The If...Then Statement

The If...Then statement is a *conditional control structure*, also called a *decision structure*, which executes a set of statements when a condition is true. The If...Then statement takes the form:

```
If condition Then
    statements
End If
```

For example, in the following If...Then statement `guess = 7` is the condition, and there is one statement that will execute when this condition is true:

```
If guess = 7 Then
    Me.lblMessage.Text = "You guessed it!"
End If
```

In the condition, the equal sign (=) is used as a relational operator to determine if the value of `guess` is equal to 7. If equal, then the Text property of `lblMessage` is changed. If not, program flow continues to the next statement after the `End If`.

The condition of an If...Then statement is a *Boolean expression*, which evaluates to either True or False. *Relational operators* can be used to form Boolean expressions. There are six relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

TIP The equal sign (=) is used as a relational operator as well as an assignment operator.

TIP The condition of an If...Then statement should never make an equality comparison between floating point numbers because of the possibility of roundoff error.
Chapter 4 Controlling Program Flow with Decision Structures

The NumberGuess application demonstrates the `If...Then` statement:

Running the application, typing 5 in the text box, and then clicking Check Guess displays no message. Typing 7 and clicking Check Guess displays the message “You guessed it!”

A Boolean variable may also be used as the condition of an `If...Then` statement because its value is either `True` or `False`. For example, the `If...Then` statement below uses a Boolean variable as its condition:

```vbnet
Dim gameOver As Boolean = True
If gameOver Then
    Application.Exit() End If
```

Review: TestGrade – part 1 of 5

Create a TestGrade application that prompts the user for a test score and then displays “Good job!” for a test score greater than or equal to 70 when Check Grade is clicked. The application interface should look similar to that shown on the right after typing 75 and clicking Check Grade. Be sure to include a TextChanged event procedure for the text box.

The If...Then...Else Statement

The `If...Then` statement can include an optional `Else` clause that is executed when the `If` condition evaluates to `False`. The `If...Then...Else` statement takes the following form:

```vbnet
If condition Then
    statements
Else
    statements
End If
```
The NumberGuess application could be modified to include an *If...Then...Else* statement:

```vbnet
If guess = SECRETNUMBER Then
    Me.lblMessage.Text = "You guessed it!"
Else
    Me.lblMessage.Text = "Try again."
End If
```

Running the application, typing a number other than 7, and then clicking Check Guess displays the message “Try again.” Typing 7 and then clicking Check Guess displays the message “You guessed it!”

The indentation used in the *If...Then...Else* statement is a code convention that makes the statement easier to read. It has no effect on the execution of the statement. The indentation also makes it easier for a programmer reading the code to follow the logic of the statement.

**Review: TestGrade – part 2 of 5**

Modify the TestGrade application so that “Good job!” is displayed for a score greater than or equal to 70 and “Study more.” is displayed for a score less than 70.

**Review: CircleArea – part 2 of 2**

Modify the CircleArea application created in Chapter 3 to display the message “Negative radii are illegal.” when a negative radius value is entered, otherwise the area of the circle should be displayed.

### Nested If...Then...Else Statements

An *If...Then...Else* statement can contain another *If...Then...Else* or *If...Then* statement, which is said to be *nested*. For example, the NumberGuess application could be modified to give the user a hint:

```vbnet
If guess = SECRETNUMBER Then 'correct
    Me.lblMessage.Text = "You guessed it!"
Else
    If guess < SECRETNUMBER Then 'too low
        Me.lblMessage.Text = "Too low."
    Else 'too high
        Me.lblMessage.Text = "Too high."
    End If
End If
```

Nested statements should be indented as good programming style.
The If...Then...ElseIf Statement

The If...Then...ElseIf statement is used to decide among three or more actions and takes the form:

```
If condition Then
    statements
ElseIf condition Then
    statements
...;
Else
    statements
End If
```

There can be multiple ElseIf clauses, and the last Else clause is optional. For example, there are three possible decisions in the If...Then...ElseIf statement below:

```
If guess = SECRETNUMBER Then    'correct
    Me.lblMessage.Text = "You guessed it!"
ElseIf guess < SECRETNUMBER Then 'too low
    Me.lblMessage.Text = "Too low."
Else
    Me.lblMessage.Text = "Too high."
End If
```

The logic used in developing an If...Then...ElseIf statement is important. For example, when testing a range of numbers, If conditions must be properly ordered because statements are executed only for the first true condition and then program flow continues to the End If.

When choosing between nested If...Then...Else statements and a single If...Then...ElseIf statement, the If...Then...ElseIf is easier to read and understand and is considered better programming style.

Review: TestGrade – part 3 of 5

Modify the TestGrade application so that “Great!” is displayed for a test score greater than or equal to 90, “Good job!” for a test score greater than or equal to 70 and less than 90, and “Study more.” otherwise.

The Select...Case Statement

The Select...Case statement is a conditional control structure that uses the result of an expression to determine which statements to execute. The Select...Case statement is sometimes preferable to the If...Then...ElseIf statement because code may be easier to read. The Select...Case statement takes the form:

```
Select expression
    Case value
        statements
    ...;
    Case Else
        statements
End Select
```
The expression must evaluate to a built-in data type. There can be multiple Case clauses, and the Case Else clause is optional. The value type should match the expression type and can be a single value, a list separated by commas, or a range separated by the keyword To. End Select is required to complete the Select...Case statement.

The Select...Case statement below uses the value of a score to determine the message to display:

```vbnet
Select Case score
    Case 0, 10    'score is 0 or 10
        Me.lblMessage.Text = "Nice try."
    Case 20 To 23    'score is 20, 21, 22, or 23
        Me.lblMessage.Text = "Great!"
    Case Else    'score not 0, 10, 20, 21, 22, 23
        Me.lblMessage.Text = "Invalid score."
End Select
```

Review: Hurricane

The Saffir-Simpson Hurricane Scale provides a rating (a category) depending on the current intensity of a hurricane. Create a Hurricane application that prompts the user for a wind speed and then displays the hurricane category. Display the speed in miles per hour (mph), knots (kts), and kilometers per hour (km/hr). Refer to the Saffir-Simpson Hurricane Scale below for wind speeds. The application interface should be similar to that shown on the right after typing 100 and clicking Category.

Category 1: 74-95 mph or 64-82 kt or 119-153 km/hr
Category 2: 96-110 mph or 83-95 kt or 154-177 km/hr
Category 3: 111-130 mph or 96-113 kt or 178-209 km/hr
Category 4: 131-155 mph or 114-135 kt or 210-249 km/hr
Category 5: greater than 155 mph or 135 kt or 249 km/hr

The Select...Case Is Statement

The Select...Case Is statement compares the result of an expression to a range of values when a relational operator is part of the value. For example, the following statement uses ranges to determine the message to display:

```vbnet
Select Case score
    Case Is < 10    'less than 10
        Me.lblMessage.Text = "Nice try."
    Case Is < 25    'less than 25
        Me.lblMessage.Text = "Good."
    Case Is >= 25    'greater than or equal to 25
        Me.lblMessage.Text = "Great!"
End Select
```
Review: TestGrade – part 4 of 5

Modify the TestGrade application to use a `Select...Case` Is and ranges of values to determine which grade to display. An A is for scores greater than 90, a B is for scores greater than or equal to 80 and less than 90, a C for scores greater than or equal to 70 and less than 80, a D for scores greater than or equal to 60 and less than 70, and a F for scores less than 60.

Generating Random Numbers

Games, simulators, screen savers, and many other types of applications require random numbers. For generating random numbers, Visual Basic includes the built-in `Rnd()` function. This function uses a formula to generate a sequence of numbers and then returns one number from the sequence. Although the numbers in the sequence vary and for most applications can be considered random, the sequence will at some point repeat. Therefore, random numbers in a computer application are referred to as pseudorandom (like random).

The `Rnd()` function returns a number that is greater than or equal to 0 and less than 1. For example, the RandomNumbers application generates and displays six random numbers when Random Numbers is clicked:

![Random Numbers screenshot]

The RandomNumbers application code includes a statement similar to:

```vbnet
Me.lblRandNum1.Text = Rnd()
```

Using `Rnd()` alone generates random numbers greater than or equal to 0 and less than 1. To generate random numbers in a greater range, `Rnd()` is multiplied by the upper limit of the range. This produces numbers from 0 to one less than the upper limit. For example, to generate random numbers greater than or equal to 0 and less than 10, the following expression is used:

```
Rnd() * 10
```

A random number in a specific range is generated by using the following expression:

```
(highNumber - lowNumber + 1) * Rnd() + lowNumber
```
highNumber is the maximum value desired and lowNumber is the minimum value. For example, the following statement generates a number greater than or equal to 10 and less than 31 \( ((30 - 10 + 1) \times \text{Rnd()} + 10 ) \) and then assigns it to a label:

\[
\text{Me.lblRandNum1.Text} = 21 \times \text{Rnd()} + 10
\]

Using this type of expression, the RandomNumbers application can be modified to display six floating point numbers in the range 10 to 30:

Numbers generated by Rnd() are floating point numbers with a decimal portion. To produce random integers (whole numbers), the Int() function can be used. Int() requires a numeric value and then returns the integer portion of that number without rounding. For example, in the following statement an integer is assigned to a label:

\[
\text{Me.lblRandNum1.Text} = \text{Int}(21 \times \text{Rnd()} + 10)
\]

The RandomNumbers application modified to display six integers in the range 10 to 30 looks similar to:

Programs using Rnd() should also include one Randomize() statement in the beginning of the event procedure before the Rnd() function is used. Randomize() initializes the random number generator so that different random numbers are generated from run to run.

**Review: RandomNumbers**

1. **CREATE A NEW PROJECT**

   Create a Windows application named RandomNumbers.
CREATE THE INTERFACE

Use the table below for setting object properties.

<table>
<thead>
<tr>
<th>Object</th>
<th>(Name)</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label1</td>
<td>lblRandNum1</td>
<td>empty</td>
</tr>
<tr>
<td>Label2</td>
<td>lblRandNum2</td>
<td>empty</td>
</tr>
<tr>
<td>Label3</td>
<td>lblRandNum3</td>
<td>empty</td>
</tr>
<tr>
<td>Label4</td>
<td>lblRandNum4</td>
<td>empty</td>
</tr>
<tr>
<td>Label5</td>
<td>lblRandNum5</td>
<td>empty</td>
</tr>
<tr>
<td>Label6</td>
<td>lblRandNum6</td>
<td>empty</td>
</tr>
<tr>
<td>Button1</td>
<td>btnRandomNumbers</td>
<td>Random Numbers</td>
</tr>
</tbody>
</table>

WRITE THE APPLICATION CODE

a. Display the Code window.
b. Add comments that include your name, assignment, and today’s date.
c. Create a btnRandomNumbers_Click event procedure and then add the statements:
   ```vbnet
   Me.lblRandNum1.Text = Rnd()
   Me.lblRandNum2.Text = Rnd()
   Me.lblRandNum3.Text = Rnd()
   Me.lblRandNum4.Text = Rnd()
   Me.lblRandNum5.Text = Rnd()
   Me.lblRandNum6.Text = Rnd()
   ```

RUN THE APPLICATION

a. Save the modified RandomNumbers project.
b. Run the application. Click Random Numbers. Write down on a piece of paper the six numbers generated.
c. Click Random Numbers again. Note that the numbers are the same as before.
d. Close the RandomNumbers application.

ADD THE RANDOMIZE() STATEMENT

a. In the btnRandomNumbers_Click event procedure, add a Randomize() statement:
   ```vbnet
   Randomize()
   Me.lblRandNum1.Text = Rnd()
   Me.lblRandNum2.Text = Rnd()
   Me.lblRandNum3.Text = Rnd()
   Me.lblRandNum4.Text = Rnd()
   Me.lblRandNum5.Text = Rnd()
   Me.lblRandNum6.Text = Rnd()
   ```
b. Save the modified RandomNumbers project and then run the application. Click Random Numbers a few times. Note that the numbers now vary.
GENERATE INTEGERS IN A RANGE

Modify the btnRandomNumbers_Click event procedure to generate random integers between 10 and less than 31, similar to:

```
Randomize()
Me.lblRandNum1.Text = Int(21 * Rnd() + 10)
Me.lblRandNum2.Text = Int(21 * Rnd() + 10)
Me.lblRandNum3.Text = Int(21 * Rnd() + 10)
Me.lblRandNum4.Text = Int(21 * Rnd() + 10)
Me.lblRandNum5.Text = Int(21 * Rnd() + 10)
Me.lblRandNum6.Text = Int(21 * Rnd() + 10)
```

RUN THE APPLICATION

Save the modified RandomNumbers project and then run the application. Click Random Numbers a few times.

PRINT THE CODE AND THEN CLOSE THE PROJECT

---

Algorithms

Programs are created to solve problems. However, problems of any complexity require outlining, or designing, a solution before typing source code. One method of designing a solution is to create an algorithm. An **algorithm** is a set of steps that outline how to solve a problem. There are various methods for implementing an algorithm. For example, an algorithm written in plain English for the NumberGuess application:

1. Determine a secret number.
2. Get a number from the player.
3. Compare the player’s number with the secret number.
4. If the player’s number is the same as the secret number go to step 5, otherwise tell the player if the number entered was too low or too high and then go back to step 2.
5. Display a message telling the player the secret number was guessed.

**pseudocode** Another way to implement an algorithm is to write it in *pseudocode*, which is a mix of English and program code. For example, the NumberGuess algorithm in pseudocode:

```
Sub btnCheckGuess_Click()
    secretNumber = 7
    Get guess from text box
    If guess = secretNumber Then
        Display “You guessed it!”
    ElseIf guess < secretNumber Then
        Display “Too low.”
    Else
        Display “Too high.”
    End If
End Sub
```
Creating an algorithm allows a programmer to think through a program before actually typing code. This helps a programmer focus on the overall structure of a program and may reduce errors in logic.

Static Variables

As discussed in Chapter 3, the scope of a variable is the set of statements that can access the variable. In addition to scope, variables have a lifetime in which they exist in memory. The lifetime of a local variable is the duration of the procedure in which it was declared. A global variable's lifetime is the duration of the program.

The lifetime of a local variable can be extended by using a Static declaration statement, which takes the form:

```
Static variableName As type = initialValue
```

A static variable is declared using the keyword Static instead of Dim. It should also be explicitly initialized in the declaration. A static variable's scope is local to the procedure in which it is declared, but its lifetime is the duration of the program. When either a static or global variable can be used, the static variable is a better choice because it is good programming style to keep the scope of a variable as narrow as possible.

Static variables are necessary in event procedures with variables that should be retained in memory throughout program execution. A variable declared in a Click event procedure is redeclared and reinitialized each time the Click event occurs unless the variable is declared as static. For example, the following declaration assigns a random number to a static variable. Unless the variable is assigned a new value, this value is retained throughout program execution:

```
Static secretNumber As Integer = Int(50*Rnd()+1)
```

Review: GuessingGame – part 1 of 4

GuessingGame is a game that can be played again and again by the same user because a new random number between 1 and 50 is generated for the secret number each time the application is started. The algorithm for Guessing Game is:

1. Generate a random secret number between 1 and 50.
2. Get a number from the player.
3. Compare the player's number with the secret number.
4. If the player's number equals the secret number go to step 5, otherwise tell the player if the number entered was too low or too high and then go back to step 2.
5. Display a message telling the player the secret number was guessed.

The GuessingGame algorithm in pseudocode:

```
Sub btnCheckGuess_Click()
    Generate a random number between 1 and 50 once when program starts
    Get guess from text box
    If guess = secret number Then
        Display “You guessed it!”
    ElseIf guess < secret number Then
```
Display “Too low.”
Else
Display “Too high.”
End If
End Sub

CREATE A NEW PROJECT
Create a Windows application named GuessingGame.

CREATE THE INTERFACE
Use the table below for setting properties.

<table>
<thead>
<tr>
<th>Object</th>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td>lblPrompt</td>
<td>Guessing Game</td>
</tr>
<tr>
<td>Label1</td>
<td>txtPlayerGuess</td>
<td>Enter a guess between 1 and 50: empty</td>
</tr>
<tr>
<td>TextBox1</td>
<td>lblMessage</td>
<td>empty</td>
</tr>
<tr>
<td>Button1</td>
<td>btnCheckGuess</td>
<td>Check Guess</td>
</tr>
</tbody>
</table>

WRITE THE APPLICATION CODE
a. Display the Code window.
b. Add comments that include your name, assignment, and today’s date.
c. Create two global variables and then create a btnCheckGuess_Click event procedure with the statements:

```vbnet
Public Class Form1
    Const MIN As Integer = 1
    Const MAX As Integer = 50

    Private Sub btnCheckGuess_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnCheckGuess.Click
        Randomize()
        Dim secretNumber As Integer = Int((MAX - MIN + 1) * Rnd()) + MIN
        Dim guess As Integer
        guess = Val(txtPlayerGuess.Text)
        If guess = secretNumber Then
            lblMessage.Text = “You guessed it!”
        ElseIf guess < secretNumber Then
            lblMessage.Text = “Too low.”
        ElseIf guess > secretNumber Then
            lblMessage.Text = “Too high.”
        End If
    End Sub
End Class
```

d. Create a txtPlayerGuess_TextChanged event procedure that assigns Nothing to the lblMessage label.
RUN THE APPLICATION

Save the modified GuessingGame application and then play the game a few times to test it.

PRINT THE CODE AND THEN CLOSE THE PROJECT

### Compound Boolean Expressions

**logical operators**

Conditions with complex criteria are formed using the *logical operators* `And` and `Or`. `And` is used to form an expression that evaluates to `True` only when both operands are true. An expression formed with `Or` evaluates to `True` when either operand is true. For example, the `If` condition is an expression formed with `Or`:

```vbnet
If guess < 1 Or guess > 50 Then 'invalid guess
    Me.lblMessage.Text = "Invalid guess."
ElseIf guess = secretNumber Then 'correct guess
    Me.lblMessage.Text = "You guessed it!"
...```

When `guess` is either less than 1 or greater than 50, “Invalid guess.” is displayed. The condition in the `If...Then...ElseIf` statement is called a *compound Boolean expression* because more than one Boolean expression determines whether the condition is true or false.

How a compound Boolean expression evaluates with `And` and `Or` operators can be shown in a truth tables. A *truth table* shows the possible outcomes of compound Boolean expressions:

<table>
<thead>
<tr>
<th>Exp1</th>
<th>Exp2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exp1</th>
<th>Exp2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

As another example, consider an application that computes a discount depending on the quantity and type of purchase:

```vbnet
If itemNum = 220 And quantity > 50 Then
    discount = 1 '1.00 discount
End If```

This `If...Then` statement executes the `discount = 1` statement if both `itemNum` is 220 and `quantity` is greater than 50.

A third logical operator is `Not`. An expression including `Not` is evaluated according to the following truth table:

<table>
<thead>
<tr>
<th>Exp</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
For example, the following statements change the label because `itemNum` is not 220:

```vbnet
If Not itemNum = 220 Then
    Me.lblMessage.Text = "No discount given."
End If
```

In the order of operations, `Not` is evaluated before `And` or `Or` is evaluated last. For example, the expression `Not 5 < 6 Or 2 > 4 And 3 < 6` evaluates to `False` because `Not 5 < 6` is performed first, then `2 > 4` and `3 < 6`, and then `False Or False`. Operator precedence can be changed by using parentheses.

### Review: RockPaperScissors – part 1 of 4

Rock Paper Scissors is a popular game used for decision making between two individuals. The rules of the game are Rock dulls Scissors, Scissors cut Paper, and Paper covers Rock. In this computer-game version, the user plays against the computer. The RockPaperScissors algorithm is:

1. The player selects either Rock, Paper, or Scissors.
2. A random number between 1 and 3 is generated to represent the computer’s choice. A 1 corresponds to Rock, a 2 to Paper, and a 3 to Scissors.
3. Compare the player’s choice to the computer’s choice.
4. Display an appropriate message.

The RockPaperScissors algorithm in pseudocode:

```vbnet
Sub btnPlay_Click()
    Get user’s choice from radio buttons
    Generate a random number between 1 and 3
    If user = radRock and comp = Rock Then
draw
    ElseIf user = radRock and comp = Paper Then
        computer wins
    ElseIf user = radRock and comp = Scissors Then
        user wins
    End If
    If user = radPaper and comp = Rock Then
        user wins
    ElseIf user = radPaper and comp = Paper Then
        draw
    ElseIf user = radPaper and comp = Scissors Then
        computer wins
    End If
    If user = radScissors and comp = Rock Then
        computer wins
    ElseIf user = radScissors and comp = Paper Then
        user wins
    ElseIf user = radScissors and comp = Scissors Then
        draw
    End If
End Sub
```

**CREATE A NEW PROJECT**

Create a Windows application named RockPaperScissors.
CREATE THE INTERFACE

Use the table below for setting properties.

<table>
<thead>
<tr>
<th>Object</th>
<th>(Name)</th>
<th>Text</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GroupBox1</td>
<td>grpThrows</td>
<td>Choose Your Throw</td>
<td></td>
</tr>
<tr>
<td>RadioButton1</td>
<td>radRock</td>
<td>Rock</td>
<td>False</td>
</tr>
<tr>
<td>RadioButton2</td>
<td>radPaper</td>
<td>Paper</td>
<td>False</td>
</tr>
<tr>
<td>RadioButton3</td>
<td>radScissors</td>
<td>Scissors</td>
<td>False</td>
</tr>
<tr>
<td>Label1</td>
<td>lblWinner</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>Button1</td>
<td>btnGo</td>
<td>Go!</td>
<td></td>
</tr>
</tbody>
</table>

WRITE THE APPLICATION CODE

a. Display the Code window.
b. Add comments that include your name, assignment, and today’s date.
c. Create a btnGo_Click event procedure and then add the statements:

```vbnet
Const ROCK As Integer = 1
Const PAPER As Integer = 2
Const SCISSORS As Integer = 3
Dim computerThrow As Integer

'Generate computer throw
Randomize()
computerThrow = Int(3 * Rnd() + 1)

If Me.radRock.Checked And computerThrow = ROCK Then
    Me.lblWinner.Text = "Computer throws Rock. It's a Draw!" 'Rock Rock
ElseIf Me.radRock.Checked And computerThrow = PAPER Then
ElseIf Me.radRock.Checked And computerThrow = SCISSORS Then
    Me.lblWinner.Text = "Computer throws Scissors. You win!" 'Rock Scissors
End If

If Me.radPaper.Checked And computerThrow = ROCK Then
    Me.lblWinner.Text = "Computer throws Rock. You win!" 'Paper Rock
ElseIf Me.radPaper.Checked And computerThrow = PAPER Then
    Me.lblWinner.Text = "Computer throws Paper. It's a Draw!" 'Paper Paper
ElseIf Me.radPaper.Checked And computerThrow = SCISSORS Then
End If

If Me.radScissors.Checked And computerThrow = ROCK Then
    Me.lblWinner.Text = "Computer throws Rock. Computer wins!" 'Scissors Rock
ElseIf Me.radScissors.Checked And computerThrow = PAPER Then
ElseIf Me.radScissors.Checked And computerThrow = SCISSORS Then
    Me.lblWinner.Text = "Computer throws Scissors. It's a Draw!" 'Scissors Scissors
End If
```
RUN THE APPLICATION
Save the modified RockPaperScissors application and then play the game a few times to test it. Close the application.

PRINT THE CODE

Review: RockPaperScissors – part 2 of 4
Modify the RockPaperScissors application to use nested If...Then...ElseIf statements to determine the outcome of game.

Review: RockPaperScissors – part 3 of 4
Modify the RockPaperScissors application to use a Select...Case statement and If...Then...ElseIf statements to determine the outcome of game. Hint: Use the computerThrow value as the expression in the Select...Case statement.

Displaying a Message Box
A message box is a predefined dialog box that displays a message for the user. A message can be displayed to alert the user to invalid data or as a reminder of options required for an application to continue. For example, the GuessingGame application could be modified to alert the user to a guess that is out of range:

The MessageBox class includes a Show() method for displaying a message box and is used in a statement that takes the form:

```
MessageBox.Show(message)
```

message is a variable, constant, or a string literal. For example, the If...Then...ElseIf statement alerts the user to a number outside the allowed range:

```
If guess < MIN Or guess > MAX Then
    MessageBox.Show("Guess out of range.")
ElseIf guess = secretNumber Then
    Me.lblMessage.Text = "You guessed it!"
ElseIf guess < secretNumber Then
    Me.lblMessage.Text = "Too low."
Else
    Me.lblMessage.Text = "Too high."
End If
```
Review: GuessingGame – part 2 of 4

Modify the GuessingGame application to display a message box with an appropriate message if the user’s guess is less than the minimum allowed number or greater than maximum.

Review: TestGrade – part 5 of 5

Modify the TestGrade application so that “Invalid grade.” is displayed in a message box if the grade entered is less than 0 or greater than 100.

Counter Variables

Many algorithms involve counting. Information such as the number of times a user enters a guess require counting each guess. Calculations such as averaging a set of numbers require counting because the sum of the numbers must be divided by the count of the numbers. Applications written for algorithms that involve counting use a counter variable for storing a number that is incremented by a constant value.

Counters are useful for keeping track of the number of times a user clicks a button, enters a guess, or types a password. The statement for incrementing a counter, often called updating a counter, takes the form:

```vbnet
counter = counter + constant
```

counter is the numeric variable that is updated. constant is the number that is added to the current value of counter. In an assignment statement the expression on the right side of the equal sign is evaluated first and then that value given to the variable on the left. This makes it possible to use the current value of counter in the expression itself. For example, the following statement updates the counter numTries by 1:

```vbnet
numTries = numTries + 1
```

Each time the statement is executed, 1 is added to the current value of numTries and then this new value is assigned to numTries.

Counters are used often in programming. Therefore, Visual Basic has additional assignment operators just for updating counters. These operators perform an operation before making an assignment. For example, the += allows the statement above to be written as:

```vbnet
numTries += 1
```

Counters sometimes count backward. For these counters, the -= operator can be used.

A counter should be initialized when it is declared and then updated by an unchanging amount. A counter in an event procedure should be declared as a Static variable so that it is initialized only once.
Review: RockPaperScissors – part 4 of 4

Modify the RockPaperScissors application to include three counters that maintain the number of wins by the player, the number of wins by the computer, and the number of draws. The scores should be updated and displayed in labels at the end of each game.

The CheckBox Control

Check boxes allow the user to select options. Unlike radio buttons, more than one check box can be selected at a time. For example, in the MorningToDo application, CheckBox objects give the user options:

![CheckBox Control Example](image)

The CheckBox control has the properties:

- **(Name)** identifies a control for the programmer. CheckBox object names should begin with `chk`.
- **Text** is the text displayed next to the box.
- **Checked** can be set to either True or False to display the check box with or without a check, respectively.

Related check boxes are sometimes placed together in a GroupBox object. As with radio buttons, a group box should be added to the form before adding check boxes.

An `If...Then` statement can be used in a program to determine if a check box is selected or cleared. For example, the following statement displays a message that depends on the state of the check box:

```vbnet
If Me.chkLunch.Checked Then 'check box selected
    MessageBox.Show("Don't forget your bottled water!")
Else 'check box cleared
    MessageBox.Show("Take lunch money!")
End If
```

A Click event procedure is sometimes coded for a check box. This procedure executes when a check box is clicked and usually includes code to determine the state of the check box and then perform actions depending on whether the check box was selected or cleared.
Implicit Line Continuation

A statement typically fits on one line, but can be continued onto the next line using a line-continuation sequence, which consists of a space followed by an underscore character (_), followed by a carriage return. For example, a condition placed onto two lines looks similar to:

```
If Not (Me.chkBed.Checked And Me.chkLunch.Checked_
    And Me.chkHomework.Checked And Me.chkTeeth.Checked) Then
...
```

In many cases, you can continue a statement on the next consecutive line without using the underscore character (_). Syntax elements that implicitly continue the statement on the next line of code include:

- after a comma (,)
- after an open parenthesis (() or before a closing parenthesis ()
- after an open curly brace ({) or before a closing curly brace (})
- after assignment operators (=, &, =, +=, -=, *=, /=, \=, ^=)
- after binary operators (+, -, /, *, Mod, <>, <, >, <=, >=, And, Or)

Review: MorningToDo

1. CREATE A NEW PROJECT
   Create a Windows application named MorningToDo.

2. CREATE THE INTERFACE
   Use the table below for setting properties.

<table>
<thead>
<tr>
<th>Object</th>
<th>(Name)</th>
<th>Text</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td></td>
<td>Morning To Do</td>
<td></td>
</tr>
<tr>
<td>CheckBox1</td>
<td>chkBed</td>
<td>Make bed</td>
<td>False</td>
</tr>
<tr>
<td>CheckBox2</td>
<td>chkLunch</td>
<td>Pack lunch</td>
<td>False</td>
</tr>
<tr>
<td>CheckBox3</td>
<td>chkHomework</td>
<td>Gather homework</td>
<td>False</td>
</tr>
<tr>
<td>CheckBox4</td>
<td>chkTeeth</td>
<td>Brush teeth</td>
<td>False</td>
</tr>
<tr>
<td>Button1</td>
<td>btnAllDone</td>
<td>All Done</td>
<td></td>
</tr>
</tbody>
</table>

3. WRITE THE APPLICATION CODE
   a. Display the Code window.
   b. Add comments that include your name, assignment, and today’s date.
   c. Create a chkLunch_Click event procedure and then add the statements:

   ```
   If Me.chkLunch.Checked Then
       MessageBox.Show("Don’t forget bottled water!")
   End If
   ```
Chapter 4 Controlling Program Flow with Decision Structures

d. Create a btnAllDone_Click event procedure and then add the statements:

```vbnet
If Not (Me.chkBed.Checked And Me.chkLunch.Checked And Me.chkHomework.Checked And Me.chkTeeth.Checked) Then
    MessageBox.Show("Did you forget something?")
Else
    Application.Exit()
End If
```

RUN THE APPLICATION

a. Save the modified MorningToDo application and then run it. Select the Pack Lunch check box and note the message box.
b. Select all the check boxes and then select All Done! The application ends.
c. Run the application again. Select only two or three of the check boxes and then select All Done! A message box is displayed. Select the remaining check boxes and then select All Done!

PRINT THE CODE AND THEN CLOSE THE PROJECT

Case Study

In this case study a PizzaOrder application will be created.

PizzaOrder Specification

PizzaOrder allows a user to select a pizza size and toppings. Topping choices include pepperoni, mushrooms, onions, and hot peppers. Once the order is placed, an order number and the pizza price are displayed. A new order clears the current order information and allows the next order to be generated. The order number should automatically increment with each new order. The following prices and toppings prices should be used:

- Regular: $6.00
- Large: $10.00
- one topping: $1.00 additional
- two toppings: $1.75 additional
- three toppings: $2.50 additional
- four toppings: $3.25 additional

PizzaOrder Design

For each pizza order there can be only one size, but many toppings. Therefore, radio buttons should be used to select the size and check boxes should be used to select the toppings:
The code design should start with an algorithm:

1. Increment order number.
2. Determine toppings price based on the selected number of toppings.
3. Determine pizza price based on the selected size and then add toppings price.
4. Display pizza price and order number.
5. When a new order is started, select the Regular radio button and clear the check boxes and labels.

The algorithm is implemented with the following pseudocode:

```vbnet
Sub btnPlaceOrder_Click()
    Const REGULAR As Decimal = 6
    Const LARGE As Decimal = 10
    Const ONE_TOPPING As Decimal = 1
    Const TWO_TOPPINGS As Decimal = 1.75
    Const THREE_TOPPINGS As Decimal = 2.5
    Const FOUR_TOPPINGS As Decimal = 3.25
    Static orderNumber As Integer = 0
    Dim numToppings As Integer
    Dim toppingsPrice As Decimal
    Dim pizzaPrice As Decimal
    Increment orderNumber
    For each topping check box selected, increment numToppings
    Use numToppings to select ONE_TOPPING, TWO_TOPPINGS, THREE_TOPPINGS, or FOUR_TOPPINGS
    If radLarge.Checked Then
        pizzaPrice = LARGE + toppingsPrice
    Else
        pizzaPrice = REGULAR + toppingsPrice
    End If
    lblOrder = “Order Number:”
    lblOrderNumber.Text = OrderNumber
    lblPrice = “Price: $”
    lblPizzaPrice.Text = PizzaPrice
End Sub

Sub btnNewOrder_Click()
    Clear check boxes
    Select the Regular radio button
    Clear labels
End Sub
```
PizzaOrder Coding

The interface and code for this Case Study are:

<table>
<thead>
<tr>
<th>Object</th>
<th>(Name)</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td></td>
<td>Pizza Order</td>
</tr>
<tr>
<td>GroupBox1</td>
<td>grpPizzaSize</td>
<td>Select pizza size</td>
</tr>
<tr>
<td>RadioButton1</td>
<td>radRegular</td>
<td>Regular</td>
</tr>
<tr>
<td>RadioButton2</td>
<td>radLarge</td>
<td>Large</td>
</tr>
<tr>
<td>GroupBox2</td>
<td>grpToppings</td>
<td>Select toppings</td>
</tr>
<tr>
<td>CheckBox1</td>
<td>chkPepperoni</td>
<td>Pepperoni</td>
</tr>
<tr>
<td>CheckBox2</td>
<td>chkMushrooms</td>
<td>Mushrooms</td>
</tr>
<tr>
<td>CheckBox3</td>
<td>chkOnions</td>
<td>Onions</td>
</tr>
<tr>
<td>CheckBox4</td>
<td>chkHotPeppers</td>
<td>Hot Peppers</td>
</tr>
<tr>
<td>Button1</td>
<td>btnPlaceOrder</td>
<td>Place Order</td>
</tr>
<tr>
<td>Button2</td>
<td>btnNewOrder</td>
<td>New Order</td>
</tr>
<tr>
<td>Label1</td>
<td>lblOrder</td>
<td>empty</td>
</tr>
<tr>
<td>Label2</td>
<td>lblOrderNumber</td>
<td>empty</td>
</tr>
<tr>
<td>Label3</td>
<td>lblPrice</td>
<td>empty</td>
</tr>
<tr>
<td>Label4</td>
<td>lblPizzaPrice</td>
<td>empty</td>
</tr>
</tbody>
</table>

The Checked property of the radRegular radio button must be set to True.

Public Class Form1

Private Sub btnPlaceOrder_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnPlaceOrder.Click
    Const REGULAR As Decimal = 6 ' $6
    Const LARGE As Decimal = 10 ' $10
    Const ONE_TOPPING As Decimal = 1 ' $1
    Const TWO_TOPPINGS As Decimal = 1.75 ' $1.75
    Const THREE_TOPPINGS As Decimal = 2.5 ' $2.50
    Const FOUR_TOPPINGS As Decimal = 3.25 ' $3.25
    Dim numToppings As Integer = 0
    Dim toppingsPrice As Decimal = 0
    Dim orderNumber As Integer = 0

    ' Increment order number
    orderNumber += 1

    ' Count selected toppings

Chapter 4 Controlling Program Flow with Decision Structures
If Me.chkHotPeppers.Checked = True Then
   numToppings += 1
End If
If Me.chkMushrooms.Checked = True Then
   numToppings += 1
End If
If Me.chkOnions.Checked = True Then
   numToppings += 1
End If
If Me.chkPepperoni.Checked = True Then
   numToppings += 1
End If

'Determine toppings price
Select Case numToppings
Case 1
   toppingsPrice = ONE _ TOPPING
Case 2
   toppingsPrice = TWO _ TOPPINGS
Case 3
   toppingsPrice = THREE _ TOPPINGS
Case 4
   toppingsPrice = FOUR _ TOPPINGS
End Select

'Determine pizza price
If Me.radLarge.Checked Then  'large pizza
   pizzaPrice = LARGE + toppingsPrice
Else
   'regular pizza
   pizzaPrice = REGULAR + toppingsPrice
End If

'Display order number and pizza price
Me.lblOrder.Text = "Order Number:"  
Me.lblOrderNumber.Text = orderNumber  
Me.lblPrice.Text = "Price: $"  
Me.lblPizzaPrice.Text = pizzaPrice
End Sub

Private Sub btnNewOrder_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnNewOrder.Click
   Me.radLarge.Checked = False
   Me.radRegular.Checked = True
   Me.chkHotPeppers.Checked = False
   Me.chkMushrooms.Checked = False
   Me.chkOnions.Checked = False
   Me.chkPepperoni.Checked = False
   Me.lblOrder.Text = Nothing
   Me.lblOrderNumber.Text = Nothing
   Me.lblPrice.Text = Nothing
   Me.lblPizzaPrice.Text = Nothing
End Sub

Running PizzaOrder, selecting options, and then clicking Place Order displays:
This case study should be tested by generating several different pizza orders and checking the price displayed by hand.

**Review: PizzaOrder**

Modify the PizzaOrder case study to include Pickup and Delivery radio buttons in a new group box. If Delivery is selected, $1.50 should be added to the total price of the pizza. Pickup should add nothing to the total price of the pizza and should be selected when the program starts and when a new order is started.

**Chapter Summary**

Conditional control structures, also called decision structures, include the **If...Then**, **If...Then...Else**, **If...Then...ElseIf**, **Select...Case**, and **Select...Case Is** statements. Each of these structures evaluates a condition to determine program flow. The condition in the **If...Then** statements is a Boolean expression that evaluates to true or false. The **If...Then...ElseIf** statement is used to decide among three, four, or more actions. The **Select...Case** statements are also used to decide among many actions.

A Boolean expression may be a Boolean variable or an expression formed using relational operators (\(=\), \(<\), \(<=\), \(>\), \(>=\), \(<>\)). A compound Boolean expression uses logical operators (\(\&\&\), \(||\), \(\neg\)).

The Rnd() function generates a random number greater than or equal to 0 and less than 1. The Int() function is used to return the integer portion of a number. The Randomize() statement is used to initialize the Rnd() function so that different random numbers are generated from program run to program run.

An algorithm is a set of steps that tell how to solve a problem. An algorithm refined using both English and program code is called pseudocode. Creating an algorithm before using the computer helps reduce logic errors.

Static variables have a local scope but a lifetime the duration of the program. Static variables are necessary in event procedures with variables that should be retained in memory throughout a program run.
A message box is used to provide information to the user, such as when invalid data has been entered. The MessageBox class includes a Show() method that displays a predefined dialog box.

A counter is a variable storing a number that is incremented by a constant value. A counter in an event procedure should be declared as a Static variable so that it is initialized only once.

A check box is an object that allows the user to select one or more options from a set of options. A Click event is sometimes coded for a check box. The procedure should include an If...Then statement to determine if the check box has been selected or cleared. Related check boxes are sometimes placed together in a group box.

The line-continuation character (_) is used to divide a statement over two or more lines. However, in many cases, such as after a comma or after an assignment operator, you can continue a statement on the next consecutive line without using the underscore character (_).

The code conventions introduced in this chapter are:

- The statements of an If...Then statement should be indented.
- The statements of an If...Then...Else statement should be indented.
- The statements of an If...Then...ElseIf statement should be indented.
- Nested statements should be indented.
- The Case statements of a Select...Case statement should be indented.
- Choose a static variable over a global variable when possible because the scope of the static variable can be kept more narrow.
- Use implicit line continuation syntax to divide long statements into two or more lines.
Vocabulary

**Algorithm**  A set of steps that outline how to solve a problem.

**Compound Boolean expression**  More than one Boolean expression determines whether the condition is True or False.

**Conditional control structure**  See Decision structure.

**Counter**  A variable used to store a value that is updated by a constant value.

**Decision structure**  A statement that uses a condition to determine which set of statements to execute.

**Lifetime**  The duration in which a declared variable exists in memory.

**Logical operators**  Operators (and, or, and not) that may be used to form a Boolean expression.

**Message box**  A predefined dialog box that displays a message for the user.

**Nested statements**  One or more statements within a statement.

**Pseudocode**  An algorithm written in both English and program code.

**Pseudorandom**  Not truly random, but like random.

**Relational operators**  Operators (=, <, <=, >, >=, and <> that can be used to form a Boolean expression.

**Static variable**  A variable with a local scope but a lifetime the duration of the program.

**Truth table**  Shows the possible outcomes of compound Boolean expressions.

**Update**  To increment a counter variable.
Visual Basic

_(underscore)_ The line-continuation character.

`+=` Assignment operator that adds the value on the right of the statement to the current value of the variable on the left and then updates the variable to store the new value.

`-=` Assignment operator that subtracts the value on the right of the statement from the current value of the variable on the left and then updates the variable to store the new value.

`(equal to)` Relational operator used to determine if one value is equal to another.

`(less than)` Relational operator used to determine if one value is less than another.

`(less than or equal to)` Relational operator used to determine if one value is less than or equal to another.

`>(greater than)` Relational operator used to determine if one value is greater than another.

`>=`(greater than or equal to) Relational operator used to determine if one value is greater than or equal to another.

`<>`(not equal to) Relational operator used to determine if one value is not equal to another.

`And` Logical operator used to form a Boolean expression. An expression formed using `And` is `True` only when the expressions it joins are all `True`.

`Not` Logical operator used to form a Boolean expression. An expression formed using `Not` is `True` only when the expression it is used with is `False`.

`Or` Logical operator used to form a Boolean expression. An expression formed using `Or` is `True` when any of the expressions it joins are `True`.

`Randomize()` Statement used to initialize the Rnd() function so that different random numbers are generated from run to run.

`Rnd()` A function used to generate a random number greater than or equal to 0 and less than 1.

`Select...Case` Statement that executes code depending on the result of an expression.

`Select...Case Is` Statement that executes code depending on a comparison of a range of values to the result of an expression.

`Static` Statement used to declare a static variable.

---

CheckBox control Used to add a CheckBox control class object to a form. Properties include (Name), Text, and Checked. Events include Click.

`If...Then` Statement that executes code when a condition is `True`.

`If...Then...Else` Statement that executes code in the Else clause when a condition is `False`.

`If...Then...ElseIf` Statement that is used to decide among three or more actions.

`Int()` A function that returns the integer portion of a number without rounding.

`MessageBox` class Used to display a predefined dialog box that displays a message and an OK button. Methods include Show().
Chapter 4 Controlling Program Flow with Decision Structures

Critical Thinking

1. Assuming the comment is correct, determine the logic error in the following statement:

   ```vba
   If grade > 90 Then
       'Display A for grade greater than or equal to 90
       Me.lblGrade.Text = "You have an A"
   End If
   ```

2. What is displayed in the label after the following statement executes? Does the label assignment reflect what was intended? If not, how should the statement be rewritten to produce the intended result?

   ```vba
   Dim score As Integer = 25
   If score >= 100 Then
       Me.lblMessage.Text = "You won!"
   ElseIf score < 100 Then
       Me.lblMessage.Text = "Good try."
   ElseIf score < 50 Then
       Me.lblMessage.Text = "Practice more."
   End If
   ```

3. Check boxes, radio buttons, and text boxes all accept user input.
   a) List the differences in the way the three accept input.
   b) Give an example of how each should be used in an application.

4. Given the statements

   ```vba
   Dim quantity As Integer = 20
   Dim price As Double = 5
   ```

   determine the value, true or false, for each of the following expressions:

   a) quantity > 10 And price > 5
   b) quantity = 15 Or price = 5
   c) quantity >= 20 And price >=2 And quantity * price >= 40
   d) Not price = 5
   e) quantity < 100 Or price > 4 And Not quantity = 20

5. Write an appropriate decision statement for each of the following:

   a) Display “Great Job” in a label named lblMessage if grade is 90 or above.
   b) Display “High Scorer” in a label named lblHigh for totalPoints between 100 and 200, inclusive.
   c) Display “Number must be less than 100.” in a message box if the value in txtGrade is greater than 100.

6. a) Which is the appropriate word for the first blank below, odd or even? Which is the appropriate word for the second blank?

   ```vba
   If number Mod 2 = 0 Then
       MessageBox.Show(“Your number is ______”) 
   Else
       MessageBox.Show(“Your number is ______”) 
   End If
   ```

   b) Rewrite the `If...Then...Else` statement from part (a) as a `Select...Case` statement.

7. List the errors in the statement below and then rewrite the statement so that it will execute as expected:

   ```vba
   If 50 <= numTickets <= 100 And Me.radStu.Checked
       MessageBox.Show = (“Both discounts.”) 
   ElseIf 50 <= numTickets <= 100 
       MessageBox.Show = (“Volume discount.”) 
   ElseIf Me.radStu.Checked 
       MessageBox.Show = (“Student discount.”) 
   Case Else 
       MessageBox.Show = (“No discount.”) 
   End If
   ```

8. Rewrite the following statement so that it does not include a nested `If...Then` statement:

   ```vba
   If Me.chkValue1.Checked Then
     If Me.chkValue2.Checked Then 
       MessageBox.Show(“Both applied.”)
   End If
   ```

9. Assume txtTest1, txtTest2, and txtTest3 contain numeric values. Write an `If...Then...Else` statement that displays in a label the average of the three numbers only if all of the numbers are between 0 and 100, otherwise a message box with an appropriate message should be displayed and the text boxes cleared.

10. Assume txtNum1 and txtNum2 contain numeric values. Write an `If...Then...ElseIf` statement that displays a message box with one of the following messages as appropriate:

    ```vba
    First number is larger
    Second number is larger
    Both numbers are equal
    ```

11. Write a statement that generates a random whole number between 5 and 50, inclusive.
12. a) List the errors in the statement below and then rewrite the statement so that it will execute as expected:

```
Select Case num
  Case 2 Or 3, num > 10
    MessageBox.Show("1st Case")
  Case 20 <= num < 30
    MessageBox.Show("2nd Case")
End Case
```

b) Rewrite the `Select...Case` statement in part (a) using an `If...Then...ElseIf` statement.

13. Assume `txtMonth` contains all uppercase text that is a month of the year, for example, SEPTEMBER. Another text box, `txtYear`, contains the year. Write a `Select...Case` statement that displays in a message box the number of days in the month entered. Hint: An efficient statement does not require 12 case values. The days in February can be determined by using the following pseudocode:

```
If year Mod 4 <> 0 Then
  use 28 days for February
ElseIf year Mod 400 = 0 Then
  use 29 days for February
ElseIf year Mod 100 = 0 Then
  use 28 days for February
Else
  use 29 for days in February
End If
```

14. Write a `btnPurchase_Click` event procedure that calculates the cost of tickets and gives free tickets on every 100th purchase. The `txtNumTickets` text box contains the number of tickets for a purchase and each ticket price is $8.00. A counter variable should be updated by one each time `Purchase` is clicked. On the 100th purchase, a message box should display “Congratulations, the tickets are free!” The counter should then be reset to zero. If the purchase is not the 100th, a message box should display the cost of the tickets. Use appropriate constants and variables.

15. Write a `btnMessage_Click` event procedure that displays one of the messages below in a message box:

```
You win $100    2% of the time
You win $10    10% of the time
You win $1    50% of the time
Thanks for trying. The rest of the time.
```

`Hint:` Use a random number between 1 and 100 and a `Select...Case` to determine the message to display.

16. Determine if each of the following statements is true or false. If false, explain why.

a) The condition of an `If...Then` statement is a Boolean expression
b) A decision structure must have an `Else` clause.
c) It is good programming style to line up the `If`, the `Else`, and the `End If` in a decision structure, and to indent the lines in between.
d) The `Select...Case` statement must have the `Case Else` clause.
e) The `Select...Case` statement can only be used if you have more than two cases.
f) Using `Rnd()` without including `Randomize()` will produce a run-time error.
g) Numbers generated by the statement `Rnd()` are integers.
h) Algorithms are designed after the source code is typed.
i) The value of local variables are always retained in memory for the duration of a program execution.
j) A compound Boolean expression uses more than one Boolean expression to determine whether a condition is true or false.
k) In a logical `And` expression, both operands must be true for the expression to evaluate to true.
l) In a logical expression, `Or` is evaluated before `Not`.
m) Message boxes can only be used in decision statements.
n) Counter variables are useful for keeping track of the number of times a specific event occurs.
o) `sum`, assigned as `sum = 1 + 2 + 3`, is a counter variable.
p) Only one check box can be selected at a time.
q) A Visual Basic statement must be typed in its entirety on a single line.
Exercises

Exercise 1  Number Of Digits
Create a Number Of Digits application that prompts the user for a number less than 100 and then when Check Number is clicked displays whether the number is one digit or two digits:

Exercise 2  Payroll
An employee should receive pay equal to time and a half for every hour worked over 40 hours.

a) Create a Payroll application that prompts the user for the number of hours worked and the hourly rate of pay and then calculates the gross weekly wages (before taxes) when Pay is clicked:

b) Modify the Payroll application so that there is an 18% deduction from gross pay, unless the employee is exempt. If an employee is exempt, “NO TAXES DEDUCTED” should be displayed in a message box and then the wages displayed. The application interface should look similar to the following for an employee that is not exempt:
Exercise 3 ——————————————————— PrintingPrices

Printing prices are typically based on the number of copies to be printed. For example:

- 0 - 499 copies $0.30 per copy
- 500 - 749 copies $0.28 per copy
- 750 - 999 copies $0.27 per copy
- 1000 copies or more $0.25 per copy

Create a PrintingPrices application that prompts the user for the number of copies to print and then when Price is clicked displays the price per copy and the total price:

Exercise 4 ——————————————————— PackageCheck

A delivery service does not accept packages heavier than 27 kilograms or larger than 0.1 cubic meters (100,000 cubic centimeters). Create a PackageCheck application that prompts the user for the weight of a package and its dimensions, and when Check Package is clicked displays an appropriate message if the package does not meet the requirements (e.g., too large, too heavy, or both):
Chapter 4 Controlling Program Flow with Decision Structures

Exercise 5  ComputerTroubleshooting

Create a ComputerTroubleshooting application that asks the user if the ailing computer beeps on startup and if the hard drive spins. If it beeps and the drive spins, have the application display “Contact tech support.” If it beeps and the drive doesn’t spin, have the application display “Check drive contacts.” If it doesn’t beep and the hard drive doesn’t spin, have the application display “Bring computer to repair center.” Finally, if it doesn’t beep and the hard drive spins, have the application display “Check the speaker connections.” The application interface should look similar to:

Exercise 6  CarModels

An auto company produced some models of cars that may be difficult to drive because the car wheels are not exactly round. Cars with model numbers 119, 179, 189 through 195, 221, and 780 have been found to have this defect. Create a CarModels application that prompts a customer for the model number of their car to find out if it is defective. When Evaluate is clicked, the message “Your car is not defective.” should be displayed if the user typed a model number without a defect. Otherwise, the message “Your car is defective. Please have it fixed.” should be displayed:

Exercise 7  Grades

Create a Grades application that allows the user to enter one letter grade (uppercase or lowercase) after another and continuously displays the number of students who passed (D or better) and the number who failed. The application interface should look similar to the following after entering 15 grades and clicking Enter Grade:
Exercise 8

PhoneBill

Create a PhoneBill application that determines a phone bill by prompting the user for calling options (call waiting, call forwarding, and caller ID). The monthly basic service charge is $25.00 and each additional calling option is $3.50. The application interface should look similar to the following after selecting options and clicking Calculate:

![PhoneBill application interface](image)

Exercise 9

Welcome

Many programs are password protected and require the user to enter a user ID and password to get access to the application. A welcome dialog box usually looks similar to:

![Welcome dialog box](image)

The password is kept secret by showing a special character, often an asterisk (*), in place of each letter typed in the text box. This can be specified from the Design window by typing * in the PasswordChar property of the text box object.

a) Create a Welcome application that prompts the user for an ID and a password. If the ID and password are correct, display a message box stating so and then end the application. If the ID is not correct display an “Incorrect ID.” message box and then clear the ID text box and allow the user to enter another ID. If the password is not correct display an “Incorrect password.” message box and then clear the Password text box and allow the user to enter another password. If the ID and password are both not correct display an “Incorrect ID and password.” message box and then clear both text boxes and allow the user to enter another ID and password. If the user has made three incorrect attempts then display a “Sorry, access denied.” message box and then end the application.

b) Modify the application to check for three different user IDs and their corresponding passwords.
Exercise 10  ————————————————————  MathTutor
Create a MathTutor application that displays math problems by randomly generating two numbers, 1 through 10, and an operator (\*, \+, -, /) and prompts the user for an answer. The application should check the answer and display a message, display the correct answer, and generate a new problem. The application interface should look similar to the following after typing a correct answer and clicking Check Answer:

![MathTutor Application Image]

Exercise 11  ————————————————————  SandwichOrder
Create a SandwichOrder application that creates a sandwich order by prompting the user for the size of the sandwich (small or large) and the fixings (lettuce, tomato, onion, mustard, mayonnaise, cheese). A small sandwich is $2.50 and a large sandwich is $4.00. Mustard and mayonnaise are free, lettuce and onion are $0.10 each, tomato is $0.25, and cheese is $0.50. The defaults should be a small sandwich with no fixings. The application interface should look similar to the following after selecting options and clicking Place Order:

![SandwichOrder Application Image]
Exercise 12  ❯  GuessingGame

The GuessingGame application created in this chapter would be better if the number of guesses the user took were displayed at the end of the game.

a) Modify the GuessingGame code to include a counter that keeps track of the number of guesses made by the user. Have the application display the total number of guesses in a message box after the user correctly guesses the secret number.

b) A binary search is a divide-and-conquer technique for efficiently searching a list of numbers that are sorted from lowest to highest. A strategy that incorporates the binary search technique can be used by the GuessingGame player when making guesses about the secret number:

1. Guess the number halfway between the lowest and highest numbers.
2. If the number guessed matches the secret number, then the player wins.
3. If the number guessed is too high, then take the number guessed minus one and make this the highest number and go back to Step 1.
4. If the number guessed is too low, then take the number guessed plus one and make this the lowest number and go back to Step 1.

For example, assuming 15 is the random number generated in the GuessingGame application, the game would play out as follows when the player uses a divide-and-conquer technique:

<table>
<thead>
<tr>
<th>Current Low</th>
<th>Current High</th>
<th>Player Types</th>
<th>Message Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>26 (i.e., (1+50)/2=25.5)</td>
<td>Too high.</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>13 (i.e., (1+25)/2=13)</td>
<td>Too low.</td>
</tr>
<tr>
<td>14</td>
<td>25</td>
<td>20 (i.e., (14+25)/2=19.5)</td>
<td>Too high.</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>16 (i.e., (14+19)/2=16.5)</td>
<td>Too high.</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>14 (i.e., (14+15)/2=14.5)</td>
<td>Too low.</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15 (i.e., (15+15)/2=15)</td>
<td>You guessed it!</td>
</tr>
</tbody>
</table>

In another program run, assuming the random number generated is 20, the game would play out as follows using the same divide-and-conquer technique:

<table>
<thead>
<tr>
<th>Current Low</th>
<th>Current High</th>
<th>Player Types</th>
<th>Message Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>26 (i.e., (1+50)/2=25.5)</td>
<td>Too high.</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>13 (i.e., (1+25)/2=13)</td>
<td>Too low.</td>
</tr>
<tr>
<td>14</td>
<td>25</td>
<td>20 (i.e., (14+25)/2=19.5)</td>
<td>You guessed it!</td>
</tr>
</tbody>
</table>

When this approach is taken, it has been proven that a player will not be required to make more than \( \log_2 n \) guesses, in this case \( \log_2 50 \), or at most 6 guesses. Try this technique yourself. Explain in your own words why this works. Would this strategy be possible if hints were not given after each guess?
Exercise 13  ———— GuessTheBlocks
Create a GuessTheBlocks application to simulate a modified version of the game Mastermind. In this game, three different colored blocks are lined up and hidden from the player. The player then tries to guess the colors and the order of the blocks. There are four colored blocks (red, green, blue, yellow) to choose from. After guessing the color of the three hidden blocks the program displays how many of the colors are correct and how many of the colors are in the right position. Based on this information the player makes another guess and so on until the player has determined the correct order and color of the hidden blocks. Use R for Red, G for Green, B for Blue, and Y for Yellow. The application interface should look similar to the following after making a guess and clicking Check Guess:

Exercise 14  ———— RockPaperScissors
Modify the RockPaperScissors application created in the reviews to include a Program menu with New Game and Exit commands. The New Game command should clear the labels and set all the radio buttons to False.

Exercise 15 (advanced)  ———— GameOf21
Create a GameOf21 application to simulate a simplified version of the game “21” against the computer. A deck with cards numbered 1 through 10 is used and any number can be repeated. The program starts by dealing the user two randomly picked cards and itself three randomly picked cards that are not revealed until Check Scores is clicked. The user may then draw one card. If the user and computer scores are both over 21, or if both are equal but under 21, the game is declared a draw. Otherwise, the winner is the one with the highest score less than or equal to 21. If one score is over 21 and the other is 21 or less, the player with 21 or less is declared the winner. The result should be displayed in a message box. The application interface should include a Program menu with Play Game and Exit commands. The application should look similar to the following after cards have been dealt and drawn and Check Scores clicked:
Exercise 16  

BasicMultiplication

Create a BasicMultiplication application that allows the user to multiply two numbers that are each in the range 0 through 10. The application should allow the user to select the first operand and the second operand. The NumberUpDown object is a good choice for getting input within a range. The NumberUpDown control looks similar to a text box, but includes arrows for scrolling through a list of numbers:

![NumberUpDown Control Example](image)

The NumberUpDown control has the properties:

- **(Name)** identifies a control for the programmer. NumberUpDown object names should begin with num.
- **Minimum** can be set to a value that will be the least value the user can select.
- **Maximum** can be set to a value that will be the greatest value the user can select.
- **Value** is the number selected by the user.

Exercise 17

The Show() method of the MessageBox class has several optional parameters. A MessageBox.Show() statement can take any of the following forms:

```
MessageBox.Show(message, title)
MessageBox.Show(message, title, buttons)
MessageBox.Show(message, title, buttons, icon)
```

The message and title parameters must be strings. Arguments for the buttons and icon parameters should be selected from the IntelliSense list that is displayed when a comma is typed after the second and third parameters. The IntelliSense list will include arguments for the following sets of buttons:

- Abort, Retry, and Ignore
- OK
- OK and Cancel
- Retry and Cancel
- Yes and No
- Yes, No, and Cancel

A second IntelliSense list will include arguments for the following icons:

- asterisk
- error
- exclamation
- hand
- information
- question
- stop
- warning

Experiment with the Show() method of the MessageBox class by modifying an existing exercise to include a message box.