode by single-stepping through code; you also can get information about execution without breaking the program run, using the `WriteLine` method of the Console class.

In the following sections, you will learn to use many of the debugging tools on the Debug toolbar (Figure 4.18) and the *Debug* menu (Figure 4.19). Note that the Debug toolbar appears automatically when you choose the *Start* command; you also can make the toolbar display full-time by right-clicking on any toolbar and choosing *Debug* from the popup menu.

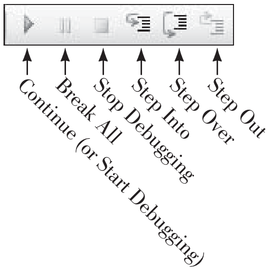


**Figure 4.18**

*The Debug toolbar with its tools for debugging programs.*

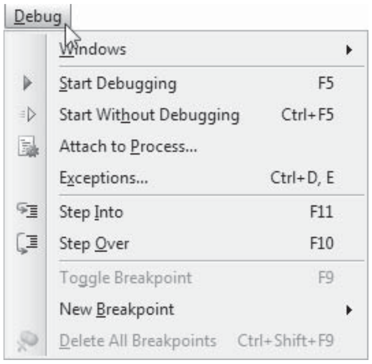


**Figure 4.19**

*The debugging options on the Debug menu.*

## Writing to the Output Window

You can place a **Console.WriteLine method** in your code. In the argument, you can specify a message to write or an object that you want tracked.

### The Console.WriteLine Method—General Form

General Form

```
Console.WriteLine(TextString);
Console.WriteLine(Object);
```

The Console.WriteLine method is overloaded, so that you can pass it a string argument or the name of an object.

### The Console.WriteLine Method—Examples

Example

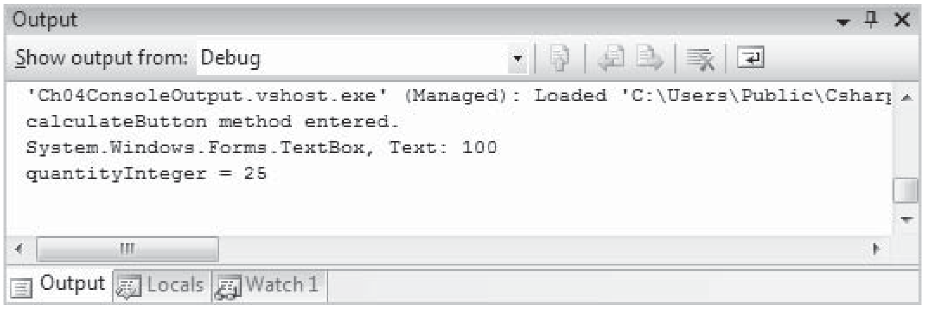
```
Console.WriteLine("calculateButton method entered.");
Console.WriteLine(quantityTextBox);
Console.WriteLine("quantityInteger = " + quantityInteger);
```

When the Console.WriteLine method executes, its output appears in the Output window. Figure 4.20 shows the output of the three example statements above. Notice the second line of output, for quantityTextBox—the class of the object displays along with its current contents.

*Note*: If the Output window is not displaying, select *View/Other Windows/Output*.

**Figure 4.20**

*The Output window shows the output of the* `Console.WriteLine` *method.*



```
'Ch04ConsoleOutput.vshost.exe' (Managed): Loaded 'C:\Users\Public\Csharp
calculateButton method entered.
System.Windows.Forms.TextBox, Text: 100
quantityInteger = 25
```

You may find it useful to place `WriteLine` methods in `if` statements so that you can see which branch the logic followed.

```
if (intCount > 10)
{
    Console.WriteLine("Count is greater than 10.");
    // Other processing.
}
else
{
    Console.WriteLine("Count is not greater than 10.");
    // Other processing.
}
```

An advantage of using `WriteLine`, rather than the other debugging techniques that follow, is that you do not have to break program execution.

## Pausing Execution with the Break All Button

You can click on the *Break All* button to pause execution. This step places the project into Debugging mode at the current line. However, generally you will prefer to break in the middle of a method. To choose the location of the break, you can force a break with a breakpoint.

## Forcing a Break

During the debugging process, often you want to stop at a particular location in code and watch what happens (which branch of an `if/else`; which methods were executed; the value of a variable just before or just after a calculation). You can force the project to break by inserting a **breakpoint** in code.

To set a breakpoint, place the mouse pointer in the gray margin indicator area at the left edge of the Editor window on a line of executable code and click; the line will be highlighted in red and a large red dot will display in the margin indicator (Figure 4.21).

After setting a breakpoint, start execution. When the project reaches the breakpoint, it will halt, display the line, and go into debug mode.

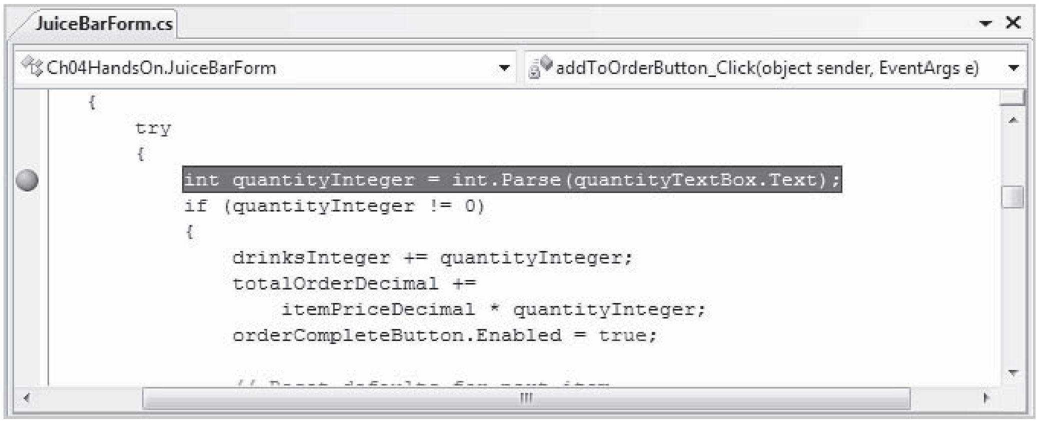
You can remove a breakpoint by clicking again in the gray margin area, or clear all breakpoints from the *Debug* menu.

✓ TIP

Place the insertion point in the line you want as a breakpoint and press F9. Press F9 again to toggle the breakpoint off. ■

Figure 4.21

*A program statement with a breakpoint set appears highlighted and a dot appears in the gray margin indicator area.*



```
{
    try
    {
        int quantityInteger = int.Parse(quantityTextBox.Text);
        if (quantityInteger != 0)
        {
            drinksInteger += quantityInteger;
            totalOrderDecimal +=
                itemPriceDecimal * quantityInteger;
            orderCompleteButton.Enabled = true;
```

## Checking the Current Values of Expressions

You can quickly check the current value of an expression such as a variable, a control, a Boolean expression, or an arithmetic expression. During debug mode, display the Editor window and point to the name of the expression that you want to view; a small label, called a *DataTip*, pops up and displays the current contents of the expression.

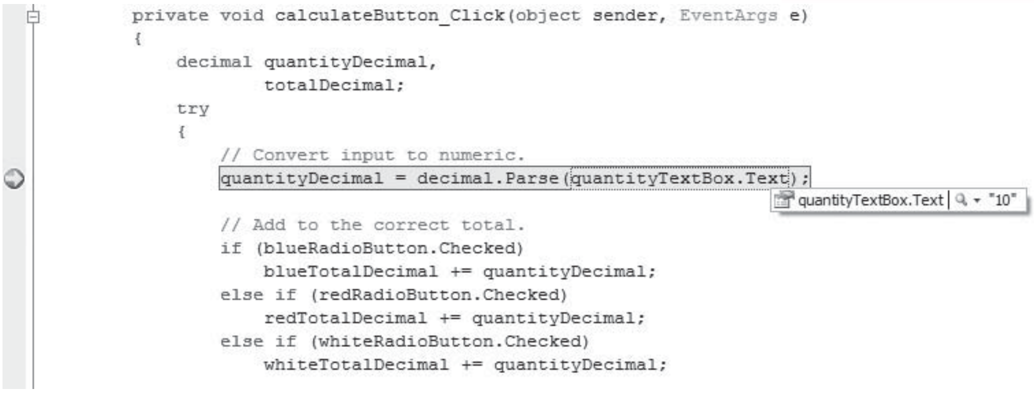
The steps for viewing the contents of a variable during run time are

1. Break the execution using a breakpoint.
2. If the code does not appear in the Editor, click on the editor's tab in the Document window.
3. Point to the variable or expression in the current procedure that you wish to view.

The current contents of the expression will pop up in a label (Figure 4.22), when the expression is in scope.

Figure 4.22

*Point to a variable name in code and its current value displays in a DataTip.*



```
private void calculateButton_Click(object sender, EventArgs e)
{
    decimal quantityDecimal,
            totalDecimal;
    try
    {
        // Convert input to numeric.
        quantityDecimal = decimal.Parse(quantityTextBox.Text);

        // Add to the correct total.
        if (blueRadioButton.Checked)
            blueTotalDecimal += quantityDecimal;
        else if (redRadioButton.Checked)
            redTotalDecimal += quantityDecimal;
        else if (whiteRadioButton.Checked)
            whiteTotalDecimal += quantityDecimal;
```

## Stepping through Code

The best way to debug a project is to thoroughly understand what the project is doing every step of the way. Previously, this task was performed by following each line of code manually to understand its effect. You can now use the Visual Studio stepping tools to trace program execution line by line and see the progression of the program as it executes through your code.

You step through code at debug time. You can use one of the techniques already mentioned to break execution or choose one of the stepping commands at design time; the program will begin running and immediately transfer to debug time.

The three stepping commands on the *Debug* menu are *Step Into*, *Step Over*, and *Step Out*. You also can use the toolbar buttons for stepping or the keyboard shortcuts shown on the menu (refer to Figure 4.19).

These commands force the project to execute a single line at a time and to display the Editor window with the current statement highlighted. As you execute the program, by clicking a button, for example, the Click event occurs. Execution transfers to the event-handling method, the Editor window for that method appears on the screen, and you can follow line-by-line execution.

### Step Into

Most likely you will use the ***Step Into*** command more than the other two stepping commands. When you choose *Step Into* (from the menu, the toolbar button, or F11), the next line of code executes and the program pauses again in debug time. If the line of code is a call to another method, the first line of code of the called method displays.

To continue stepping through your program execution, continue choosing the *Step Into* command. When a method is completed, your form will display again, awaiting an event. You can click on one of the form's buttons to continue stepping through code in an event-handling method. If you want to continue execution without stepping, choose the *Continue* command (from the menu, the toolbar button, or F5). *Note*: The keyboard shortcuts may differ, depending on the keyboard mapping selected in the *Options* dialog box. The shortcuts shown here are for C# development settings; if you are using general development settings, perhaps in a shared environment with other language development, the keyboard shortcuts may differ.

### Step Over

The ***Step Over*** command also executes one line of code at a time. The difference between *Step Over* and *Step Into* occurs when your code has calls to other methods. *Step Over* displays only the lines of code in the current method being analyzed; it does not display lines of code in the called methods.

You can choose *Step Over* from the menu, from the toolbar button, or by pressing F10. Each time you choose the command, one more program statement executes.

### Step Out

You use the third stepping command when you are stepping through a called method. The ***Step Out*** command continues rapid execution until the called method completes, and then returns to debug mode at the statement following the call, that is, the next line of the calling method.

### Continuing Program Execution

When you have seen what you want to see, continue rapid execution by pressing F5 or choosing *Continue* from the Debug toolbar or the *Debug* menu. If you want to restart execution from the beginning, choose the *Restart* command.

### Stopping Execution

Once you have located a problem in the program's code, usually you want to stop execution, correct the error, and run again. Stop execution by selecting *Stop Debugging* from the *Debug* menu or the toolbar button, or press the keyboard shortcut: Shift + F5.

*Note*: The keyboard shortcuts differ depending on the keyboard mapping scheme selected in *Tools / Options / Environment / Keyboard* with *Show All Settings* selected.

### Edit and Continue

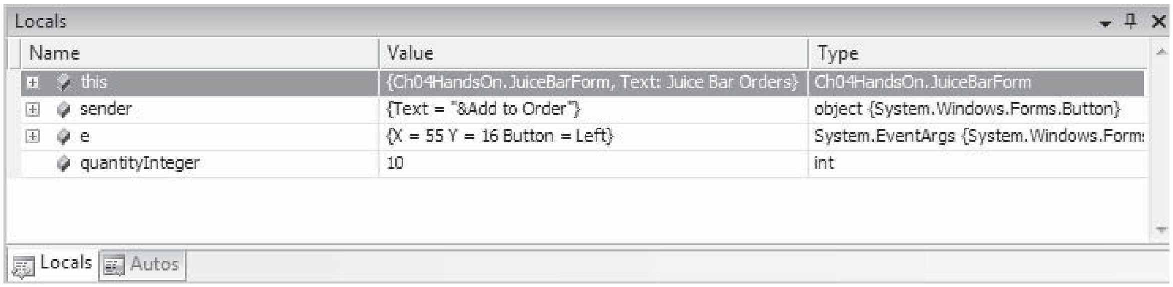
C# 2005 introduced a new feature: Edit and Continue. You can use this feature to save time when debugging programs. When your program goes into Debugging mode and you make minor modifications to the code in the Editor, you may be able to continue execution without stopping to recompile. Press F5 or choose *Debug / Continue*. If the changes to the code are too major to continue without recompiling, the debugger does not allow the changes. Stop program execution, make the changes, and recompile the program.

## The Locals Window

Sometimes you may find that the **Locals window** displays just the information that you want (Figure 4.23). The Locals window displays all objects and variables that are within scope at debug time. That means that if you break execution in the calculateButton_Click event method, all variables local to that method display. You also can expand the `this` entry to see the state of the form's controls and the values of class-level variables. Display the Locals window from the toolbar button or the *Debug / Windows / Locals* menu item, which appears only when a program is running, either in run time or debug mode.

**Figure 4.23**

*The Locals window shows the values of the local variables that are within scope of the current statement.*



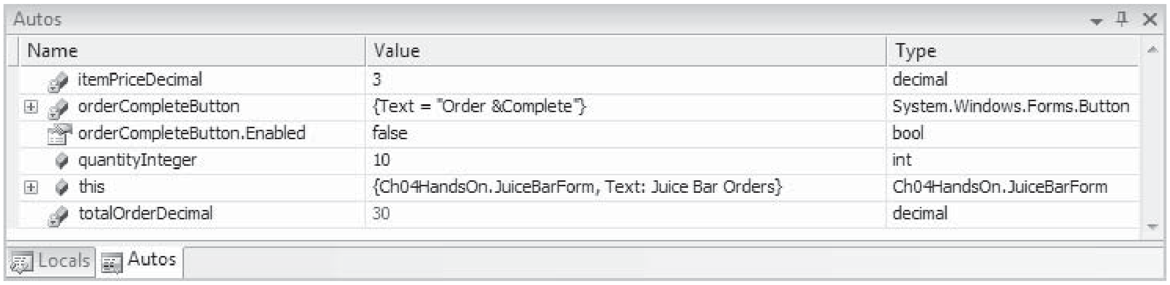
| Name | Value | Type |
|---|---|---|
| ⊞ this | {Ch04HandsOn.JuiceBarForm, Text: Juice Bar Orders} | Ch04HandsOn.JuiceBarForm |
| ⊞ sender | {Text = "&Add to Order"} | object {System.Windows.Forms.Button} |
| ⊞ e | {X = 55 Y = 16 Button = Left} | System.EventArgs {System.Windows.Form: |
| quantityInteger | 10 | int |

### The Autos Window

Another helpful debugging window is the **Autos window**. The Autos window "automatically" displays all variables and control contents that are referenced in the current statement and a few statements on either side of the current one (Figure 4.24). Note that the highlighted line in the Editor window is about to execute next; the "current" statement is the one just before the highlighted one.

**Figure 4.24**

*The Autos window automatically adjusts to show the variables and properties that appear in the previous few lines and the next few lines.*



| Name | Value | Type |
|---|---|---|
| itemPriceDecimal | 3 | decimal |
| ⊞ orderCompleteButton | {Text = "Order &Complete"} | System.Windows.Forms.Button |
| orderCompleteButton.Enabled | false | bool |
| quantityInteger | 10 | int |
| ⊞ this | {Ch04HandsOn.JuiceBarForm, Text: Juice Bar Orders} | Ch04HandsOn.JuiceBarForm |
| totalOrderDecimal | 30 | decimal |

You can view the Autos window when your program stops at a breakpoint. Click on the Autos window tab if it appears, or open it using the *Debug / Windows / Autos* menu item. Again, you must be in either run time or debug mode to see the menu item.

*Note*: The Autos window is not available in the Express Edition.

✓ TIP

To use any of the debugging windows, you must be in debug mode. ■

## Debugging Step-by-Step Tutorial

In this exercise, you will learn to set a breakpoint; pause program execution; single-step through program instructions; display the current values of properties, variables, and conditions; and debug a C# program.

### Test the Project

**STEP 1:** Open the debugging project from the StudentData folder, which you downloaded from the text Web site (www.mhhe.com/C#2008). The project is found in the Ch04Debug folder.

**STEP 2:** Run the program.

**STEP 3:** Enter color Blue and quantity 100, and press Enter or click on the *Calculate* button.

**STEP 4:** Enter another color Blue and quantity 50, and press Enter. Are the totals correct?

**STEP 5:** Enter color Red and quantity 30, and press Enter.

**STEP 6:** Enter color Red and quantity 10, and press Enter. Are the totals correct?

**STEP 7:** Enter color White and quantity 50, and press Enter.

**STEP 8:** Enter color White and quantity 100, and press Enter. Are the totals correct?

**STEP 9:** Exit the project. You are going to locate and correct the errors in the red and white totals.

### Break and Step Program Execution

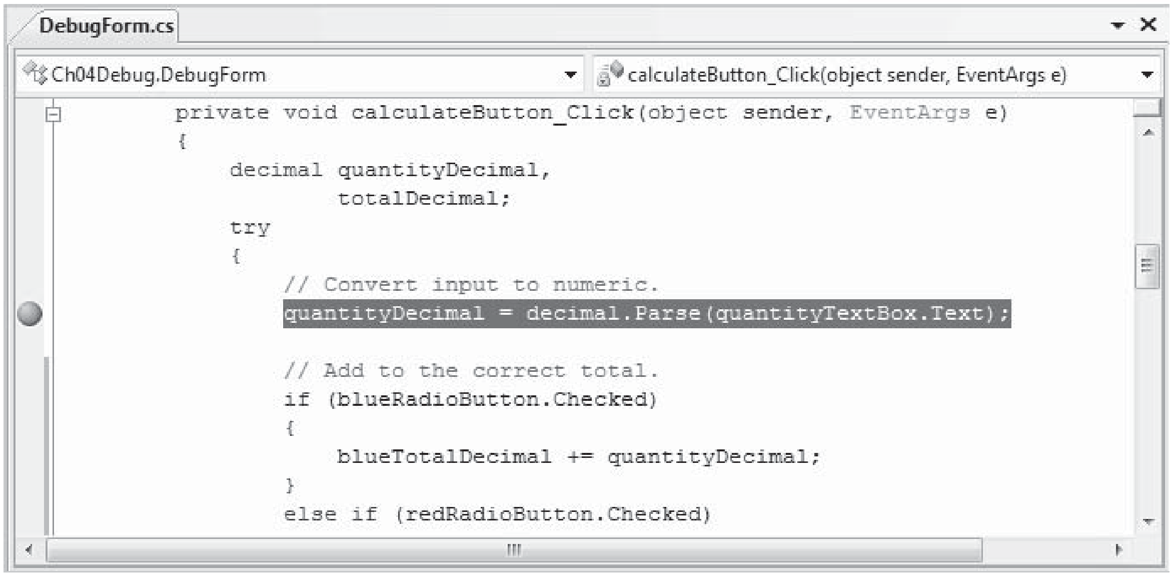
**STEP 1:** Display the program code. Scroll to locate this line, which is the first calculation line in the calculateButton_Click event method:

```
quantityDecimal = decimal.Parse(quantityTextBox.Text);
```

**STEP 2:** Click in the gray margin indicator area to set a breakpoint on the selected line. Your screen should look like Figure 4.25.

**Figure 4.25**

*A program statement with a breakpoint set appears highlighted, and a dot appears in the gray margin indicator area.*



```
        private void calculateButton_Click(object sender, EventArgs e)
        {
            decimal quantityDecimal,
                    totalDecimal;
            try
            {
                // Convert input to numeric.
                quantityDecimal = decimal.Parse(quantityTextBox.Text);

                // Add to the correct total.
                if (blueRadioButton.Checked)
                {
                    blueTotalDecimal += quantityDecimal;
                }
                else if (redRadioButton.Checked)
```

**STEP 3:** Run the project, enter Red and quantity 30, and press Enter. The project will transfer control to the `calculateButton_Click` method, stop when the breakpoint is reached, highlight the current line, and enter debug time (Figure 4.26). If the form is on top of the IDE window, click on the IDE or its taskbar button to make the VS IDE window appear on top. *Note*: The highlighted line has not yet executed.

**STEP 4:** Press the F11 key, which causes C# to execute the current program statement (the assignment statement). (F11 is the keyboard shortcut for *Debug / Step Into*.) The statement is executed, and the highlight moves to the next statement (the `if` statement).

**STEP 5:** Press F11 again; the condition (blueRadioButton.Checked) is tested and found to be *false*.

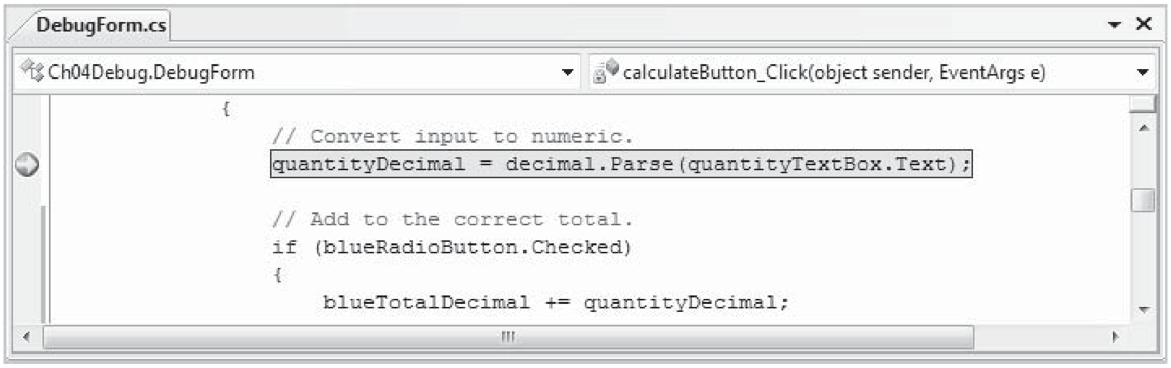
**STEP 6:** Continue pressing F11 a few more times and watch the order in which program statements execute.

**TIP**

You can change the current line of execution in debug mode by dragging the yellow current-line indicator arrow on the left side ■

**Figure 4.26**

*When a breakpoint is reached during program execution, C# enters debug time, displays the Editor window, and highlights the breakpoint line.*



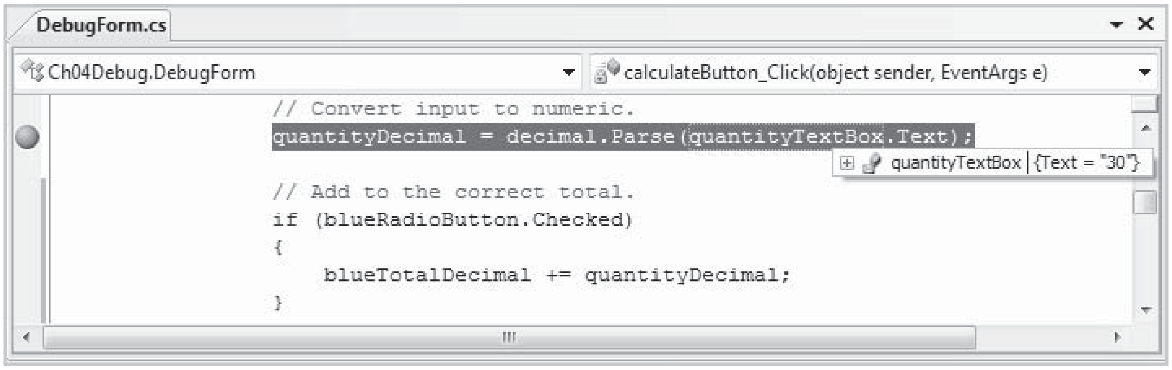
```
                {
                    // Convert input to numeric.
                    quantityDecimal = decimal.Parse(quantityTextBox.Text);

                    // Add to the correct total.
                    if (blueRadioButton.Checked)
                    {
                        blueTotalDecimal += quantityDecimal;
```

## View the Contents of Properties, Variables, and Boolean Expressions

**STEP 1:** Scroll up if necessary and point to `quantityTextBox.Text` in the breakpoint line; the contents of the Text property pop up (Figure 4.27).

**Figure 4.27**

*Point to a property reference in code and the current contents pop up.*



DebugForm.cs
Ch04Debug.DebugForm
calculateButton_Click(object sender, EventArgs e)
quantityTextBox | {Text = "30"}

```
                    // Convert input to numeric.
                    quantityDecimal = decimal.Parse(quantityTextBox.Text);

                    // Add to the correct total.
                    if (blueRadioButton.Checked)
                    {
                        blueTotalDecimal += quantityDecimal;
                    }
```

**STEP 2:** Point to `quantityDecimal` and view the contents of that variable. Notice that the Text property is enclosed in quotes and the numeric variable is not.

**STEP 3:** Point to `blueRadioButton.Checked` in the `if` statement; then point to `redRadioButton.Checked`. You can see the Boolean value for each of the radio buttons.

**STEP 4:** Point to `redTotalDecimal` to see the current value of that total variable. This value looks correct, since you just entered 30, which was added to the total.

## Continue Program Execution

**STEP 1:** Press F5, the keyboard shortcut for the *Continue* command. The *Continue* command continues execution.

If the current line is any line other than the closing brace } of the method, execution continues and your form reappears. If the current line is }, you may have to click on your project's Taskbar button to make the form reappear.

**STEP 2:** Enter color Red and quantity 10. When you press Enter, program execution will again break at the breakpoint.

**STEP 3:** The 10 you just entered should be added to the 30 previously entered for Red, producing 40 in the Red total.

**STEP 4:** Use the *Step Into* button on the Debug toolbar to step through execution. Keep pressing *Step Into* until the 10 is added to redTotalDecimal. Display the current contents of the total. Can you see what the problem is?

*Hint*: redTotalDecimal has only the current amount, not the sum of the two amounts. The answer will appear a little later; try to find it yourself first.

You will fix this error soon, after testing the White total.

TIP

You can execute several lines at once by right-clicking the line you would like to debug next and choosing *Run To Cursor*. ■

## Test the White Total

**STEP 1:** Press F5 to continue execution. If the form does not reappear, click the project's Taskbar button.

**STEP 2:** Enter color White and quantity 100, and press Enter.

When execution halts at the breakpoint, press F5 to continue. This returns to rapid execution until the next breakpoint is reached.

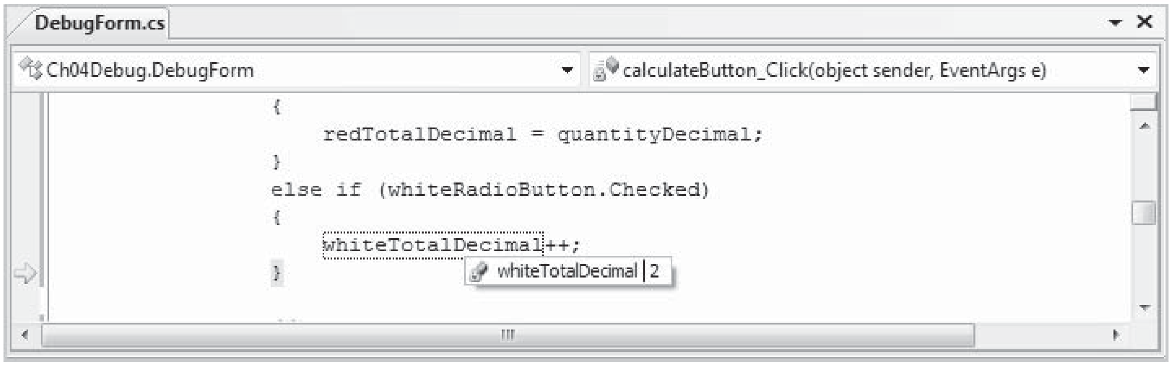
Enter color White and quantity 50, and press Enter.

Press F11 several times when execution halts at the breakpoint until you execute the line that adds the quantity to the White total. Remember that the highlighted line has not yet executed; press *Step Into* one more time, if necessary, to execute the addition statement.

Point to each variable name to see the current values (Figure 4.28). Can you see the problem?

Figure 4.28

*Point to the variable name in code and its current value displays as 2 decimal.*

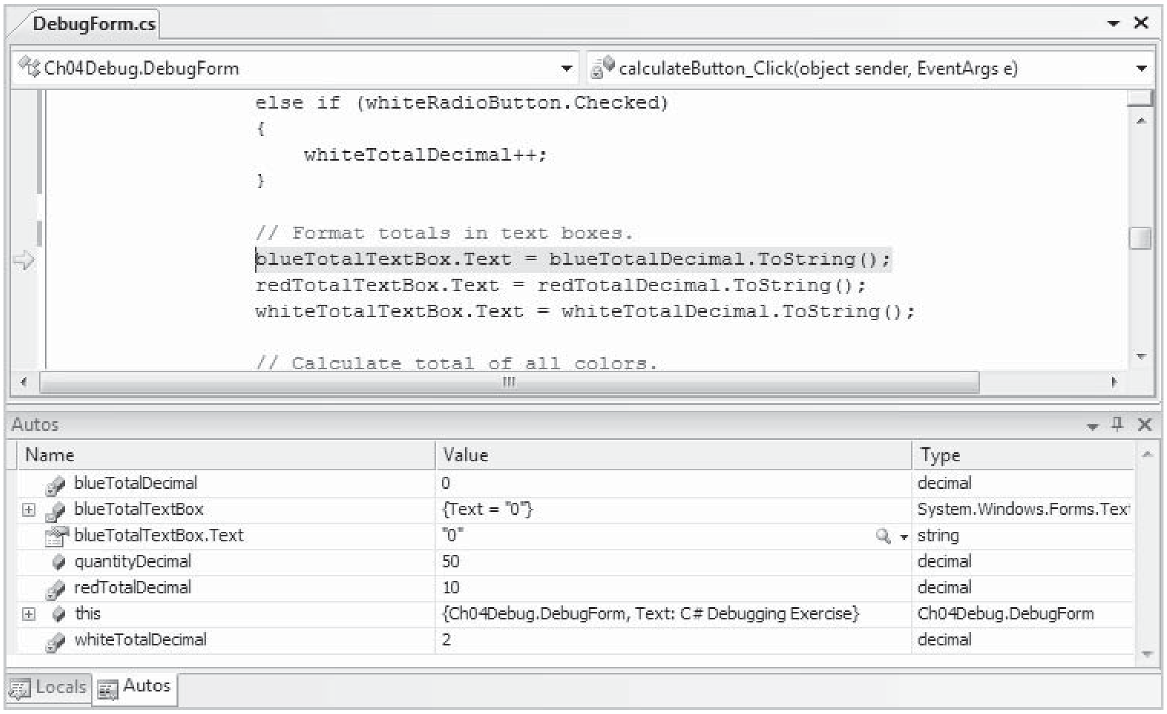


```
{
    redTotalDecimal = quantityDecimal;
}
else if (whiteRadioButton.Checked)
{
    whiteTotalDecimal++;
}
```

**STEP 3:** Display the Autos window by clicking on its tab. If the tab does not appear, select *Debug / Windows / Autos.* The Autos window displays the current value of all properties and variables referred to by a few statements before and after the current statement (Figure 4.29). *Note*: If you are using the Express Edition, you can substitute the Locals window for the Autos window.

**Figure 4.29**

*The Autos window displays the current contents of variables and properties in the statements before and after the current statement.*



```
else if (whiteRadioButton.Checked)
{
    whiteTotalDecimal++;
}

// Format totals in text boxes.
blueTotalTextBox.Text = blueTotalDecimal.ToString();
redTotalTextBox.Text = redTotalDecimal.ToString();
whiteTotalTextBox.Text = whiteTotalDecimal.ToString();

// Calculate total of all colors.
```

| Name | Value | Type |
|---|---|---|
| blueTotalDecimal | 0 | decimal |
| ⊞ blueTotalTextBox | {Text = "0"} | System.Windows.Forms.Tex |
| blueTotalTextBox.Text | "0" | string |
| quantityDecimal | 50 | decimal |
| redTotalDecimal | 10 | decimal |
| ⊞ this | {Ch04Debug.DebugForm, Text: C# Debugging Exercise} | Ch04Debug.DebugForm |
| whiteTotalDecimal | 2 | decimal |

**STEP 4:** Identify all the errors. When you are ready to make the corrections, continue to the next step.

**TIP**

Display keyboard shortcuts on ToolTips, as in Figure 4.30, by selecting *Tools / Customize / Show shortcut keys in ScreenTips.* ■

## Correct the Red Total Error

**STEP 1:** Stop program execution by clicking on the *Stop Debugging* toolbar button (Figure 4.30).

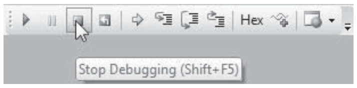


**Figure 4.30**

*Click on the Stop Debugging button on the Debug toolbar to halt program execution.*

**STEP 2:** Locate this line:

```
redTotalDecimal = quantityDecimal;
```

This statement replaces the value of redTotalDecimal with quantityDecimal rather than adding to the total.

**STEP 3:** Change the line to read

```
redTotalDecimal += quantityDecimal;
```

## Correct the White Total Error

**STEP 1:** Locate this line:

```
whiteTotalDecimal++;
```

Of course, this statement adds 1 to the White total, rather than adding the quantity.

**STEP 2:** Correct the line to read

```
whiteTotalDecimal += quantityDecimal;
```

## Test the Corrections

**STEP 1:** Press F5 to start program execution. Enter color White and quantity 100; press Enter.

**STEP 2:** When the program stops at the breakpoint, press F5 to continue.

**STEP 3:** Enter White and 50, and press Enter.

**STEP 4:** At the breakpoint, clear the breakpoint by clicking on the red margin dot for the line.

**STEP 5:** Press F5 to continue and check the White total on the form. It should now be correct.

**STEP 6:** Enter values for Red twice and make sure the total is correct.

**STEP 7:** Test the totals for all three colors carefully and then click *Exit*.

## Test the Exception Handling

**STEP 1:** Set a breakpoint again on the first calculation line in the calculateButton_Click event handler.

**STEP 2:** Run the program, this time entering nonnumeric characters for the amount. Click on *Calculate*; when the program stops at the breakpoint, press F11 repeatedly and watch program execution. The message box should appear.

**STEP 3:** Stop program execution.

## Force a Run-Time Error

For this step, you will use a technique called *commenting out* code. Programmers often turn code lines to comments to test the code without those lines. Sometimes it works well to copy a section of code, comment out the original to keep it unchanged, and modify only the copy. You'll find it easy to uncomment the code later, after you finish testing.

In C# you can turn code lines into comments in several ways: You can add "//" to the beginning of a line, surround a group of lines with "/*" and "*/", or use the *Comment out the selected lines* toolbar button.

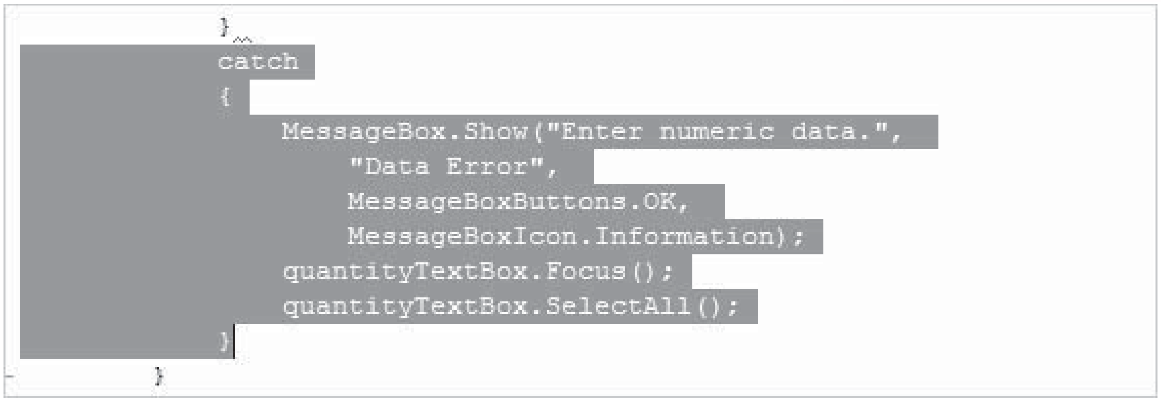
**STEP 1:** Select *Delete All Breakpoints* from the *Debug* menu if the menu item is available. The item is available only when there are breakpoints set in the program. Click *Yes* on the confirmation dialog.

**STEP 2:** At the left end of the line with the `try` statement, add two slashes, turning the line into a comment.

**STEP 3:** Scroll down and locate the exception-handling code. Highlight the lines beginning with `catch` and ending with the closing brace for the `catch` block (Figure 4.31).

**Figure 4.31**

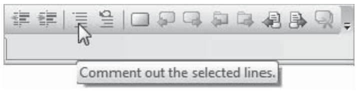
*Select the lines to convert to comments for debugging.*



```
          }
          catch
          {
              MessageBox.Show("Enter numeric data.",
                  "Data Error",
                  MessageBoxButtons.OK,
                  MessageBoxIcon.Information);
              quantityTextBox.Focus();
              quantityTextBox.SelectAll();
          }
      }
```

**STEP 4:** Click on the *Comment out the selected lines* button on the Text Editor toolbar (Figure 4.32). The editor adds double slashes to the start of each of the selected lines.

**Figure 4.32**

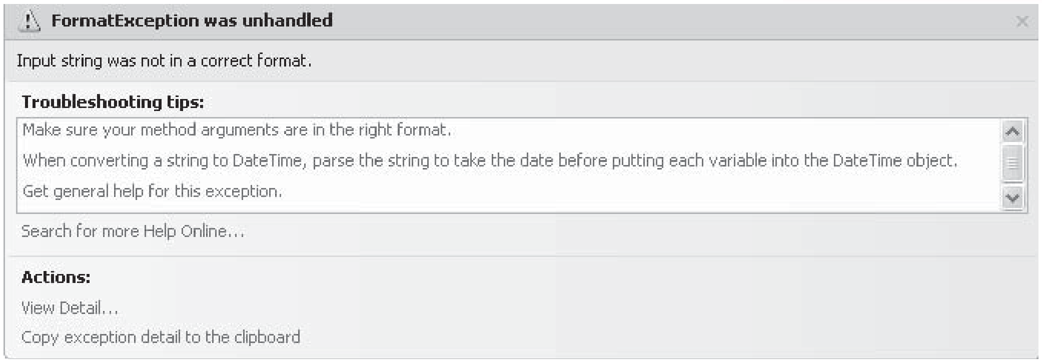


*Click the Comment out the selected lines toolbar button to temporarily make program lines into comments.*

**STEP 5:** Run the project. This time click the *Calculate* button without entering a quantity.
A run-time error will occur (Figure 4.33).
Click *Stop Debugging* to cancel execution.

**Figure 4.33**

*The missing data cause an exception and run-time error.*



FormatException was unhandled

Input string was not in a correct format.

**Troubleshooting tips:**

Make sure your method arguments are in the right format.

When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.

Get general help for this exception.

Search for more Help Online...

**Actions:**

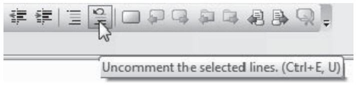
View Detail...

Copy exception detail to the clipboard

**STEP 6:** After you are finished testing the program, select the commented lines and click on the *Uncomment the selected lines* button (Figure 4.34).

*Note*: You can just click an insertion point in a line or select the entire line when you comment and uncomment lines.

**Figure 4.34**



*Click the Uncomment the selected lines toolbar button after testing the program.*

# Summary

1. C# uses the `if/else` statement to make decisions. An `else` clause is optional and specifies the action to be taken if the expression evaluates *false*. If there are multiple statements for the `if` or `else`, the statements must be enclosed in braces.
2. UML activity diagrams can help visualize the logic of an `if/else` statement.
3. The Boolean expressions for an `if` statement are evaluated for *true* or *false*.
4. Boolean expressions can be composed of the relational operators, which compare items for equality, greater than, or less than. The comparison of numeric values is based on the quantity of the number, while character comparisons are based on the ANSI code table. Strings may use the equal and not equal operators or the string methods `Equals` and `CompareTo`.
5. The `ToUpper` and `ToLower` methods of the String class can convert a text value to upper- or lowercase.
6. The `&&`, `||`, `&`, and `|` logical operators may be used to combine multiple expressions. With the `&&` operator, both expressions must be true for the entire expression to evaluate *true*. For the `||` operator, if either or both expressions are true, the entire expression evaluates as *true*. When both `&&`

and `||` are used in a compound Boolean expression, the `&&` expression is evaluated before the `||` expression. `||` and `&&` short circuit the expression so that the second part of a compound expression may not be tested; use a single symbol (`|` or `&`) to not short circuit and thus force evaluation of all expressions.

7. A nested `if` statement contains an `if` statement within either the *true* or *false* actions of a previous `if` statement. An `else` clause always applies to the last unmatched `if` regardless of indentation.
8. The state of radio buttons and check boxes is better tested with `if` statements in the event handler for a button, rather than coding event handlers for the radio button or check box. Use individual `if` statements for check boxes and nested `if` statements for multiple radio buttons.
9. The `MessageBox.Show` method can display a multiline message if you concatenate a NewLine character (`\n`) to specify a line break.
10. You can choose to display multiple buttons in a message box. The `MessageBox.Show` method returns an object of the DialogResult class, which you can check using the DialogResult constants.
11. Data validation checks the reasonableness or appropriateness of the value in a variable or property.
12. The `switch` statement can test an expression for multiple values and substitute for nested `if` statements.
13. You can assign the event-handling method for a control in the Properties window. A single method can be assigned to multiple controls, so that the controls share the event handler.
14. You can use the `sender` argument in an event-handling method to determine which control caused the method to execute.
15. One method can call another method. To call an event-handling method, you must supply the `sender` and `e` arguments.
16. A variety of debugging tools are available in Visual Studio. These include writing to the Output window, breaking program execution, displaying the current contents of variables, and stepping through code.

## Key Terms

ANSI code *162*  
Autos window *202*  
Boolean expression *160*  
breakpoint *198*  
call *188*  
comparison operators *160*  
compound Boolean expression *166*  
condition *160*  
`Console.WriteLine` method *197*  
DialogResult object *178*  
`if` statement *159*  
Locals window *201*  
logical operator *166*  
nested `if` *168*  
NewLine (`\n`) character *176*  
relational operator *160*  
short circuit *167*  
*Step Into* *200*  
*Step Out* *200*  
*Step Over* *200*  
`switch` statement *182*  
`ToLower` method *165*  
`ToUpper` method *165*  
validation *180*

## Review Questions

1. What is the general format of the statement used to code decisions in an application?
2. What is a Boolean expression?
3. Explain the purpose of relational operators and logical operators.
4. How does a comparison performed on numeric data differ from a comparison performed on string data?
5. How does C# compare the Text property of a text box?
6. Why would it be useful to include the `ToUpper` method in a comparison?
7. Name the types of items that can be used in a comparison.
8. Explain a Boolean variable test for *true* and *false*. Give an example.
9. Give an example of a situation where nested `if`s would be appropriate.
10. Define the term *validation*. When is it appropriate to do validation?
11. Define the term *checking a range*.
12. When would it be appropriate to use a `switch` structure? Give an example.
13. Explain the difference between *Step Into* and *Step Over*.
14. What steps are necessary to view the current contents of a variable during program execution?

## Programming Exercises

4.1 Lynette Rifle owns an image consulting shop. Her clients can select from the following services at the specified regular prices: Makeover $125, Hair Styling $60, Manicure $35, and Permanent Makeup $200. She has distributed discount coupons that advertise discounts of 10 percent and 20 percent off the regular price. Create a project that will allow the receptionist to select a discount rate of 10 percent, 20 percent, or none, and then select a service. Display the total price for the currently selected service and the total due for all services. A visit may include several services. Include buttons for *Calculate*, *Clear*, *Print*, and *Exit*.

4.2 Modify Programming Exercise 4.1 to allow for sales to additional patrons. Include buttons for *Next Patron* and *Summary*. When the receptionist clicks the *Summary* button, display in a summary message box the number of clients and the total dollar value for all services rendered. For *Next Patron*, confirm that the user wants to clear the totals for the current customer.

4.3 Create a project to compute your checking account balance.
*Form*: Include radio buttons to indicate the type of transaction: deposit, check, or service charge. A text box will allow the user to enter the amount of the transaction. Display the new balance in a ReadOnly text box or a label. Calculate the balance by adding deposits and subtracting service charges and checks. Include buttons for *Calculate*, *Clear*, *Print*, and *Exit*.

4.4 Add validation to Programming Exercise 4.3. Display a message box if the new balance would be a negative number. If there is not enough money to cover a check, do not deduct the check amount. Instead, display a message box with the message "Insufficient Funds" and deduct a service charge of $10.

4.5 Modify Programming Exercise 4.3 or 4.4 by adding a *Summary* button that displays the total number of deposits, the total dollar amount of deposits, the number of checks, and the dollar amount of the checks. Do not include checks that were returned for insufficient funds, but do include the service charges. Use a message box to display the summary information.

4.6 Piecework workers are paid by the piece. Workers who produce a greater quantity of output are often paid at a higher rate.

*Form*: Use text boxes to obtain the person's name and the number of pieces completed. Include a *Calculate* button to display the dollar amount earned. You will need a *Summary* button to display the total number of pieces, the total pay, and the average pay per person. A *Clear* button should clear the name and the number of pieces for the current employee and a *Clear All* button should clear the summary totals after confirming the operation with the user.

Include validation to check for missing data. If the user clicks on the *Calculate* button without first entering a name and the number of pieces, display a message box. Also, you need to make sure to not display a summary before any data are entered; you cannot calculate an average when no items have been calculated. You can check the number of employees in the Summary event handler or disable the *Summary* button until the first order has been calculated.

| **Pieces completed** | **Price paid per piece for all pieces** |
|---|---|
| 1–199 | .50 |
| 200–399 | .55 |
| 400–599 | .60 |
| 600 or more | .65 |

4.7 Modify Programming Exercise 2.2 (the flag viewer) to treat radio buttons and check boxes in the proper way. Include a *Display* button and check the settings of the radio buttons and check boxes in the button's event handler, rather than making the changes in event handlers for each radio button and check box.

*Note*: For help in basing a new project on an existing project, see "Copy and Move a Windows Project" in Appendix C.

4.8 Create an application to calculate sales for Catherine's Catering. The program must determine the amount due for an event based on the number of guests, the menu selected, and the bar options. Additionally, the program maintains summary figures for multiple events.

*Form*: Use a text box to input the number of guests and radio buttons to allow a selection of Prime Rib, Chicken, or Pasta. Check boxes allow the user to select an Open Bar and/or Wine with Dinner. Include buttons for *Calculate*, *Clear*, *Summary*, and *Exit*. Display the amount due for the event in a label or ReadOnly text box.

**Rates per Person**

| | |
|---|---|
| Prime Rib | 25.95 |
| Chicken | 18.95 |
| Pasta | 12.95 |
| Open Bar | 25.00 |
| Wine with Dinner | 8.00 |

*Summary*: Display the number of events and the total dollar amount in a message box. Prompt the user to determine if he or she would like to clear the summary information. If the response is Yes, set the number of events and the total dollar amount to zero. Do not display the summary message box if there is no summary information. (Either disable the *Summary* button until a calculation has been made or test the total for a value.)

# Case Studies

## Custom Supplies Mail Order

Calculate the amount due for an order. For each order, the user should enter the following information into text boxes: customer name, address, city, state (two-letter abbreviation), and ZIP code. An order may consist of multiple items. For each item, the user will enter the product description, quantity, weight, and price into text boxes.

You will need buttons for *Add This Item*, *Update Summary*, *Clear*, and *Exit*.

For the *Add This Item* button, validate the quantity, weight, and price. Each must be present and numeric. For any bad data, display a message box. Calculate the charge for the current item and add the charge and weight into the appropriate totals, but do not display the summary until the user clicks the *Update Summary* button. Do not calculate shipping and handling on individual items; rather, calculate shipping and handling on the entire order.

When the user clicks *Add This Item* for a new order, the customer information should be disabled so that the state cannot be changed until the next customer.

When the *Update Summary* button is clicked, calculate the sales tax, shipping and handling, and the total amount due for the order. Sales tax is 8 percent of the total charge and is charged only for shipments to a California address. Do not charge sales tax on the shipping and handling charges. The shipping and handling should be calculated only for a complete order.

*Optional*: Disable the *Add This Item* button when the *Summary* button is pressed.

The *Clear* button clears the data and totals for the current customer.

The shipping and handling charges depend on the weight of the products. Calculate the shipping charge as $0.25 per pound and add that amount to the handling charge (taken from the following table).

| Weight | Handling |
|---|---|
| Less than 10 pounds | $1.00 |
| 10 to 100 pounds | $3.00 |
| Over 100 pounds | $5.00 |

Display the entire amount of the bill in controls titled *Dollar amount due*, *Sales tax*, *Shipping and handling*, and *Total amount due*.

| Test data<br>Description | Quantity | Weight | Price |
|---|---|---|---|
| Planter | 2 | 3 | 19.95 |
| Mailbox | 1 | 2 | 24.95 |
| Planter Box | 2 | 3 | 19.95 |

**Test data output for taxable (if shipped to a California address)**

| | |
|---|---|
| Dollar Amount Due | $104.75 |
| Sales Tax | 8.38 |
| Shipping and Handling | 6.50 |
| Total Amount Due | 119.63 |

**Test data output for nontaxable (if shipped outside of California)**

| | |
|---|---|
| Dollar Amount Due | $104.75 |
| Sales Tax | 0.00 |
| Shipping and Handling | 6.50 |
| Total Amount Due | 111.25 |

## Christopher's Car Center

Create a project that determines the total amount due for the purchase of a vehicle. Include text boxes for the base price and the trade-in amount. Check boxes will indicate if the buyer wants additional accessories such as a stereo system, leather interior, and/or computer navigation. A group box for the exterior finish will contain radio buttons for Standard, Pearlized, or Customized detailing.

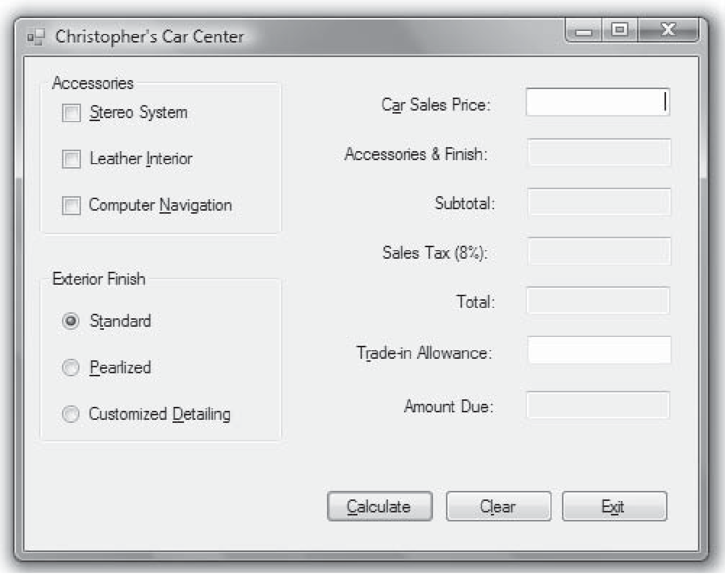
Have the trade-in amount default to zero; that is, if the user does not enter a trade-in amount, use zero in your calculation. Validate the values from the text boxes, displaying a message box if necessary.

To calculate, add the price of selected accessories and exterior finish to the base price and display the result in a control called *Subtotal*. Calculate the sales tax on the subtotal and display the result in a Sales Tax control. Calculate and display the total in a *Total* control. Then subtract any trade-in amount from the total and display the result in an *Amount Due* control.

Include buttons for *Calculate*, *Clear*, and *Exit*. The *Calculate* button must display the total amount due after trade-in.

*Hint*: Recall that you can make an ampersand appear in the Text property of a control by including two ampersands. See the tip on page 82 (Chapter 2).

| Item | Price |
|---|---|
| Stereo System | 425.76 |
| Leather Interior | 987.41 |
| Computer Navigation | 1,741.23 |
| Standard | No additional charge |
| Pearlized | 345.72 |
| Customized Detailing | 599.99 |
| Tax Rate | 8% |



## Xtreme Cinema

Design and code a project to calculate the amount due for rentals. Movies may be in Blu-Ray (BD) format or DVD format. BD rent for $5.00 each and DVDs rent for $4.50. New releases are $1 additional charge.

On the form include a text box to input the movie title and radio buttons to indicate whether the movie is in DVD or BD format. Use one check box to indicate whether the person is a member; members receive a 10 percent discount. Another check box indicates a new release.

Use buttons for *Calculate*, *Clear for Next Item*, *Order Complete*, *Summary*, and *Exit*. The *Calculate* button should display the item amount and add to the subtotal. The *Clear for Next Item* clears the check box for new releases, the movie title, and the radio buttons; the member check box cannot be changed until the current order is complete. Include validation to check for missing data. If the user clicks on the *Calculate* button without first entering the movie title and selecting the movie format, display a message box.

For the *Order Complete* button, first confirm the operation with the user and clear the controls on the form for a new customer.

The *Summary* button displays the number of customers and the sum of the rental amounts in a message box. Make sure to add to the customer count and rental sum for each customer order.

## Cool Boards

Cool Boards does a big business in shirts, especially for groups and teams. They need a project that will calculate the price for individual orders, as well as a summary for all orders.

The store employee will enter the orders in an order form that has text boxes for customer name and order number. To specify the shirts, use a text box for the quantity, radio buttons to select the size (small, medium, large, extra large, and XXL), and check boxes to specify a monogram and/or a pocket. Display the shirt price for the current order and the order total in ReadOnly text boxes or labels.

Include buttons to add a shirt to an order, clear the current item, complete the order, and display the summary of all orders. Do not allow the summary to display if the current order is not complete. Also, disable the text boxes for customer name and order number after an order is started; enable them again when the user clicks on the button to begin a new order. Confirm the operation before clearing the current order.

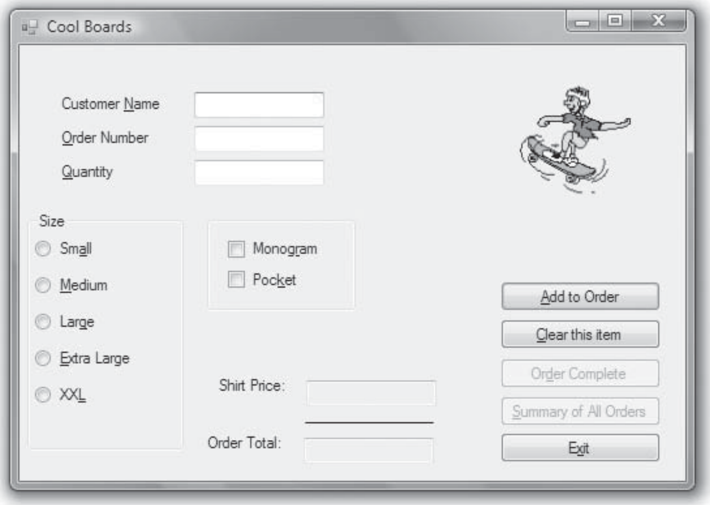
When the user adds shirts to an order, validate the quantity, which must be greater than zero. If the entry does not pass the validation, do not perform any calculations but display a message box and allow the user to correct the value. Determine the price of the shirts from the radio buttons and check boxes for the monogram and pockets. Multiply the quantity by the price to determine the extended price, and add to the order total and summary total.

Use constants for the shirt prices.

Display the order summary in a message box. Include the number of shirts, the number of orders, and the dollar total of the orders.

**Prices for the shirts**

| | |
|---|---|
| Small, medium, and large | $10 |
| Extra large | 11 |
| XXL | 12 |
| Monogram | Add $2 |
| Pocket | Add $1 |


Cool Boards
Customer Name
Order Number
Quantity
Size
○ Small
○ Medium
○ Large
○ Extra Large
○ XXL
□ Monogram
□ Pocket
Shirt Price:
Order Total:
Add to Order
Clear this item
Order Complete
Summary of All Orders
Exit