# Visual Basic 2008 Programming

*Business Applications with a Design Perspective*

Jeffrey J. Tsay

# Table of Contents

# Chapter 5

# Decision

Beginning with Chapter 2, " Visual Basic Programming Concepts," you have seen the need for the If statement (or block) to make a program work properly. It is not an exaggeration to claim that you just cannot write a program without the use of the If block. This is true because however simple a programming problem may be, it is bound to involve some decisions; that is, depending on the condition of one thing, something else will have to be handled differently. For example, suppose your program has a check box, which is used to ascertain whether a patient of a clinic is insured. If so, you want to enable a group box for the user to enter insurance information; otherwise, the group box will be disabled. You will need to use the If block in the check box's CheckedChanged event to determine whether the check box is checked or unchecked, and to execute the proper code accordingly.

When you used the If block previously, you basically looked at the syntax. It is now time to take a closer look at its structure and additional details. Sometimes, various alternative actions need to be taken depending on the result obtained from a single expression, a situation commonly referred to as branching. In such a case, coding with the If block structure can become cumbersome. VB provides an alternative structure, the Select Case block, to handle this situation more concisely. This structure is examined in this chapter.

Both the If block and the Select Case block structures involve the use of logical expressions, which are expressions that can be evaluated to a Boolean value, True or False. For you to thoroughly understand both structures, it is best to have a basic understanding of logical expressions first.

After you finish this chapter, you should be able to:

- Enumerate relational and logical operators and use them in logical expressions,
- Code the If block structure to solve various programming problems that involve decisions,
- Design various alternatives to the If block, and
- Use the Select Case structure.

## 5.1 Logical Expressions

As you may recall, a simple If statement has the following syntax:

```
If Condition Then Statement
```

Where *Condition* represents an expression that can be evaluated to True or False, and *Statement* represents a VB statement such as the assignment statement explained in previous chapters.

The conditional expression is recognized as the logical expression. Two types of operators are used to construct a logical expression: relational operators and logical operators. Let's take a look at relational operators first.

### *Relational Operators*

Relational operators are used to compare the operands, and decide whether the relation is true. For example, you can use the equal operator to test whether two variables, A and B, are equal by coding:

```
If A = B Then MsgBox("A and B are equal")
```

When A is actually equal to B, the expression is True and the message, "A and B are equal" will be displayed; otherwise, the expression will be False, and the message will not be displayed. The following table provides a list of the relational operators.

| Relational Operator | Example | Result/Explanation |
|---|---|---|
| = | `A = B` | True when A is actually *equal* to B; otherwise, False. |
| <> | `A <> B` | True when A is *not equal* to B; otherwise, False. |
| > | `A > B` | True when A is *greater than* B; otherwise, False. |
| >= | `A >= B` | True when A is *greater than or equal to* B; otherwise, False. |
| < | `A < B` | True when A is *less than* B; otherwise, False. |
| <= | `A <= B` | True when A is *less than or equal to* B; otherwise, False. |

## Numeric Comparisons

Typically, relational operators are used to compare numeric data. For example, the following code checks whether the hours worked in a week entered by the user is greater than or equal to 168. If so, a message box is displayed.

```
If Val(txtHoursWorked.Text) >= 168 Then
    MsgBox("Have you ever slept?", MsgBoxStyle.Question)
    txtHoursWorked.Focus
End If
```

In the code, if the number of hours worked is found to be greater than or equal to 168, the message box displays a question mark with a message teasing the user. (Additional details concerning the use of the MsgBox statement are presented in Chapter 6, "Input, Output, and Procedures.") The text box's Focus method is then used to reset the focus on the text box so that the user can correct the number.

### Tip

Recall that the minimum requirement for a user-friendly program is that the message should provide concise information in a friendly tone. The message displayed in the above code is not really a good example. A better message would be:

```
  MsgBox( "The maximum number of hours allowed per week is 60.",
    MsgBoxStyle.Information)
```

Additionally, if the upper limit is 60, the hours worked should be compared with 60, not 168.

## Potential Problem with Conversion

When the operands involved are of different data types, a conversion will have to take place before the comparison is done. Occasionally, a conversion can bring about some unexpected result. Consider this example:

```
Const RateS As Single = 0.095
Const RateD As Double = 0.095
Private Sub btnCompare_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompare.Click
    Dim A As Single, D As Double
    A = RateS * 99895
    D = RateD * 99895
    If A = D Then
        MsgBox("A and D are equal.")
    Else
        MsgBox("A and D are not equal.")
    End If
End Sub
```

If you test the above code in an event procedure, you should see the message "A and D are not equal" is displayed. Why? If you inspect the computational results, D has a value of 9490.025, but A has 9490.024. A is less accurate because its data type is Single, and is unable to represent the data with the exact precision.

Although the advice is to always use the same data type to avoid data conversion in any operations,including comparisons, you occasionally may be faced with a situation when it is necessary to perform mixed-data type comparisons. In such a case, instead of comparing for equality, you can allow a very small difference between the two values to avoid the problem. For example, you can work around the problem with the previous example by modifying the code for the comparison as follows:

```
If Math.Abs((A - D) / D) < 0.000001 Then
```

In this statement, you assume that you are willing to accept a relative difference of $10^{-6}$ as virtually equal; however, this approach will do well only if D is not close to zero. If D is anywhere near zero, you should check the absolute difference, that is, Math.Abs(A – D), instead of the relative difference.

## String Comparisons

The familiar relational operators can also be used to compare string data. For example, you can compare to see whether a name entered in a text box such as txtName is the same as the variable CustName by coding the following:

```
If txtName.Text = CustName Then . . .
```

Notice that all the six relational operators can be used in string expressions. The question is how are the string data compared? For example, is A greater than a? No. The comparison is based on the keycode value of the characters. Because A has a keycode value of 65, and a has a value of 97, a is greater than A.

Note that this result is obtained based on VB's default setting for comparison, *Option Compare Binary*. You can change the setting to *Option Compare Text*, under which the comparison is not case sensitive; that is, a and A are considered equal. Similar to other option settings, such as Option Explicit and Option Strict, you can place Option Compare Text at the beginning of a module (code window) to effect text comparison; or, you can set the option in the project Options dialog box. The steps are exactly the same as setting Option Explicit, which is described in Chapter 4, "Data, Operations, and Built-in Functions."

### Look It Up

Search the Index tab of the help file for the keyword KeyCode enumeration member. It shows the predefined keycode names of most keys.

## Comparing Strings with Unequal Lengths

What happens when two strings are not of the same length? Basically, the comparison is carried out from left to right in pairs, one character after another until an inequality is found or one of the strings runs out of characters; therefore, the string "b" is greater than "abcd" because the first character in the first string, "b" is greater than the first character in the second string. Also, the string "ab" is greater than "a". The latter string runs out of characters before an inequality can be determined.

### Tip

When you set Option Compare Text, all string comparisons will no longer be case sensitive. In some cases, you may want to preserve the case sensitive comparison (that is, set Option Compare Binary) but occasionally need to perform character comparison that is not case sensitive. In such cases, you can use the UCase or LCase functions before the comparison is performed. For example, the code:

```
If UCase(txtName.Text) = UCase(CustName)  Then …
```

will result in True for a name McCord and MCCORD, or any variations of upper- or lowercase spellings on either side of the equality comparison.

## Assignment Statements

Notice that in addition to being used in If statements, the logical expression can also be a part of an assignment statement. That is, you can assign the result of a comparison (or logical operation, explained in the following section) to a numeric variable as follows:

```
D = A = B
```

In this statement, the first = represents the assignment operation (move the result of the operation on the right side to the variable on the left), and the second = is the equality comparison operator that compares whether A and B are equal. In this example, assuming D is a Boolean variable, it will have a value True if A is indeed equal to B; otherwise, it will have a value False. If D is of any kind of numeric data type other than Boolean, an implicit data type conversion will be required. The result will be –1 if A is equal to B; otherwise, it will be 0.

Of course, similar to any typical assignment statement, the result of a comparison can also be assigned to a property setting of any VB control that expects True or False. For example, suppose you would like to enable or disable a check box named chkDorm, depending on whether the variable, Score, is greater than or equal to 80. You can code the following:

```
chkDorm.Enabled = Score >= 80
```

This statement will enable the chkDorm check box when Score is greater than or equal to 80; otherwise, the check box will be disabled.

## Logical Operators

Logical operators are primarily used in logical expressions to perform operations on Boolean data. For example, the following code tests whether the conditions that the purchase amount (named Purchase) is greater than $10,000 and that the account has no previous balance (named Balance) are both true:

```
Status = Purchase > 10000 And Balance = 0
```

Notice that there is the And logical operator between the two comparisons. The And operator requires both of the comparisons to be True to produce a result of True; that is, the variable, Status, will have a value True if the purchase amount is greater than $10,000 and the previous balance is equal to zero.

The following table provides a list of the logical operators often used in programs.

| Logical Operator | Example | Remark/Explanation |
|---|---|---|
| Not | `Not (A = B)` | The Not operator negates the operand on its right; therefore, in the example if A is equal to B, the entire expression will be False. |
| And | `Purchase > 10000 And Balance = 0` | When the expressions on both sides of the And operator are True, the result is True; otherwise, False. |
| AndAlso | `Purchase > 10000 AndAlso Balance = 0` | The result is the same as the And operator, but the operator works differently. When the first expression is false (Purchase is not greater than 10000), the entire expression is false. The second portion is not evaluated. |
| Or | `Purchase > 10000 Or Balance = 0` | When either expression on both sides of the Or operator is True, the result is True; otherwise, False. |
| OrElse | `Purchase > 10000 OrElse Balance = 0` | The result is the same as the Or operator, but the operator works differently. When the first expression is True (Purchase is greater than 10000), the entire expression is true. The second |

| | | |
|---|---|---|
| | | portion is not evaluated. |
| Xor | `Purchase >10000 Xor Balance = 0` | When exactly one of the two expressions on both sides of the Xor (exclusive Or) operator is True, the result is True; otherwise False. That is, when both sides are True or when both are False, the result is False. |

Notice the difference between And and AndAlso and that between Or and OrElse. In general, And and AndAlso should produce the same result; so do Or and OrElse. The differences lie in their ways of obtaining the results and the implication thereof. Both the AndAlso and OrElse operators employ a quick out algorithm. In the case of AndAlso, the result is true only if both operands (expressions) are true, similar to the And operator; however, in the process of evaluation, when the first expression is false, it knows the result must be false and therefore will not evaluate the second expression. By the same token, in the case of OrElse, the result is true if the first expression is true. The operator will not evaluate the second expression. On the other hand, both the And and Or operators evaluate both of their operands regardless of the outcome of the first expression; therefore, AndAlso and OrElse are more efficient. This point is discussed further in Section 5. You may then wonder why there is a need for the And and Or operators. The answer is given at the end of this sub-section.

## Uses of Logical Operators

Here are two examples of using the logical operators:

*Example 1.* A clinic patient who is insured and assigns the right to claim the insurance will be charged a co-payment, while his insurance company will pay for the remaining charges. Assume that the variables, Insured and Assigned, have a value "Y" if the patient is insured, and has assigned the insurance claim to the clinic. To compute the charge to each party, you can code the following:

```
If Insured = "Y" And Assigned = "Y" Then
    ChargeToPatient = CoPay
    ChargeToInsurance = TotalCharge - ChargeToPatient
End If
```

*Example 2.* A store will give a 20% discount to a customer who is either a student or has purchased at least $1,000 worth of goods. Assume that the chkStudent check box is used to identify whether a customer is a student, and the Purchase variable represents the amount of purchase. The computation of discount can be coded as follows:

```
If chkStudent.Checked Or Purchase >= 1000 Then
    Discount = Purchase * .2
End If
```

## The Xor Operator

Although the first three logical operators correspond well to the daily life connotation, the Xor operator may call for some additional explanation. As explained in the table, this operator sets the result to True only if exactly one of its two operands is True. An easy way to understand how it works is to think of it as an inequality operator; that is, when its two Boolean operands are not equal, the result is true. As its name implies, this operator is most useful in handling a situation where the conditions are mutually exclusive. For example, when making a journal entry in accounting records, the user can enter only a debit or credit amount for an account involved in that transaction. Assume the two text boxes, txtDebit and txtCredit, are used for the user to enter the amount and only positive amounts can be entered. The following code can be used to check that one and only one amount is entered:

```
If Val(txtDebit.Text) > 0 Xor Val(txtCredit.Text) > 0 Then
    ' Only one amount has been entered; OK
Else
    'Either both are zero or both amounts are entered.
```

```
    'Display a message
    MsgBox "Please enter exactly one amount for the account."
End If
```

### Bit-Wise Operations

The logical operators, except for AndAlso and OrElse, actually operate on a bit-wise basis. When the operands are numeric data of Short, UShort, Integer, UInteger, Long ,or ULong type rather than the Boolean type, the results can be different from what are expected. Appendix A, "Number Systems and Bit-wise Operations," discusses the additional fine points and some of the uses of these operators in this aspect.

## *Operational Precedence*

Chapter 4 listed the operational precedence of various numeric operators used in an expression. All these rules still hold for all conditional expressions used in an If statement, but now there are two more types of operators: relational and logical operators. How are these new operators ranked? These new operators rank lower in their operational precedence than any of the numeric operators. In addition, the relational operators have precedence over the logical operators, among which Not has the highest precedence followed by And. The following is the list of operators again, ranking from the highest to the lowest in operational precedence:

1. Power (^)
2. Unary negation (-)
3. Multiplication and division (*, /)
4. Integer division (\)
5. Remainder (Mod)
6. Addition and subtraction (+, -)
7. Equal (=)
8. Not equal (<>)
9. Less than, Greater than (<, >)
10. Greater than or equal to (>=)
11. Less than or equal to (<=)
12. Not
13. And, AndAlso
14. Or, OrElse and Xor

Too many rules to remember? If you think so, use parentheses to explicitly express the order of operations. Parentheses help add clarity to what needs to be done first.

### Try This

Want to verify that the And operator actually takes precedence over the Or operator? Consider the following code:

```
    MsgBox(True Or True And False)
```

If Or and And rank equal in precedence, the result should be False; however, if And has the precedence, the result should be True.
Enter the code in the form's click event, run the program, and click the form. What do you see?

Now you have seen how relational and logical operators work. These operators are used to construct logical expressions, which most likely are used in If statements. It is time to take a closer look at the If statement (block).

## 5.2 The If Block

Recall that a simple If statement has the following syntax:

```
If Condition Then Statement
```

When the expression in the condition portion is True, the statement portion is executed; otherwise, it is ignored. For example, you can code the following:

```
If Score > = 90 Then Grade = "A"
```

This statement says if Score is greater than or equal to 90, an "A" is assigned to Grade. If Score is less than 90, nothing is assigned to Grade because the statement portion is ignored.

Note that the condition can be any logical expression. In addition, recall that an expression can be as simple as a variable, control property, or constant, as explained in Chapter 4; therefore, it is correct to code the following:

```
If rdbMale.Checked Then Sex = "M"
```

This statement says if rdbMale.Checked is True, assign "M" to the variable, Sex. Because the Checked property of the radio button rdbMale is either True or False (which is what the computer is checking for), there is no need to code:

```
If rdbMale.Checked = True Then Sex = "M"
```

Any expression that can be evaluated to a Boolean value (either True or False) can be placed in the condition portion:

```
If CBool(A) Then cboZipCode.SelectedIndex = 0
```

Note that A is assumed to be a numeric variable in this line. The SelectedIndex property of the combo box, cboZipCode, will be set to 0 when A is any value but zero. Recall again that a numeric expression is converted to True when it is nonzero and False when zero.

### Tip

In practice, you will find that the simple If statement is seldom used because in most cases either more than one statement will have to be executed or a more complex If structure is called for. Actually, this simple If statement can become a point of confusion. Use the simple If block discussed immediately below even if only one statement is to be executed. It offers the flexibility of adding more statements to execute. It is also easier to trace the logic of your program.

### *Simple If Block*

When a condition is true, you may need to execute more than one statement. In such cases, the previous simple structure would be deficient. The following simple If block will serve the purpose better:

```
If Condition Then
    Statements to be executed
End If
```

Notice that the first line of the block starts with an If, and ends with the keyword Then. There should be no other expression or statement (except for comments) after Then; otherwise, the line will be interpreted as a simple If statement explained previously. Also notice that the block ends with an End If statement. All statements within these two lines will be executed when the condition in the If line is True. Consider the following code in an event procedure:

```
Private Sub chkSort_CheckedChanged(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkSort.CheckedChanged
    If chkSort.Checked Then
        cboSortField1.Enabled = True
        cboSortField2.Enabled = True
    End If
End Sub
```

The If statement checks to see whether the chkSort check box is checked. If so, both the combo boxes containing sort field 1 and sort field 2 are enabled. (*Note:* To test the preceding code, set the Enabled property of the two combo boxes to False at design time.)

What happens to the two combo boxes in the previous example if the check box is clicked off? Nothing inside the If block will be executed. Both can be either enabled or disabled before the event procedure is triggered; however, once enabled, no code inside the procedure will disable the two combo boxes.

What if you really mean to disable the two combo boxes when the check box is clicked off? Given what you have learned in this chapter so far, one way to solve this is to code the procedure as follows:

```
Private Sub chkSort_CheckedChanged(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkSort.CheckedChanged
    cboSortField1.Enabled = False
    cboSortField2.Enabled = False
    If chkSort.Checked Then
        cboSortField1.Enabled = True
        cboSortField2.Enabled = True
    End If
End Sub
```

In the code, the two combo boxes will always be disabled first (regardless of the check state of the check box) and then enabled if the check box is found checked.

## *The If…Then…Else…End If Block*

Although the previous code gives the correct result, you probably feel weird about it. Apparently, it is not completely efficient because it always disables the combo boxes even when they will be enabled immediately inside the If block. Is there an If block structure that allows the execution of a group of statements when a condition is True, and another group when the condition is False? Yes, there is—it was covered in Chapter 2. The syntax structure appears as follows:

```
If Condition Then
    Statements to be executed when condition is True
Else
    Statements to be executed when condition is False
End If
```

With this structure, we can code the preceding example as follows:

```
Private Sub chkSort_CheckedChanged(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkSort.CheckedChanged
    If chkSort.Checked =vbChecked Then
        cboSortField1.Enabled = True
        cboSortField2.Enabled = True
    Else
        cboSortField1.Enabled = False
        cboSortField2.Enabled = False
     End If
End Sub
```

This code will not only produce correct results but literally do what you expect it to: Enable the combo boxes only when the check box is clicked on and disable them only when the check box is clicked off.

### Tip

Beware of the code structure. Some VB beginner may fail to see the *syntax error* in the following structure:

```
If A > B Then Console.WriteLine(A)
Else Console.WriteLine(B)
End If
```

When an If statement ends with an executable statement (as in the first line), VB considers the If block complete; therefore, the Else and End If statements below will have no context**.** *The correct structure* should be:

```
    If A > B Then
        Console.WriteLine(A)
    Else
        Console.WriteLine(B)
    End If
```

## Leaving One Side Out

Note that with this syntax structure, it is okay to omit statements from either the True or False side of the block. For example, the following code should cause no syntax problem:

```
Private Sub chkCitizen_CheckedChanged(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkCitizen.CheckedChanged
    If chkCitizen.Checked Then
        'Respondent is a citizen. No problem.
    Else
        'Respondent is not a citizen. Reject the service.
        MsgBox("You are not qualified to serve as a Juror.")
    End If
End Sub
```

This event procedure is triggered when the user clicks the check box that asks whether the respondent is a citizen. As you can imagine, the check box is used in a data entry screen for the jury selection questionnaire. If the box is checked off, the qualification message is displayed. In this example, a statement is provided only on the False side of the block. You may wonder why not directly test whether the check box is unchecked as follows:

```
    If chkCitizen.Checked = False Then
    'Respondent is not a citizen. Reject the service.
        MsgBox("You are not qualified to serve as a Juror.")
    End If
```

Both blocks of preceding code should work properly. The second block appears to be more concise; however, some companies have coding standards that require all conditions in the If statement be expressed in affirmation. They maintain that such a coding standard makes the code easier to review because of code logic uniformity.

It is also perfectly acceptable to provide code for only the True side of the If block as illustrated in the following code:

```
Private Sub txtName_Validating(ByVal sender As System.Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) Handles txtName.Validating
        If Len(Trim(txtName.Text)) = 0 Then
            MsgBox("Please enter name before proceeding further.")
            e.Cancel = True 'Reset focus on txtName
        Else
        End If
End Sub
```

This event procedure is invoked when the user moves away from a text box named txtName. The If statement checks whether the VB control is blank. If so, an error message is displayed. In addition, the Cancel property of the event argument object e is set to True, which in effect resets the focus back on txtName. As mentioned in Chapter 3, "Some Visual Basic Controls and Events," the Validating event is triggered when the user leaves the control and before the focus is set on the next control. The event is used to perform field-level data validation, which is discussed more thoroughly in Chapter 10, "Special Topics in Data Entry." Note that the previous If block can actually do without the Else line. You can think of the Else statement as being there as a reminder that no statement has been coded on the False side.

## *The If…Then…ElseIf…Then…Else…End If Block*

Many times, a decision can be much more complicated than that in the previous example; that is, several conditions, rather than just the either or situation, will determine the conclusion. For example, a teacher may assign an A to any score greater than or equal to 90, a B if the score is greater than or equal to 80, and a C to all other scores. Is there an If block that can handle this situation conveniently? Yes! The syntax appears as follows:

```
If Condition1 Then
    Statements to be executed when condition1 is True
ElseIf Condntion2 Then
    Statements to be executed when condition2 is True
ElseIf Condition3 Then
    Statements to be executed when condition3 is True
Else
    Statements to be executed when none of the above conditions are True
End If
```

In this structure, you can have as many ElseIf statements as desired following the If statement. You can then code a catch all Else statement after all of the ElseIf statements. It is an error to place an ElseIf after an Else statement. Again, the entire block is closed with an End If statement.

When the block is executed, the conditions in the If block are tested one after another top down (condition1, condition2, and so on) until one condition is true. At that point, those statements under that condition are executed, and the remainder of the If block is skipped. If none of the conditions are true, the statements in the Else portion (if any) are executed.

Using this structure, the teacher's grade assignment problem can be coded as follows:

```
If Score >= 90 Then
    Grade = "A"
ElseIf Score >= 80 Then
    Grade = "B"
Else
    Grade = "C"
End If
```

The code will first check whether Score is greater than or equal to 90. If so, A is assigned to Grade and the remainder of the If block is ignored. If the first condition is not true, the ElseIf statement checks to see whether Score is greater than or equal to 80. If so, B is assigned to Grade and the remaining part of the If block is ignored. On the other hand, if the condition in the ElseIf statement is not true, the statement in the Else block will be executed; that is, C is assigned to Grade.

### Tip

Beware! The following code will not obtain the desired results:

```
If Score < 80 Then
    Grade = "C"
ElseIf Score >= 80 Then
    Grade = "B"
ElseIf Score >= 90 Then
    Grade = "A"
End If
```

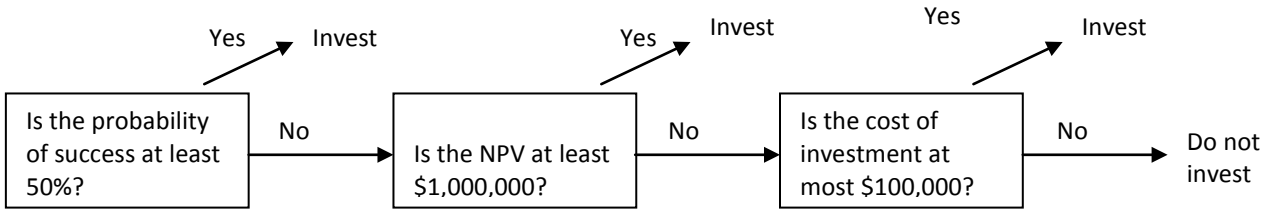
Why? If the score is less than 80, a C is assigned; otherwise, the ElseIf Score >= 80 will be always true, resulting in a B being assigned. There will be no A even if the score is 100. When you are coding this If structure, always test the most restrictive condition first.

## Varying the Criteria

Of course, this structure can be used to handle more complex situations. Consider this investment decision example. An independent oil exploration company will invest in an oil exploration project if a pretest indicates

that the probability of finding oil is at least 50%; otherwise, the net present value of the project will have to be greater than $1 million. If not, the company will invest only if the project will cost less than $100,000. The decision rule can be depicted as in Figure 5-1.

## Figure 5-1
## Decision rules for the investment decision




Note that unlike the previous example, at each stage of the decision, a different factor is involved in this problem. This, however, poses no problem to this If structure. Assume Invest is a Boolean variable indicating whether to invest. Also, the variables, Probability, NPV, and Cost, have been assigned proper values. One way to solve the problem is to code the following:

```
If Probability >= .5 Then
    Invest = True ‘Invest when the probability of success is high
ElseIf NPV >= 1000000 Then
    Invest = True ‘Invest when the expected return is high
ElseIf Cost <= 100000 Then
    Invest = True ‘Invest when the cost is low
Else
    Invest = False ‘ don’t invest if none of the above is true
End If
```

## A Key Validation Procedure: Another Example

Similar to the previous “If Then Else” block, any part of the “If Then ElseIf Then” structure may leave out statements without causing a syntax problem. To illustrate, consider a data-entry key validation routine. You would like to be sure that keys entered into a text box named txtNumber are numbers. If not, you want to display an error message and suppress the key. How do you code this?

As you may recall, when the user presses a key, the KeyPress event is triggered and the KeyChar of the key is also passed as one of the event arguments to this event procedure. In coding this event, you should inspect the key to see if it is numeric.

Be aware that not all the keys that are nonnumeric are bad keys for this data validation purpose. The control keys, such as the Enter key and the Backspace key, are also captured in the KeyPress event. All these control keys have a keycode value lower than 32 (the keycode value of the spacebar, a constant named Keys.Space). You should also display an error message and suppress the key if it does not belong to either of the previous two groups of keys.

How do you check for a numeric key? The Keys enumeration provides constant names for all keys. The keycode values of the keys 0 and 9 are Keys.D0 (48) and Keys.D9 (57), respectively. You can compare the keycode value of the key pressed to see if it is in this range. One way to implement the key validation procedure is as follows:

```
Private Sub txtNumber_KeyPress(ByVal sender As System.Object, ByVal e As
  System.Windows.Forms.KeyPressEventArgs) Handles txtNumber.KeyPress
    Dim KeyAscii As Integer
    KeyAscii = AscW(e.KeyChar) ‘Convert the character to a numeric keycode
    If KeyAscii < Keys.Space Then
```

```
        ' These are control keys, ignore.
    ElseIf KeyAscii >= Keys.D0 AndAlso KeyAscii <= Keys.D9 Then
        ' These are numeric keys, ignore.
    Else
        ' These are neither control keys nor numeric keys.
        ' Display an error message
        MsgBox("Numeric keys only, please!")
        e.Handled = True ' Suppress the key
    End If
End Sub
```

In this code example, no statement appears in either the If or the ElseIf portion of the If block; however, the code works exactly the way you want it to. The key pressed by the user (KeyChar) is first converted to an integer variable (KeyAscii) using the AscW function, which gives the numeric value of a key in unicode. The KeyAscii value is first compared with Keys.Space (32). If the value is less than 32, the key is a control key; otherwise (the ElseIf block), the value is checked to see whether it is within the range of numeric keys. If not (the Else block), an error message is displayed. In addition, the event's Handled argument is assigned a value of True. Assigning True to this argument tells the system that the key has been handled properly and not to process it further. This, in effect, suppresses the key, and no key will appear in the text box.

### Tip

Do you want to know the keycode values of the numeric keys? Try the following code in the form click event:

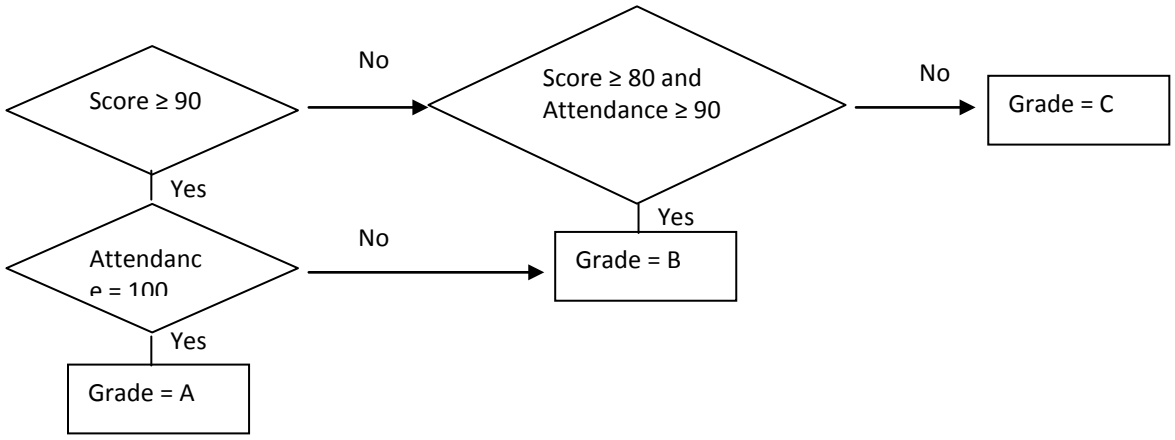
```
Dim I As Integer
For I = Asc("0") To Asc("9")
    MsgBox(Chr(I) & " " & I)
Next I
```

You can also find the keycode value of a key name in your code by resting the mouse pointer on the key name. For example, if you have Keys.D0 in your code, just rest the mouse pointer on D0. You should see the IntelliSense shows a declared constant value of 48.

## *Nesting If Blocks*

Any of the If blocks can be further nested to handle complex conditions. As an illustration, assume a teacher's grading system requires that a student must have not only a score of 90 or better, but also a perfect attendance to earn an A; otherwise, a B is given. A student with a score in the range of 80 to 89 must have at least a 90% attendance record to earn a B. All other students will get a C. These decision rules are depicted in the flowchart in Figure 5-2.

**Figure 5-2**
**Flowchart of grade assignment**

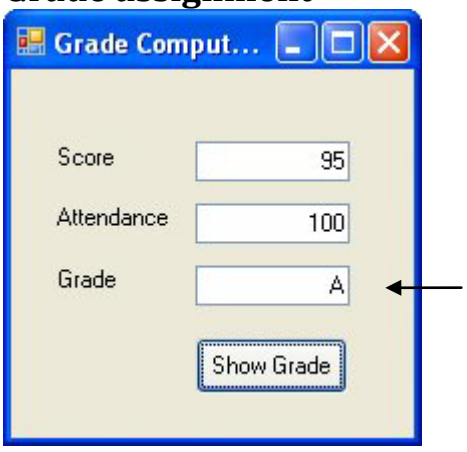


The two text boxes in the user interface shown in Figure 5-3 are used to enter both the score (txtScore) and the attendance percentage (txtAttendance). When the user clicks on the Show Grade button (btnGrade), the grade is displayed in the label named lblGrade. The following code provides a solution:

```
Private Sub btnGrade_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnGrade.Click
    Dim Score As Integer
    Dim Attendance As Integer
    Dim Grade As String
    Score = Val(txtScore.Text)
    Attendance = Val(txtAttendance.Text)
    If Score >= 90 Then
        If Attendance = 100 Then
            Grade = "A"
        Else
            Grade = "B"
        End If
    ElseIf Score >= 80 AndAlso Attendance >= 90 Then
        Grade = "B"
    Else
        Grade = "C"
    End If
    lblGrade.Text = Grade
End Sub
```

In the preceding code, the score and attendance percentage are obtained from the two text boxes, txtScore and txtAttendance, respectively. The score is first compared with 90. If it is greater or equal to 90, the second-level If statement checks whether the attendance percentage is equal to 100. If so, an A grade is assigned; otherwise, a B grade is assigned. The ElseIf statement checks whether the score is greater than or equal to 80 and the attendance is at least 90. If so, a B grade is assigned; otherwise, a C grade is assigned. The resulting grade is displayed in a text box, named txtGrade.

## Figure 5-3
## Grade assignment




In this project, the user enters the score and attendance percentage then clicks the Show Grade button. The resulting score is displayed in another textbox, named txtGrade.

## Will This Work?

You may wonder why not just code as follows to solve the previous problem:

```
If Score>=90 AndAlso Attendance = 100 Then
    Grade = "A"
ElseIf Score >= 80 AndAlso Attendance >= 90 Then
    Grade = "B"
Else
    Grade = "C"
End If
txtGrade.Text = Grade
```

This would appear much simpler. But *the modified code would not work correctly*. The initial problem states that if a student has a score of at least 90 as well as perfect attendance, they will earn an A. Without perfect attendance, they will earn a B. The original solution produces the correct result. The modified code will assign a C to a student who has a score of 100 but with an 80% attendance! The 80% attendance will fail not only the If test, but also the ElseIf test, thus falling into the catch all Else block for an assignment of a grade C.

## Rule for Nesting

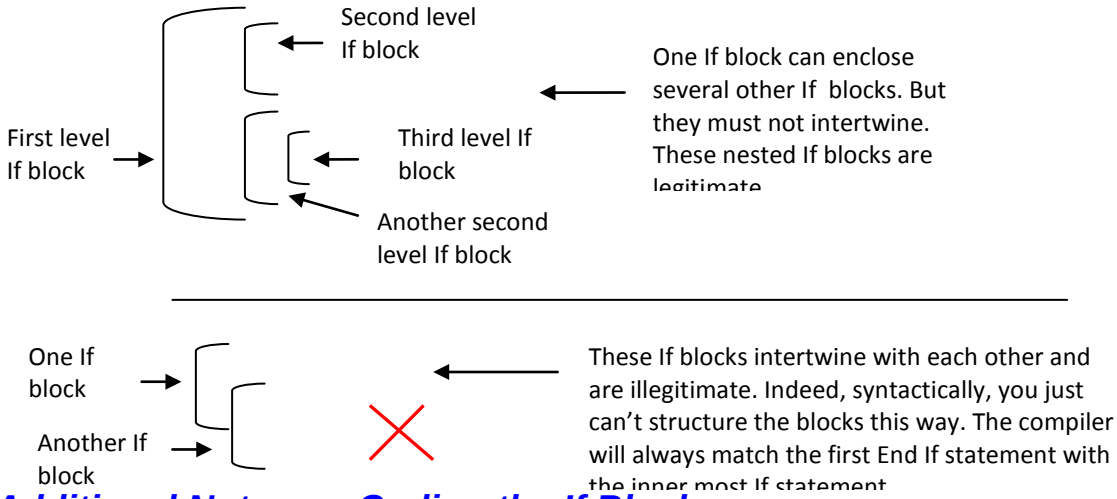
When If blocks are nested, one rule to keep in mind is that one block should enclose the other, constituting a relationship of the inner and outer blocks. It is not logical (and, therefore, not allowed) to intertwine any If blocks. Figure 5.4 shows both the acceptable and unacceptable nesting blocks.

## If Blocks: A Summary of Syntax

In terms of syntax, the If block can be structured several ways. The following table provides a summary.

| Syntax Structure | Example | Remark |
|---|---|---|
| `If condition Then stmt` | `If rdbMale.Checked Then Sex = "M"` | This is not a preferred coding structure. |
| `If Condition Then`<br>`  Statement(s)`<br>`End If` | `If Len(txtName.Text) = 0 Then`<br>`  MsgBox("Must have a name")`<br>`  e.Cancel = True`<br>`End If` | Use this structure to handle situations in which one or more statements must be executed when a certain condition is True, but no action is required when the condition is False. |
| `If Condition Then`<br>`  Statements`<br>`Else`<br>`  Statements`<br>`End If` | `If chkPrintAll.Checked Then`<br>`  lstZipCode.Enabled = False`<br>`Else`<br>`  lstZipCode.Enabled = True`<br>`End If` | Use this structure to handle situations in which a group of statements need to be executed when the condition is True; otherwise, another group of statements are to be executed. |
| `If Condition1 Then`<br>`  Statement group 1`<br>`ElseIf Condition2 Then`<br>`  Statement group 2`<br>`ElseIf Condition3 Then`<br>`  Statementgroup 3`<br>`Else`<br>`  Statementgroup 4`<br>`End If` | `If rdbBreakfast.Checked Then`<br>`  Charge = 4.50`<br>`ElseIf rdbLunch.Checked Then`<br>`  Charge = 7.50`<br>`ElseIf rdbDinner.Checked Then`<br>`  Charge = 10.50`<br>`Else`<br>`  Charge = 0`<br>`End If` | Use this structure when a series of mutually exclusive conditions will require different actions, or cause different results. |

**Figure 5-4**
**Legitimate and illegitimate nesting of If blocks**




## *Additional Notes on Coding the If Block*

Because most programming problems involve many decisions, If blocks are used extensively in programs. It is desirable to consider several issues before leaving this topic.

### Indenting

When If blocks are used, the programming logic becomes less than straightforward because some statements may or may not be executed based on the result of the If expression. Keeping track of all possible results can become challenging. Therefore, it is important to provide as much visual aid as possible in the code to facilitate future review and current logic development/tracing.

One important aspect of this is to indent the code properly. Fortunately, the IDE editor indents your code automatically. Basically, all statements in each portion of the If block are indented. In a nested block, the inside If block skeleton (If, ElseIf, Else, and End If) is indented just like the inside statements. In addition, the statements inside the inner If block are further indented. Such a layout clearly indicates the level of logical nesting. (Note that the compiler will even catch your failure to include a matching End If statement.) After completing the coding of a complex structure of nested If block, take advantage of this extra help by the editor and exam the structure carefully for hints about proper placement of your code block.

### Use of Comments

The importance of using comments (remarks) in programs cannot be overemphasized. Comments can be especially helpful to people who review code that contains many If statements. In many cases, the meaning of the expressions (conditions) in the If statement is not self-explanatory. Comments help you and the reviewer follow the logic of the code much more easily. Consider the following statement:

```
If KeyAscii <Keys.Space Then
```

Without any comments with it, it may take a while for a beginner to figure out what the If statement is attempting to accomplish. Adding a remark similar to the following line can make it much easier for anybody who reads the code:

```
‘ These are control keys and can be ignored
```

### Syntax and Semantic

Beware that you can be syntactically correct, but end up with a wrong result. Consider the following code:

```
If 21 <= Age <=35 Then
    MsgBox("You are a young adult")
End If
```

The code is meant to test whether the variable, Age is in the range of 21 to 35; however, if you test the code, you will find that the message box always displays "You are a young adult" even when Age is 100! Why? In the execution, 21 is first compared with Age to see whether the relation <= is True, yielding an implicit numeric value of either –1 (for True) or zero (for False). When this result is compared with 35, it is always less than 35, so the result is always True.

Note that you can obtain the above result only if you turn off Option Strict (the default). If you set Option Strict On, the compiler will tell you that the implicit conversion from Boolean (result of 21 <= Age) to the data type of 35 (say, Integer) is not allowed. Another good reason to set Option Strict On, isn't it?

## Tip

The correct code to check the age range of 21 and 35 is:

```
If  21 <=Age AndAlso Age <= 35 Then
```

## Efficiency

Another issue concerning the use of the If block relates to the efficiency (speed of execution) versus the clarity of code. Consider this line of code (used in some countries to determine whether a person is a candidate for military draft):

```
If Sex = "M" And Age >=16 Then
```

Although the line of code is clear, it is not as efficient as the following code:

```
If Sex = "M" AndAlso Age >= 16 Then
```

In the first code sample, three operations are always involved in the expression: Sex is compared with "M," Age is compared with 16, and an And operation is performed on the results from the two comparisons. On the other hand, in the second code sample, when the first expression is false (Sex is not equal to "M"), the AndAlso operator will not evaluate the second expression but conclude that the result is false; therefore, this operator is more efficient than the And operator. Note that the use of the AndAlso operator has the same effect as the following code structure:

```
If Sex = "M" Then
    If Age >= 16 Then
```

As you can see, the code line using the AndAlso operator appears to be simpler and should be the preferred way of coding.

Note also the order of the expressions in the If condition can affect the efficiency. The probability of a person being a male is approximately 50%, but the probability of a person being 16 or older is much higher. With the current code, the chance that the second expression is tested is only one half. If the order of the expressions is reversed, however, the chance that Age >= 16 is true is much higher, resulting in more need to evaluate the second expression (Sex = "M") before a conclusion can be reached. The current code is more efficient than if you reverse the order of the two expressions.

Similarly, the OrElse operator is more efficient than the Or operator in evaluating the condition in the If statement. Because the OrElse operator stops evaluating the second expression as soon as it finds the first expression in the condition is true, you should place those expressions with higher probability of being true first. For example, suppose you are writing a program to diagnose a disease that always causes abnormal body temperature mostly fever but occasionally below normal. Your code will be more efficient if you write:

```
If Temperature >100 OrElse Temperature < 97
```

## Code Clarity and Alternatives

As already mentioned, when the code involves too many If blocks (especially nested ones), it can become challenging to trace the logic. The effects of the code can become much harder to figure out. Are there any alternatives? Yes, the possibilities include the following:

- ***Use of the If operator***—The If operator has the syntax:

```
If([Condition], expression 1, expression 2)
```

  This operator returns the value of expression 1 when the condition is evaluated to be True; otherwise, it returns the value of expression 2. For example, in the following statement:

```
A = If(X > 0, Y, Z)
```

  A will have a value Y when X > 0; otherwise, it will have a value of Z.

  The If operator uses short-circuit evaluation. Only the expression that is to be returned will be evaluated; that is, if the condition evaluates to be true, only expression 1 will be evaluated and returned. On the other hand, if the condition is false, only expression 2 will be evaluated and returned.

  The If operator is new to VB and works better than the IIf function previously available. Both take the same three parameters. However, the IIf function evaluates all three parameters before a value is returned; as a result, it is slower than the If operator.

### Tip

Always use the If operator in place of the IIf function. The latter is slower because it evaluates all three parameters. Further, it is more error-prone because both expressions are evaluated regardless of the condition. For example, IIf(X = 0, 0, Y / X) will have an error when X is 0; however, If(X = 0, 0, Y / X) will execute correctly regardless of the value of X.

- ***Computations***—With careful analysis, many of the apparent needs for the If block can be converted into computational formulas. Consider the following code that you used in one of the previous illustrations.

```
Private Sub chkSort_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkSort.Click
    If chkSort.Checked Then
        cboSortField1.Enabled = True
        cboSortField2.Enabled = True
    Else
        cboSortField1.Enabled = False
        cboSortField2.Enabled = False
    End If
End Sub
```

  The procedure enables or disables the cboSortField1 and cboSortField2 combo boxes, depending on whether the chkSort check box is checked. An alternative way to code the procedure is given as follows:

```
Private Sub chkSort_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles chkSort.Click
    cboSortField1.Enabled = chkSort.Checked
    cboSortField2.Enabled = chkSort.Checked
End Sub
```

  In the code, both combo boxes' Enabled properties are assigned the value of chkSort's Checked property. Recall that the value of this property is True when the check box is checked on; and False when the check box is checked off. When the check box is checked (unchecked), both combo boxes' Enabled properties will be set to True (false), causing the two combo boxes to be enabled (disabled). (Again, to test the code, set the Enabled property of the two combo boxes to False at design time.)

  As another example, assume that a bookstore sells only three book titles. A customer can buy only one copy of each book, but can buy any combination of the three books. The chkBook1, chkBook2, and chkBook3 check boxes are used to represent whether the customer wants each of the books. Their prices are represented by the variables Price1, Price2, and Price3, respectively. When the customer is

ready to check out, the cashier clicks on the check boxes and the button, btnCompute, to compute the total purchase. The result is displayed in the label lblTotal. The visual interface appears as in Figure 5-5.

## Figure 5-5
## Visual interface for the Only-Three bookstore


This bookstore sells only three books. After the customer makes selections, the cashier clicks the Compute button to get the total purchase.

One possible solution to this problem is as follows:

```
Private Sub btnCompute_click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompute.Click
    Dim Total as Single
    Total = 0
    If chkBook1.Checked Then
        Total = Price1
    End If
    If chkBook2.Checked Then
        Total = Total + Price2
    End If
    If chkBook3.Checked Then
        Total = Total + Price3
    End If
    lblTotal.Text = Format(Total, "Standard")
End Sub
```

Because the Checked property is True when it is checked and False when it is unchecked, the preceding code can be converted into a computational formula. The modified code appears as follows:

```
Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
System.EventArgs) Handles btnCompute.Click
Dim Total As Single
    Total = Math.Abs(CSng(chkBook1.Checked)) * Price1 + _
    Math.Abs(CSng(chkBook2.Checked)) * Price2 + _
    Math.Abs(CSng(chkBook3.Checked)) * Price3
    lblTotal.Text = Format(Total, "Standard")
End Sub
```

How does the code work? Consider the first book. If its check box is checked, chkBook1.Checked will be True, which when converted to Single is –1. The Math.Abs method will give 1 as a result. The following expression

```
Math.Abs(CSng(chkBook1.Checked)) * Price1
```

will result in a value of Price1. On the other hand, if the check box is off, the result will be zero. The same is true for the expressions for the other two books. Adding the results of the three expressions should yield the correct total. The next line of code uses the Format function to format the value into a string, which is then displayed in (assigned to) the lblTotal label . As in the enabling/disabling combo boxes example, the computational alternative has much shorter code, and the If condition is embedded in the formula.

**Tip**

The following code will also produce the correct result for this example, and is more efficient because the Math.Abs method is called only once.

```
Dim Total As Single
Total = CSng(chkBook1.Checked) * Price1 + CSng(chkBook2.Checked) * Price2 _
    + CSng(chkBook3.Checked) * Price3
Total = Math.Abs(Total)
lblTotal.Text = Format(Total, "Standard")
```

The preceding code focused on computing the total and did not consider how Price1, Price2, and Price3 obtained their values. They should be declared as class level variables, and can be initialized in the Form Load event. So that you can focus on the current discussion, a simple solution is given here. A better solution is proposed in exercise 5-29.

```
Dim Price1 As Single
Dim Price2 As Single
Dim Price3 As Single
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    Price1 = 35
    Price2 = 45
    Price3 = 60
End Sub
```

Return to the discussion of computing the total. Although the alternative appears to be more concise, you must not get carried away with this approach. The meaning of the code may not be as apparent or intuitive as the If block. Sometimes, this alternative approach may not produce the correct result or execute efficiently. When you decide to code a problem using computational formulas in place of the If block, be sure that you test the code thoroughly. In addition, provide comments in the code to explain how the formula works so that the formula will not become a mystery when you review it in the future.

- ***Use of the Select Case Block*—**If the conditions in the "If… ElseIf" block depend on the different results of the same expression, one good alternative to the If block is the Select Case Block. This is explained in the next section.

## 5.3 The Select Case Block

As explained in Section 5.2, an alternative to the If block is the Select Case block, which is particularly useful when the decision depends on the result of one single expression. Consider the previous example of assigning a grade based strictly on the student's score. The grade will be A, B, or C if the score is at least 90, 80, or below 80, respectively. This problem can be conveniently solved by the use of the Select Case block.

## *Syntax Structure*

The Select Case block has the following syntax structure:

```
Select Case Expression
Case Criterion 1
    Statements (group1) to be executed when criterion 1 matches the result of
expression
Case Criterion 2
    Statements (group 2) to be executed when criterion 2 matches the result of
expression
.
.
Case Criterion n
    Statements (group n) to be executed when criterion n matches the result of
expression
Case Else
    Statements (group n+1) to be executed when none of the above criteria match
the result of expression
End Select
```

Where the expression can be any kind, such as numeric, string, or logical and the criteria should have the same type as the expression. Each criterion can consist of a list of single values and ranges. As an illustration, using the Select Case block, the grading problem based purely on score can be coded as follows:

```
Select Case Score
Case Is >= 90
    Grade = "A"
Case 80 To 90
    Grade = "B"
Case Else
    Grade = "C"
End Select
```

As you can see from the illustration, Score is the expression whose value will be evaluated against the criteria stated in the Cases. The first Case (Is >= 90) has an open range; that is, any Score value greater than or equal to 90 will belong to this Case and will result in the statement Grade = "A" being executed. The second Case has a closed range (where both ends are specified) and is expressed with the keyword To. A score in the range of 80 to 90 will result in the statement Grade = "B" being executed. Finally, if none of the cases are true, the statement(s) in the catch all Case Else will be executed.

### Tip

When specifying the range in the Case statement, always specify the lower value first; otherwise, the criterion will never be True. In checking the range, the computer assumes the first number is the lower end of the range and the second value is the higher end. That is, the following Case will never be True:

```
Case 90 To 80
```

### Mutual Exclusiveness of Cases

Notice that the cases are mutually exclusive; that is, only one of the case blocks will be executed. The testing of cases starts from the top and proceeds down. As soon as a criterion (Case) is found true, no more comparisons will be performed. In the previous example, when a score is 90, "A" will be assigned to Grade. The second case will never be tested (even though 90 is also a value in the second case), resulting in a correct assignment decision.

Why not just code the second case “Case 80 to 89”? This will work correctly as long as Score is an integer (Long, ULong, Integer, UInteger, Short, or UShort) variable; however, if Score is a variable that can take on a fractional value, a score of 89.99 will not belong to this case and will become a “Case Else.”

## Try This

Draw a text box on a new form. Enter the following code in the form’s click event:

```
Select Case Val(TextBox1.Text)
    Case 0 To 60 ‘First message
        MsgBox("The number is in the range of 0 and 60.")
    Case 50 To 100 ‘Second message
        MsgBox("The number is in the range of 50 and 100.")
End Select
```

Run the program; then enter a number in the range of 50 and 60. You will always see the first message displayed. The second message is displayed only when you enter a value greater than 60. Do you see how the block works when the criteria in the Case statements are not mutually exclusive?

## Tip

Beware of the logic error like the following:

```
Select Case Age
    Case Is >= 5
        Admission = Half
    Case Is>= 12
        Admission = One
    Case Else
        Admission = 0
End Select
```

You will see that all admissions will be either zero or half priced when the routine runs. Why? Consider the age of 20. It is greater than 5, so the statement Admission = Half will be executed even though the age is greater than 12. To correct the error, place Case Age Is >= 12 as the first criterion. Always place the most restrictive criterion as the first case when coding this structure.

## Detailed Syntax Rules

When coding the criteria, observe the following syntax rules:

- To specify a value to test for equality, give the value. For example, “Case 80” would mean to compare whether the expression in the Select Case statement (the expression, such as Score) is equal to 80.
- To specify several values to test for equality, separate the values in the list with commas. For example, “Case 80, 81, 85” would mean to compare whether the expression is equal to 80, 81, or 85.
- To specify a closed range, insert the keyword To between the lower- and upper-end values. For example, “Case 80 To 90” would mean to test whether the expression is within the range of 80 to 90 (inclusive). Note that the first value must be less than or equal to the second value.
- To specify an open range, use the keyword Is. For example, “Case Is < 0” would mean to test whether the expression is negative.

You can combine all these situations in one Case statement by separating the criteria with commas. For example, you can code the following:

```
Case 50, 80 To 90, Is < 0
```

## Tip

Can you see what’s wrong with the following code?

```
Select Case Age
    Case Age >= 12
```

```
    'Statements
Case Age >= 5
    'Statements
Case Else
    'Statements
End Select
```

The variable Age should be removed from each Case statement. As coded, the first case statement will compare Age with 12 first, resulting in a value of either True or False before this value is compare with Age (the expression specified in the Select Case statement). You will see that the statements in the Case Else block always get executed. If you have set Option Strict On, you will be informed of the conversion restriction that should alert you to the problem. Always set Option Strict On to avoid unexpected problems.

## Variations in the Case Structure

Similar to the If block, you can omit statements from any of the cases. For example, in the preceding section, you used the If block to check for numeric keys for a txtNumber text box. That routine can be replaced by the use of the Select Case block as follows:

```
Private Sub txtNumber_KeyPress(ByVal sender As System.Object, ByVal e As
  System.Windows.Forms.KeyPressEventArgs) Handles txtNumber.KeyPress
    Dim KeyAscii As Integer
    KeyAscii = AscW(e.KeyChar)
    Select Case KeyAscii
        Case Is < Keys.Space
            ' These are control keys; ignore
        Case Keys.D0 To Keys.D9
            'These are numeric keys; ignore
        Case Else
            ' These are neither control keys nor numeric keys;
            ' display an error message
            MsgBox("Numeric key only, please.")
            e.Handled = True 'Suppress the key
    End Select
End Sub
```

This code allows only numeric keys. What if you also expect a decimal point to be entered? The keycode value for period (decimal point) is 46. (There is no predefined name for this key.) To accommodate this, you can insert the following line:

```
Case 46
```

Without any statement following this Case, no message will be displayed, similar to the previous two Cases. Because all three cases are handled in the same way (do nothing), you could combine the three cases together:

```
Select Case KeyAscii
Case Is < Keys.Space, Keys.D0 To Keys.D9, 46
' Legitimate keys; do nothing
Case Else
' Unexpected keys; display a message
    MsgBox("Numeric key only, please.")
    e.Handled = True 'suppress the key
End Select
```

Note that in the first Case statement, commas are used to separate the three criteria. It is interesting to point out that this Case provides a good example of a list of all types of criteria allowed by the syntax: open-ended range, closed range, and single value. Of course, you can add more to the list. For example, if you also expect negative numbers, you can also include 45 (key code for "-") in the Case line:

```
Case Is < Keys.Space, Keys.D0 ToKeys.D9, 45, 46
```

Note that when coding individual values in the list, their order does not really matter; however, for convenience and clarity, you should list them in the order that gives you the clearest meaning.

### Dealing with Constants

The use of the numeric values 45 and 46 in the previous code should cause you to be concerned about code clarity. When you come back to review the code after a while, you may not know what these two values represent. One possible solution is to use the Asc function. Recall that this function gives the ASCII value of a key. In place of 45 and 46, you can code the following:

```
Case Is < Keys.Space, Keys.D0 To Keys.D9, Asc("-"), Asc(".")
```

The code is clearer; however, it will execute a bit slower because the Asc function will need to be called to give the ASCII values of the two keys. Another alternative is to declare two constants for the two keys and then use them wherever they are needed. For example, you can declare the following:

```
Const KeyMinus As Integer = 45
Const KeyPeriod As Integer = 46
```

Then code the Case statement as follows:

```
Case Is < Keys.Space, Keys.D0 To Keys.D9, KeyMinus, KeyPeriod
```

This approach takes a bit more code, but the code is both efficient and clear.

### Try This

Enter "Case < Keys.Space" in the code window for the previous example; then move the cursor to another line. You will see the line you have just entered is changed to "Case Is < Keys.Space". The IDE does more than you expect. This also indicates that the IDE checks the syntax of each line you enter as soon as you move away from it.

## *Nesting Select Case Blocks*

You can nest Select Case blocks in the same way as the If blocks are nested. That is, within each Case of a Select Case block, you can code another Select Case block. The nesting can practically go as deep as you would like. Beware of the problem with clarity, though. As in the case of the If blocks, you should never intertwine a Select Case block with another.

### An Example

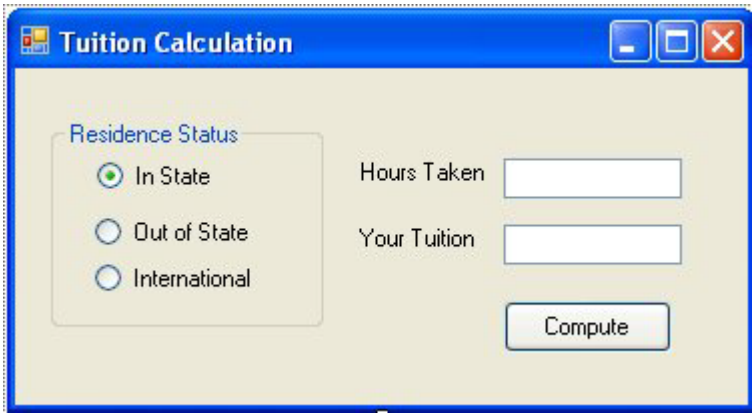
As an illustration of nesting the Select Case blocks, assume that a university charges tuition based on the student's residence status and the number of hours taken, as shown in the following table.

| Hours Taken | In State | Out of State | International |
|---|---|---|---|
| 1–5 | 850 | 1,250 | 1,700 |
| 6–8 | 1,000 | 1,500 | 2,000 |
| 9–11 | 1,300 | 2,000 | 2,700 |
| 12–over | 1,500 | 2,300 | 3,200 |

You are to design a project to determine the student's tuition when the student enrolls. How should you proceed?

As usual, you will need to design the visual interface first. The student must declare a residence status among the three categories. These categories represent mutually exclusive alternatives that will most likely remain unchanged for a long while, so radio buttons will be a good choice. The number of hours taken can be entered into a text box. The visual design can appear as in Figure 5-6.

**Figure 5-6**
**Visual Interface for Tuition Calculation**




In this application, the user selects the residence status, enters the number of hours taken, and then clicks the Compute button. The tuition is displayed in the text box named txtTuition.

A list of the properties of the VB controls used in code is given in the following table.

| Control | Property | Setting | Remark |
|---|---|---|---|
| Radio button | Text | In State | |
| | Name | rdbInState | |
| | Checked | True | As the default option |
| Radio button | Text | Out Of State | |
| | Name | rdbOutOfState | |
| Radio button | Text | International | |
| | Name | rdbInternational | |
| Text box | Name | txtHoursTaken | To enter hours taken |
| Text box | Name | txtTuition | To display tuition |
| | ReadOnly | True | So that the user cannot accidentally change the computed result |
| Button | Text | Compute | To initiate tuition computation |
| | Name | btnCompute | |

Notice that the Checked property of the rdbInState radio button is set to True so that it can be used as the default status. When the user clicks the Compute button, your program will show the result in the txtTuition text box.

There are several ways to determine a student's tuition. One possibility is to use the nested Select Case blocks. As you can see from the previous table, two factors jointly determine the amount of tuition: residence status and hours taken. Either of these factors can be considered the first level of branching. Suppose you use the hours taken as the first level. You can then use the following skeleton to code the problem:

```
Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompute.Click
    Dim Tuition As Decimal
    Select Case Val(txtHoursTaken.Text)
        Case 1 to 5
            'For the first bracket
        Case 6 To 8
            'For the second bracket
        Case 9 to 11
            'For the third bracket
        Case Else
            '12 or more hours
    End Select
```

```
    txtTuition.Text = Format(Tuition, “Standard”) ‘Display the tuition
End Sub
```

Within each Case, you can code the tuition based on the residence status. Should you also use the Select Case structure? You can, but it takes a bit more analysis. Instead, begin with the more straightforward If block first. Inspect the table again; under the 1-5 hours bracket, you see that a student with an in state, out of state, and international status will pay 850, 1250, 1700, respectively. The first bracket can be coded as follows:

```
Select Case Val(txtHoursTaken.Text)
    Case 1 to 5 ‘For the first bracket
        If rdbInState.Checked Then
            Tuition = 850
        ElseIf rdbOutOfState.Checked Then
            Tuition = 1250
        Else
            Tuition = 1700
        End If
    Case 6 To 8 ‘For the second bracket
    Case 9 to 11 ‘For the third bracket
    Case Is >= 12  ’12 or more hours
End Select
```

The rest of the brackets can be coded in a similar fashion and is left to you as an exercise (5-30). For simplicity, the case where the user enters a zero or nothing was also left out. The additional code to take care of this is also left to you.

Note that the If block can be conveniently used here to handle the radio buttons because the state (Checked property) of each button is a different expression. Using the Select Case structure to handle multiple expressions like this can be a challenge. By syntax, the Select Case structure is more natural to branch based on one single expression.

## Identifying the Radio Button Selected

To force the use of Select Case structure to handle radio buttons, a commonly used approach is to introduce a class level variable, whose value can then be set in the Click event of each radio button. For example, when rdbInState, rdbOutOfState, or rdbInternational is clicked, the variable ResidenceStatus can be set to 0, 1, or 2. ResidenceStatus can then be used as an expression in the Select Case statement. The following code should set the value of ResidenceStatus properly.

```
Dim ResidenceStatus As Integer
Private Sub rdbInState_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles rdbInState.Click
    ResidenceStatus = 0
End Sub
Private Sub rdbOutOfState_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles rdbOutOfState.Click
    ResidenceStatus = 1
End Sub
Private Sub rdbInternational_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles rdbInternational.Click
    ResidenceStatus = 2
End Sub
```

After the residence status is determined, you can add the following Select Case block to determine the tuition of the first bracket of hours taken.

```
Select Case ResidenceStatus
    Case 0 ‘In State
        Tuition = 850
    Case 1 ‘Out of State
        Tuition = 1250
```

```
        Case 2 ‘International
            Tuition = 1700
    End Select
```

The complete code for the button Click event procedure using the nested Select Case blocks can then appear as follows:

```
    Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
      System.EventArgs) Handles btnCompute.Click
        Dim Tuition As Decimal
        Select Case Val(txtHoursTaken.Text)
            Case 1 To 5 ‘First bracket
                Select Case ResidenceStatus
                Case 0 ‘In State
                    Tuition = 850
                Case 1 ‘Out of State
                    Tuition = 1250
                Case 2 ‘International
                    Tuition = 1700
                End Select
            Case 6 To 8 ‘Second bracket
                Select Case ResidenceStatus
                Case 0 ‘In State
                    Tuition = 1000
                Case 1 ‘Out of State
                   Tuition = 1500
                Case 2 ‘International
                    Tuition = 2000
                End Select
            Case 9 To 11 ‘Third bracket
                Select Case ResidenceStatus
                    Case 0 ‘In State
                        Tuition = 1300
                    Case 1 ‘Out of State
                        Tuition = 2000
                    Case 2 ‘International
                        Tuition = 2700
                End Select
            Case Is >= 12 ‘12 hours or more
                Select Case ResidenceStatus
            Case 0 ‘In State
                Tuition = 1500
            Case 1 ‘Out of State
                Tuition = 2300
            Case 2 ‘International
                Tuition = 3200
            End Select
        End Select
        txtTuition.Text = Format(Tuition, “Standard”) ‘Display tuition
    End Sub
```

The logic in this procedure appears fairly straightforward. Identifying the radio button selected, however, takes some analysis. Incidentally, there is a trick that you can use with the Select Case structure to identify fairly easily the radio button selected, but it takes some inverted thinking. Rather than explaining its logic here, the trick will be presented in the Explore and Discover exercise (5-11) for you to examine.

## More on the Syntax of Select Case

Because the tuition is jointly determined by the student's residence status and hours taken, can you code the cases as shown here?

```
Select Case ResidenceStatus And HoursTaken
    Case 0 And 1 To 5
    .
    .
End Select
```

No. Although ResidenceStatus And HoursTaken can be considered as an expression, it means something different to the computer than what a beginner may have in mind. Recall that the expression in the Select Case statement is first evaluated to arrive at a value that will be used to match against the criteria in the Cases. In an attempt to evaluate the expression, the computer will first perform a logical And operation on the variables ResidenceStatus and HoursTaken.

Suppose that ResidenceStatus has a value zero (0). No matter what value HoursTaken may be, the result of the And operation will be a zero. (Recall that zero is equal to False and the And logical operator requires that both of its operands be True for the result to be True; otherwise, the result will be False, which is a zero.) The first Case statement that has a zero will be considered the match. Assuming there are no other problems, the result will be both unexpected and unpredictable.

Now, consider this statement:

```
Case 0 And 1 To 5
```

1 To 5 is a range expression; however, an And operator is on its left, which is not expected; therefore, the statement has a syntax error.

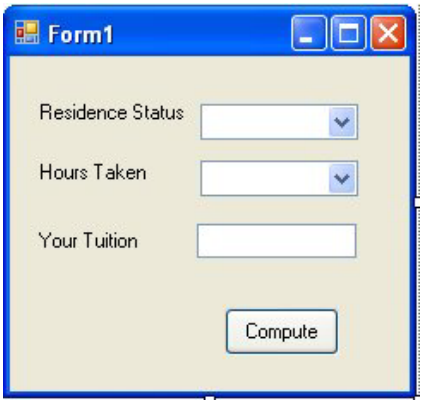
## Mixing the Select Case Structure with the If Block

As you may have already noticed in the preceding example, a Select Case structure can have If blocks within its structure. Conversely, an If block can also contain Select Case structures. There is no restriction to which structure can enclose the other as long as the nesting constitutes a relationship of the inner and outer blocks, as discussed in the preceding section.

## An Alternative Design

The tuition example can be designed somewhat differently. For example, instead of using the text box to enter hours taken and radio buttons for residence status, you can use two combo boxes for both. The first combo box can be used to display the residence status; and the second one, for the brackets of hours taken. The visual interface can then appear as in Figure 5-7.

## Figure 5-7
## Alternative user interface design for tuition calculation




In this alternative design for the preceding tuition problem, the user will select the residence status and the bracket of hours taken from the combo boxes. When he or she clicks the Compute button, the tuition is displayed in the textbox named txtTuition.

A list of the properties of the controls used in code is given in the following table.

| Control | Property | Setting | Remark |
|---|---|---|---|
| Combo box | Name | cboResidenceStatus | |
| | DropDownStyle | DropDownList | To disallow user entry |
| Combo box | Name | cboHoursTaken | |
| | DropDownStyle | DropDownList | To disallow user entry |
| Text box | Name | txtTuition | To display tuition |
| | ReadOnly | True | So that the user cannot accidentally change the computed result |
| Button | Text | Compute | To initiate tuition computation |
| | Name | btnCompute | |

In the Form Load event, the two combo boxes can be populated using the Items.Add method as follows:

```
Private Sub frmTuitionCalculation_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    With cboResidenceStatus.Items
        'Populate combo box with residence status
        .Add("In State")
        .Add("Out Of State")
        .Add("International")
    End With
    cboResidenceStatus.SelectedIndex = 0 'Set default status (In state)
    With cboHoursTaken.Items
        'Populate combo box with brackets of hours taken
        .Add("1 to 5")
        .Add("6 to 8")
        .Add("9 to 11")
        .Add("12 or More")
    End With
    cboHoursTaken.SelectedIndex = 3
End Sub
```

Notice that you have also added statements to set the SelectedIndex properties of both combo boxes. This will make the items corresponding to the SelectedIndex values appear in the controls' text boxes as the default selections when the program starts.

The Click event procedure for the button can then be modified with the following code skeleton:

```
Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
```

```
  System.EventArgs) Handles btnCompute.Click
    Dim Tuition As Decimal
    Select Case cboResidenceStatus.SelectedIndex
        Case 0 'In state
            Select Case cboHoursTaken.SelectedIndex
                Case 0 ' 1 to 5 hours
                Case 1 ' 6 to 8 hours
                Case 2 ' 9 to 11 hours
                Case 3 ' 12 or more hours
            End Select
        Case 1 'Out Of state
            'Similar to the Case for In state
        Case 2 'International
            'Similar to the Case for In state
    End Select
    txtTuition.Text = Format(Tuition, "Standard")
End Sub
```

With the comments in the structure, you should be able to complete the remaining code.

Notice that in the procedure, the residence status is used as the first-level Select Case structure. This is done to show that either of the two factors can be the first level without affecting the result. This structure should appear to you logically more natural because you typically would check the residence status before asking about the number of hours taken.

Notice also that the Select Case statements now use the combo boxes' SelectedIndex properties as the expressions. Because the DropDownStyle properties of both controls are set to DropDownList, the user cannot enter any data into the text portion of the combo box. Instead, the user can select an item from the list. Such an action will set the controls' SelectedIndex properties.

### Which Design Do You Prefer?

The original design has the advantage of code clarity. When you review the code, you should be able to have a good feel about what the code is supposed to do; however, it is not flexible. For example, if the hours taken for each bracket change, and the number of brackets remain the same, you will need to change the program to handle the new situation. The second design has the advantage of code brevity. It is also more flexible in handling the changes in data. Although you use the Items.Add method to populate the combo boxes, in a real application, the items should be read from a file or database. A change in the hours taken for each bracket will not necessitate a revision of the program as long as the number of brackets remains the same. The code, however, is not as clear because the SelectedIndex itself says very little about the underlying data. In such a case, it is important that you include comments to provide the additional details.

There are actually other alternatives to handle this problem. Another alternative that is even more flexible is considered in an exercise in Chapter 8, "Arrays and Their Uses." These alternatives are presented here to illustrate the use of the Select Case structure and the If block to solve a problem. Hopefully, each alternative design will inspire you to become more resourceful in identifying solutions to your next problem.

## *Additional Notes*

The following notes provide additional information pertaining to topics discussed in chapter 4.

### Block Level Declaration

Declaration of variables at the block level was discussed in Chapter 4, "Data, Operations, and Built-In Functions." It was mentioned that variables declared at this level have a block scope. But what constitutes a block in this

context? In the case of the If structure, each of the If, ElseIf, and Else blocks is independent of each other, so the following declarations pose no problem.

```
If A = B Then
    Dim I As Integer
    .
ElseIf A > B Then
    Dim I As Integer
    .
Else
    Dim I As Integer
    .
End If
```

Each of the variables named I is separate and distinct from the other variables of the same name; when the execution leaves the block where I is declared, it will no longer be recognized. Note, however, you cannot declare I in these blocks if you have declared a procedure level variable with the same name. By the same token, if you have inner blocks in one of these blocks, you will not be able to (and should not) declare another I.

These same rules apply to the Select Case structure. You can declare the same variable in each of the mutually exclusive Case blocks, but you cannot make any of these declarations if you have already declared a variable of the same name at the procedure level, nor can you declare the same variable in an inner block (either Case or If block).

### TypeOf Operator versus TypeName

As discussed in chapter 4, both the TypeOf operator and the TypeName function deal with data type determination. When needed, you should find that the TypeName function can be handily used in the Select Case structure. For example, if your code will vary depending on the type of a variable, A, you can code:

```
Select Case TypeName(A)
Case “Integer”
.
Case “String”
.
.
End Select
```

You just cannot use the TypeOf operator in the above Select Case statement.

On the other hand, the TypeOf operator will be a better choice for an If statement. For example the code `If TypeOf(A) Is String` will execute faster than `If TypeName(A) = “String”` because the latter involves string comparison.

## Summary

- The If structure involves testing whether the expression following the If keyword is True or False. In most cases, this expression contains the relational operators and/or logical operators.
- The relational operators include =, <>, >, <, >=, and <=. Each compares its two operands to determine whether the stated relation, such as equal, is True.
- Commonly used logical operators include Not, And, Or, Xor, AndAlso, and OrElse. The first four operators perform operations on their operands on a bit-wise basis. (Additional details are discussed in Appendix A, “Number Systems and Bit-wise Operations.”) The last two operators strictly operate on Boolean data, and employ the quick out algorithm—that is, as soon as the operators are able to draw conclusion on the result, they stop evaluating any expressions thereafter.

- String comparisons can be either Binary (in which the uppercase of any letters is not considered equal to the lowercase of the same letters) or Text (in which the uppercase of any letters is considered equal to the lowercase of the same letters). You make your choice by either coding Option Compare Text (or Binary) in the module or setting Option Compare in the project's Options dialog box.
- If you consider the operational precedence confusing or hard to memorize, use pairs of parentheses to enclose the expression(s) that you want performed first.
- You can construct the If structure in four different ways:
    - Simple If statement:

```
If condition Then Statement
```

    - Simple If block:

```
If condition Then
    Statements
End If
```

    - If … Else block:

```
If condition Then
    Statements
Else
    Statements
End If
```

    - If … ElseIf… Else block

```
If condition1 Then
    Statements
ElseIf condition2 Then
    Statements
ElseIf condition3 Then
    …
Else
    Statements
End If
```

- The practical meaning of the condition (expression) in the If structure may not be very clear. It is advisable to include comments to explain the purpose.
- The computer evaluates logical or relational expressions in a mechanical way, which can be quite different from the way you interpret them; for example, 20 <= Age <= 35. Be aware of the difference. Also, set Option Strict On. It can help identify some of the potential problems.
- For logical expressions, the AndAlso and OrElse operators are more efficient than the And and Or operators.
- In some cases, computations can replace the use of If blocks. The code is typically shorter and can be more efficient, but the resulting code may not be as clear. Weigh the alternatives carefully when you have a choice.
- The Select Case structure can be used to replace the If … ElseIf… Else… End If structure when the branching depends on the result of one single expression.
- Both the If block and the Select Case structure can be nested and can also be nested with each other. There is no restriction as to which structure must contain the other structure.

- Two alternative interface designs were presented to handle the tuition calculation problem. They yielded different solutions with interesting trade-offs between program flexibility and code clarity.
- You can declare block level variables in both the If blocks and the Case blocks, but you cannot declare the same variable in an inner block as the one in the outer block.

## Explore and Discover

**5-1. Use of Logical Operators.** Relational operators work on both numeric and string data. Can logical operators do the same? Try the following statements. (*Note:* Set Option Strict Off. To test these statements, place them in the Form Click event procedure with a Dim A as Integer statement at the beginning; then insert a line of code to display the result between the lines below.)

```
A = “34” And “32”
A = “34” Or 32
A = “XY” Or “AB”
```

Why do the first two run, but not the last? (*Answer:* Logical operators work on numeric data only. Data type conversions are performed automatically by VB in the first two cases, making the code executable. It is impossible to convert the strings to numbers in the last case; so the operation fails.)

**5-2. Comparing the And and the AndAlso operators.** Enter the following code in the form click event.

```
    Dim A As Integer = 64000
    Dim B As Integer = 64000
    If A > 64000 And (A * B) < 192000 Then
        MsgBox("B is less than 3")
    Else
        MsgBox("B is a least 3")
    End If
```

Run the program and then click the form. What do you find? Because the And operator evaluates both expressions, you have an overflow error when the computer is evaluating A * B. Now replace the And operator in the If statement with the AndAlso operator. Run the program, and click the form again. Do you still have the same problem? You should not, because when the expression A > 64000 is not true, the AndAlso operator will not evaluate the second expression.

**5-3. Comparing the Or and the OrElse operators.** Enter the following code in the form click event. Run the program and then click the form. What do you see?

```
    Dim A As Integer = 64000
    Dim B As Integer = 64000
        If A + B > 64000 Or (A * B) > 64000 Then
            MsgBox("Either the sum or the prodcut will be greater than 64000.")
        Else
            MsgBox("Neither the sum or the product will be greater than 64000.")
     End If
```

Now change the Or operator to OrElse, and test the program again. Do you still run into the same problem? For an explanation for the difference, read the last paragraph in 5-2.

**5-4. What is passed to an event procedure?** Create a new project. Draw a text box on the form as well as in the code window; then create a KeyPress event for the text box. (Select TextBox1 from the object box on the upper left corner of the code window; then select the KeyPress procedure from the procedure box on the upper right corner.) Inside the event procedure, type ”e.” (without quotes). The IntelliSense should display a list of choices. You should see KeyChar and Handled among the list.

Create a MouseDown event procedure. Inside the procedure type ”e.” again. What do you see this time?

You know that e is a structure that contains the event arguments passed to your event procedure. Not all events have the same arguments. To find more explanation for the arguments in the keypress event, search the Index tab of the Help file using KeyPressEventArgs as the keyword.

**5-5. What is Its Numeric Value? S**et Option Strict Off in your new project. Draw a check box on the form, name it chkTest, and set its Text property to Test; then enter the following code:

```
Private Sub chkTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles chkTest.Click
    Dim I As Integer
    I = chkTest.Checked
    MsgBox("Checked = " & chkTest.Checked)
    MsgBox("I = " & I)
End Sub
```

Run the project and then click the check box a few times. What do you see? True gives a numeric value of –1, and False gives 0.

**5-6. Handling Data Type Conversion.** Refer to Exercise 5-5. Set Option Strict On. What do you see in the code window? The following statement is underlined because option strict does not allow a Boolean value to be converted to an integer implicitly:

```
   I = chkTest.Checked
```

Change the statement to the following, and test the program again. It should work properly.

```
I = CInt(chkTest.Checked)
```

**5-7. Assigning a Value to the Boolean Variable.** Set Option Strict Off in your new project. Place a text box and a button on the form; then name the text box txtNumber and the button btnCheck. Set the button’s Text property to Check. Enter the following code:

```
Private Sub btnCheck_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCheck.Click
    Dim BoolTest As Boolean
    Dim TheNumber As Single
    TheNumber = Val(txtNumber.Text)
    BoolTest = TheNumber
    MsgBox("The resulting Boolean value is " & BoolTest)
    If BoolTest = TheNumber Then
        MsgBox("BoolTest and TheNumber are equal.")
    Else
        MsgBox("BoolTest and TheNumber are Not equal.")
    End If
End Sub
```

Run the program, enter any number (try at least these numbers: -1, 0, 1000, -30, and 0.005), and click the button. What does the computer display? What conclusion can you draw from this experiment? What do you also learn? (*Answer:* Any nonzero value will be converted to True for a Boolean variable. Only a value of 0 is interpreted as False. Beware of the potential problem with comparing a Boolean variable with another variable of another data type after assigning the same value to both.)

**5-8. Handling Data Conversions.** Refer to Exercise 5-7, and set Option Strict On. What do you see in the code window? You should see many underlines because the compiler no longer permits implicit conversions. You need to convert data explicitly. Revise the code as follows:

```
Private Sub btnCheck_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCheck.Click
    Dim BoolTest As Boolean
```

```
    Dim TheNumber As Single
    TheNumber = CSng(Val(txtNumber.Text))
    BoolTest = CBool(TheNumber)
    MsgBox("The resulting Boolean value is " & BoolTest)
    If CSng(BoolTest) = TheNumber Then
        MsgBox("BoolTest and TheNumber are equal.")
    Else
        MsgBox("BoolTest and TheNumber are Not equal.")
    End If
End Sub
```

Test the program again. Does it work the same way as the preceding exercise? Is there any advantage of coding the program with setting Option Strict On? It forces you to state explicitly what you mean to do. When errors are caused by implicit conversions, it is harder to identify their sources because what we assume and what the computer actually does may not be the same.

**5-9. Playing with the Truth.** Set Option Strict Off. Place the following code in the form click event:

```
    Dim B As Boolean
    Dim L As Long
    Dim S As String
    B = 3 = 3
    L = 3 = 3
    S = 3 = 3
    MsgBox("B = " & B & ", L = " & L & ", S = " & S)
    MsgBox("Len(B) = " & Len(B) & ",Len(L) = " & Len(L) _
      & ", Len(S) = " & Len(S))
    S = L
    MsgBox("S = " & S & ", Len(S) = " & Len(S))
```

Run the project and then click on the form. Does everything turn out to be as expected?
Set Option Strict On. Do you see many underlines indicating syntax problems with the code? Again, setting Option Strict On is one way to detect potential problems with the code.

**5-10. Inverting the Range of the Case Criterion.** Draw a button and a text box on a form; name the button btnCheck, and set its Text property to Check. Name the text box txtNumber. Enter the following code:

```
Private Sub btnCheck_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCheck.Click
    Select Case Val(txtNumber.Text)
        Case 90 To 10 “Note that these two values are inverted
            MsgBox("The value is in the range")
        Case Else
            MsgBox("The value is out of the range")
    End Select
End Sub
```

Run the program, and enter any number between 10 and 90. Which message does the computer display? When the range of the criterion is inverted, the criterion will never be tested to be true.

**5-11. Inverting the Criterion for the Select Case Block.** Typically, the Select Case statement involves a certain expression whose value will vary depending on one or more of the variables or control properties used in the expression. Seldom will you see a single constant used as the expression; however, consider the following code used in the event procedure btnTest click.

```
Private Sub BtnTest_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Select Case True
        Case rdbOne.Checked
            MsgBox(“Radio button1 is selected.”)
        Case rdbTwo.Checked
```

```
            MsgBox("Radio button2 is selected.")
        Case rdbThree.Checked
            MsgBox("Radio button3 is selected.")
        Case Else
            MsgBox("No radio button is selected.")
    End Select
End Sub
```

Does it mean anything to you? Here, the constant True is used as the expression, whereas the Checked property of each radio button is used as the criterion. The comparison is inverted.

Draw on a new form a button, a group box, and three radio buttons inside the group box. Name the button btnTest. Name the three radio buttons rdbOne, rdbTwo, and rdbThree. Enter the preceding code. Start the program. Click one (or none) of the radio buttons. Then click the button. What do you find? It's a tricky way to test which radio button in a group is selected, isn't it?

**5-12. Block Level Declaration.** In an event procedure, declare three variables I, J, and K as Integer; then insert the following code:

```
If J > K Then
    Dim I As Integer
Else
    Dim I As Integer
End If
```

Do you see Dim I underlined in each case? Remove the procedure level declaration for I from the event procedure. Do you still see the underlines? You are allowed to declare the same variable in mutually exclusive blocks.

Now try to enter the following code; that is, insert another If bock with the same declaration:

```
If J > K Then
    Dim I As Integer
    If J > K Then
        Dim I As Integer
    End If
    Else
        Dim I As Integer
End If
```

Do you see any code underlined? Multiple declarations of the same variable are not allowed in a block where the variable is still in scope. Try to declare any variables, such as K, L, and M, and observe the results. Is any part of the code underlined?

**5-13. Block Level Declaration.** Replicate the block declarations in the preceding exercise with the Select Case structure. You should be able to conclude that the compiler treats the If block and the Select Case block in the same manner for variable declarations.

## Exercises

**5-14. Sorting Two Random Numbers.** Draw a button on a new form. Name it btnShow and then set its Text property to Show. Provide the code so that when the user clicks the button, your program will display two random numbers in the range of 0 to 1, smaller one first. (*Hint:* Swap the values if the random numbers are out of order.)

**5-15. Sorting Two Random Numbers in a Lottery.** Modify the preceding project so that the random numbers are integers in the range of 1 to 50, representing the first two numbers drawn from a lottery. Again, the smaller number should be shown first. In addition, the two numbers should not be the same.

**5-16. Sorting Three Random Numbers in a Lottery.** Modify the preceding project so that the program will display three numbers instead of two. (*Hint:* As each number is drawn, make sure it is not the same as the previous one[s]. Sort the first two numbers in order; then compare the third number with the smaller one. If the third one is smaller, you have found the order; otherwise, compare the third one with the larger number.)

**5-17. Ordering Names.** Write a routine so that it will display two names in alphabetic order regardless the order the names are given. Note that the comparison should not be case sensitive.

Test your routine by drawing two text boxes and a button on a new form. When your program runs, it will display the names entered in the text boxes in order when the user clicks the button.

**5-18. Toggling a Button.** Create a new project, and draw a button on the form. Name the button btnTraffic. Provide the code so that when the program starts, the button's text will display Go. When the user clicks the button, it will display Stop. When the user clicks the button again, it displays Go again. Each subsequent click will cause the button to alternate its text between Go and Stop.

**5-19. `I Know What's on Your Mind.`** Design a game that will guess the letter that is on the player's mind. The player is to pick one of the first four letters in the alphabet (a, b, c, or d) without revealing it. Your program is allowed to ask the player two questions. In each question, it presents two letters and asks whether the player's letter is on the list. Your program should be able to tell the player the letter the user has picked after the two questions. (Hint: Suppose the lists your program presents are: a, d [first list] and a, c [second list]. The following table shows the letter the player picked.)

| First Answer | Second answer | Letter Picked |
|---|---|---|
| Yes | Yes | A |
| Yes | No | D |
| No | Yes | C |
| No | No | B |

**5-20. Make the Game More Interesting.** Refer to Exercise 5-19, and revise the game to make it more interesting. Instead of limiting on the first four letters of the alphabet, modify your code so that it will show four consecutive letters, such as m, n, o, p, beginning with a letter randomly generated by your program. Your program will then instruct the player to pick any letter from the four and proceed to ask two questions in the same manner as described in Exercise 5-9. (*Hint:* Use the Next method of the Random object to generate a random number in the range of 0 and 22. Use this random number—say, R—to produce the four consecutive letters [the first letter should be Chr(65 + R)]. Assign these four letters to four variables.)

**5-21. Rotating Background Colors.** Write a procedure that rotates the background color of a label, lblSign, in blue, red, green, yellow, and back to blue each time the user clicks a button named btnChange with a text, "Change." (*Hint:* In the event procedure, type the keyword, Color. Then type a dot [period]. The IntelliSense should display the list of color names you need in your code. Use a Static integer variable to keep track of the count. Use the Mod operator to generate the number sequence 0, 1, 2, 3, 0, 1…and so on, which can then be used in a Select Case block.)

**5-22. Rotating Background Colors Automatically.** Modify the preceding project so that the colors are rotated automatically every half of a second after your program starts. (*Hint:* You may need to also modify the code so that the count will be limited to 0, 1, 2, and 3; that is, not to allow the number to grow any larger. If you allow the number to keep increasing and let the program run for a long while, the number may exceed the variable's upper limit causing an overflow error.)

**5-23. Alternative Solution to the Exploration Problem.** The oil exploration investment decision problem presented in this chapter has an alternative solution. Write a line of code using relational and logical operators to come up with a correct value for the variable Invest. (*Question:* Which coding alternative executes more efficiently?)

**5-24. Random Judgment.** When your program starts, it automatically displays one of the three questions on a label named lblQuestion. (Provide three favorite questions of your own. Use the Rnd function to decide which question to display.) Below the displayed question is a text box named txtAnswer for the user to type in the answer. When the user presses the Enter key, your program will display (using MsgBox) either "Yeah! You are right!" or "Nah! You are wrong!" depending on the number given by the Rnd function; that is, when the random value is less than 0.5, your program will display the Yeah! message; otherwise, it will display the Nah! message. (*Hint:* Place your code in the text box's KeyPress event and then check for the Keys.Return. On the form, use a label to instruct the user to press the Enter key when the user completes the answer.)

**5-25. What Number Is on My Mind?** When the user clicks on the button named btnPlay with the text Play, your program generates a number in the range of 1 to 1,000. The user will guess the number by typing a number in a text box named txtNumber and clicking the Check button named btnCheck. If the guess is too low, your program will display a message, "Nope! Higher." If it is too high, it will display, "Nah! Lower." When the answer is correct, your program says, "Yes, you've got it. But it takes *n* seconds for you to find out." where *n* is computed from the time the Play button is clicked until the time a correct answer is obtained. (*Hint:* Use the DateDiff function to compute the elapse time.)

**5-26. Port of Entry.** When you come back at the port of entry from a trip to a foreign country, the officer from the Immigration and Naturalization Service (a computer guarding the entrance) displays a check box with a question, "Are you a U.S. citizen?" If you check the box, the computer displays a text box with the label, "Enter U.S. Passport Number." If you uncheck the box, the computer displays three text boxes with proper labels, Nationality, Passport Number, and Visa Number, respectively. Design the proper visual interface and then provide the necessary code. (*Question:* Can you do this one without using the If block?)

**5-27. Long-Term Assets.** A part of a long-term asset entry screen has a check box with the text, "Depreciable Asset?" When the box is checked, a text box with the label, "Estimated Life" and a combo box with the label, "Depreciation method" are enabled; otherwise, they are disabled. The allowable depreciation methods are straight-line, sum-of-years' digit, ACRE, modified ACRE, and double-declining balance methods. The most often used method is the straight-line method. (Populate the combo box with the Items.Add method in the form load procedure.) (*Question:* Can you do this without using the If block? The routine will be shorter if you do it this way.)

**5-28. Broker's Commission.** A broker charges her commission based on the amount of trading involved. If the amount is less than $1,000, the commission rate is 2%. *Additional* commission on the amount between $1,000 and $5,000 is 1%; between $5,000 and $20,000 is 0.5%; and above $20,000 is 0.025%. There is a minimum charge of $15.

The user interface to compute the commission should allow the user to enter the Date, Ticker Symbol, Price, and Number of Units (shares). When the program starts, the date field should be automatically filled with the current date. When the user clicks the Compute button, the computer should display the amount of trade as well as the commission in two labels with text box appearance), separately.

**5-29. Revisiting the Only-Three Bookstore Problem.** In the Only-Three Bookstore example in the text, the prices (Price1, Price2, and Price3) are assigned in the form load procedure. Actually, the prices are also shown on the texts of the check boxes. A good design should obtain the same data from only one source so that the results are always consistent. Assume that the check boxes' Text properties should be the sole source of data. Modify the project so that the form load procedure will assign the prices as given in the check boxes' Text properties. (*Hint:*

Use the InStr function to obtain the position of the "$" in the text; then use the CSng function to convert the string to the Single type.)

**5-30. The Tuition Problem.** In Section 5.3 of this chapter, after the tuition problem is introduced, a partial solution was given using an If block as the second-level nesting. Complete the remaining code. For consistency, all the remaining second-level nesting should use the If block.

**5-31. Year-End Bonus.** A company determines the year-end bonuses for its employees based on rank and performance. The bonus is computed as a percentage of the employee's annual salary as given in the following table.

| Rank | Excellent | Good | Mediocre |
|---|---|---|---|
| High | 100% | 75% | 50% |
| Middle | 80% | 50% | 30% |
| Low | 60% | 25% | 15% |

Your form should provide an interface for the user to specify the employee's rank, performance level, and the employee's annual salary. When the user clicks the Compute button, your program will show the amount of bonus for the employee in a label, which should have the appearance of a text box.

## Projects

**5-32. Computing the FICA Withholding.** The FICA tax is computed based on a person's annual income. The formula consists of two parts: the Social Security tax and Medicare tax. Quite a few years ago, the Social Security tax rate was 6.2%, and had an upper limit of an annual income of $45,000; that is, there was no Social Security tax on any income above $45,000. The Medicare tax rate was 1.45% with an upper limit on an annual income of $120,000; that is, there was no Medicare tax on any income above $120,000.

Assume the same tax rate and bracket. Each time an employee is paid, your company has to withhold Social Security taxes based on the employee's current and previous income and withholding. In a real-world situation, the employee's previous income and withholding would have been kept in a file (database), and read in after an employee number was entered. In this project, your program will make the user enter the employee's previous income as well as the current income. Your program will then proceed to compute the previous and current FICA withholding when the user clicks the Compute button. Your program should display the previous and current withholding as well as the total income and withholding including the current amounts. (*Hint:* To compute the current withholding, compute the total withholding based on total income and then subtract the previous withholding based on previous income from it.)

Design a user interface for this purpose. Use three group boxes—one for previous data, another for current data, and the other for total amounts. Use text boxes for fields that require user input as well as for fields that will be computed by your program. Set the ReadOnly properties to True for all text boxes that are used to show computation results. You should add two buttons: one for computation and the other for quitting.

Provide code for the computation and display of results as specified. The following table can be used to verify the accuracy of your program:

| Previous Income (Entered) | Previous Withholding (Computed) | Current Income (Entered) | Current Withholding (Computed) |
|---|---|---|---|
| 0 | 0 | 10,000 | 765 |
| 40,000 | 3,060 | 5,000 | 382.50 |
| 40,000 | 3,060 | 10,000 | 455 |
| 40,000 | 3,060 | 80,000 | 1,470 |
| 40,000 | 3,060 | 100,000 | 1,470 |
| 80,000 | 3,950 | 40,000 | 580 |

| | | | |
|---|---|---|---|
| 80,000 | 3,950 | 80,000 | 580 |

**5-33. Computing Total for Guest Check.** A restaurant offers two kinds of drinks: St. Helen Volcano ($15) and Bloody Mary ($12). They are so strong that diners are allowed to order only one drink per day. Appetizers include shrimp cocktail ($5.95), seasoned mushrooms ($6.45), and hibachi-style crab rangoon ($6.75). Diners can order any combinations, but are restricted to only one drink and one appetizer. Entrees include Maine lobster ($24.95), New York strip steak ($21.95), and oriental vegetables ($28.95). The portions are so large that no diner should order more than one. Finally, desserts include spice cake ($8.75) and Ginseng ice cream ($8.75). Diners can order either, both, or none.

Design a user interface that servers of the restaurant can use to enter the order for each diner. Restrict your choice of VB controls to only radio buttons and check boxes. Use as many group boxes as you deem desirable to group items on the form.

Provide necessary code so that when the server clicks the Compute button, the computer will display in the lblTotal label the total charges for a diner. (*Hint:* You may find the code much shorter using formulas as an alternative to If blocks.)

**5-34. Computing the Total for a Guest Check: A Variation.** Modify the code in the preceding project (5-33) to satisfy the requirement that the restaurant would rather not have the Compute button, but for the computer to display the total as soon as the server clicks on any of the available choices on the restaurant menu. Be aware that when a check box is clicked, it can be checked or unchecked.