Introduction

Objectives

- To learn the syntax of the two types of programming blocks provided in Visual Basic for controlling the execution sequence of a program.
- To be able to decide which of the different types of programming blocks is more appropriate in a given application.
- To be able to program effectively using either type of programming block.

Programming Decision Making

Decision making is an important feature of most programs. The steps that must be performed in the solution to a problem often depend on certain conditions being satisfied. So it is not surprising that all programming languages, including Visual Basic, provide statements that permit conditions to be tested. Based on such tests, the programmer can have the program skip some parts of the program in order to execute only those tasks he or she wishes to perform based on the outcome of the test.

Visual Basic provides two types of programming blocks for the conditional execution of instructions:

1. **IF Block**: Based on the value of a Boolean expression, an IF block permits the programmer to indicate what to do when the expression is True and when it is False.
2. **Case Block**: This block permits the programmer to perform different tasks for each value or range of values that an expression may have.
IF-ELSE-ENDIF

A basic IF block has the following syntax:

If conditional expression Then
    Steps to perform when the value of the conditional expression is True
Else
    Steps to perform when the value of the conditional expression is False
End If

The conditional expression must evaluate to True or False. Therefore, it must be either a Boolean expression or a Relational expression. If you are unsure what these expressions are, you should refer back to Unit 4 of the Study Guide and reread "Relational Operations." As an example, supposing we wish find the integer that is largest among three given integers. A Sub procedure containing the steps required (i.e., the algorithm) might be defined as follows:

Sub Largest_Value(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer)
    ' Outer If block:
    If a < b Then
        ' Nested If block 1:
        If b < c Then
            TextBox4.Text = c
        Else
            TextBox4.Text = b
        End If
    Else
        ' Nested If block 2:
        If a < c Then
            TextBox4.Text = c
        Else
            TextBox4.Text = a
        End If
    End If
End Sub
The Sub procedure places the value of the largest integer into a text box, TextBox4, so that the result can be displayed in a Form. This example illustrates the fact that IF blocks can be "nested": that is, an IF block can be placed inside another IF block. In the example there are two IF blocks inside the "outer" IF block as indicated by the comment statements of the program.

The logic of nested IF blocks can be hard to follow. You may have encountered some difficulty in analyzing the steps of the Sub procedure by reading its body. As a consequence, we will look at some other ways that the body might be defined.

**Simplifying Nested IF Blocks**

One approach is to place the nested IF block inside a general procedure, such as a Sub procedure, thus employing the technique of *procedural abstraction* described in Unit 5. Doing so can be advantageous if the procedure can be reused. In the example above, note how the two IF blocks are identical except for the names of the variables. As suggested previously, such cases can often be better expressed by writing a general procedure for the repetitive code.

Following this strategy, SUB procedure, Larger_Value, is written first that compares two values and returns the largest. This was the purpose of the inner IF blocks in the previous example:

```vbnet
Sub Larger_Value(x As Integer, y As Integer)
    If x < y Then
        TextBox4.Text = y
    Else
        TextBox4.Text = x
    End If
End Sub
```

Next the SUB procedure, *Largest_Value*, can be rewritten with the inner IF block replaced by SUB procedure calls as follows:
Sub Largest_Value(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer)
    If a < b Then
        'Largest is either b or c:
        Larger_Value(b, c)
    Else
        'Largest is either a or c:
        Larger_Value(a, c)
    End If
End Sub

Besides being physically shorter, the logic of this program, although identical to that of the previous program, is probably easier to read. An additional benefit of using a Sub is that it provides an opportunity for the programmer to describe the purpose of the (previously nested) IF block with a meaningful name.

Decisions Based on Keystrokes

A common decision in event-driven programming is to perform a task based on the key that is pressed on the keyboard. Pressing a key is an example of an "event" and so there is an event method associated with it. The introduction of IF blocks provides an opportunity to introduce a new method—the "KeyPress" method—extending those previously described in Unit 3.

Since the pressing of a key is often associated with typing into a TextBox, a KeyPress method can be defined for TextBoxes. For example, suppose we want to write a program that permits the user to signal the completion of the entry of data into a TextBox. With the methods discussed previously, we could use a Button for this purpose. However, an alternate way is for the user to press the "Enter" (or "Return") key on the keyboard after typing into the TextBox. To implement the signalling of the completion of data entry in this way, a suitable KeyPress event method needs to be defined.

The KeyPress method detects not just the pressing of the Enter key but also the pressing of any key on the keyboard. Therefore a suitable test is required that compares the key pressed with the Enter key. How is this test done?

The answer is that when any key on the keyboard is pressed, the ANSI character codeword for that key is generated as described in Unit 3. Therefore, if we know the ANSI code for the "Enter" key (13₁₀), we can test for that codeword every time a KeyPress event occurs.
The following method processes KeyPress events for a TextBox named TextBox1:

```vbnet
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As System.Windows.Forms.KeyPressEventArgs) _
Handles TextBox1.KeyPress
    If e.KeyChar = Chr(13) Then
        data = TextBox1.Text
    Else
    End If
End Sub
```

In this example, the conditional expression is `e.KeyChar = Chr(13)`. This is a relational expression, and so is either True or False. If the relational expression is True, then the assignment statement `data = TextBox1.Text` is executed: that is, the contents of TextBox1 is stored in the class variable called data. On the other hand, if the expression is False, "nothing" happens since there are no statements to execute following the Else statement.

Some parts of the statements in this example will be new to you because they have not yet been discussed in this Study Guide. You should, however, recognize that the Sub has two formal parameters, the first named `sender` of type `Object` and the second named "e" of type `System.Windows.Forms.KeyPressEventArgs`.

The parameter "e" is an object called an "event argument." This object provides information about the key that was just pressed. As you should remember from Unit 1, objects have properties. Since we are interested in the character associated with the key that was pressed, it is available to us as the "KeyChar" property of the object e. So, just as `TextBox1.Text` retrieves the Text property of TextBox1, so the expression `e.KeyChar` retrieves the KeyChar property of e, which is a string of length 1.

In order to compare the string given by `e.KeyChar` with the "Enter" character, we must determine what ANSI code is assigned to the Enter key. By looking it up in Appendix A of the textbook, you will find that Enter (also called "Return," "Carriage Return," and "CR") has the codeword "13." The function call, "Chr(13)," creates a character string of one character whose codeword is 13 and so represents the Enter key. It can be compared with the value of `e.KeyChar`, by using the relational expression: `e.KeyChar = Chr(13)`
The ELSEIF Clause

The following example illustrates an "extended" form of the IF block that is useful when there are more than two possibilities to consider. When comparing two numbers, x and y, there are three possibilities: x is less than y, x is equal to y, or x is greater than y. Each of these possibilities can be expressed by a relational expression, but there are more than two cases to consider. One solution is to use a nested IF block:

```plaintext
If x < y Then
    Steps to perform if x is less than y
Else
    If x = y Then
        Steps to perform if x equals y
    Else
        Steps to perform if x is greater than y
    End If
End If
```

An alternate way of defining this IF block without having to use a nested block is to express it as follows:

```plaintext
If x < y Then
    Steps to perform if x is less than y
ElseIf x = y Then
    Steps to perform if x equals y
Else
    Steps to perform if x is greater than y
End If
```

This example introduces an "ELSEIF clause" into the structure of the IF block. For each additional distinct case that must be tested, another ELSEIF clause can be introduced.

ELSEIF clauses can be used when the programmer wishes to select a set of instructions to perform based on the value of some expression. One such application is in the design of a program that counts the number of values that occur in each of several different intervals. Such an application is called a "frequency counter." The following example provides a simple illustration:
Program Example Using IF

Problem Description
A sports marketing firm wants to test the theory that people have a preference for low numbers on sports uniforms. So, a computer is placed in a public area where people are asked to type their favourite number between 1 and 99 into a text box in a window. The program then keeps track of which one of four intervals the value falls. The intervals are 1 to 10, 11 to 25, 26 to 50, and 51 to 99. As well, the number of times the person enters a value not in any of the intervals is also counted.

The primary procedure for this application uses an IF block. Class variables — inv_1_to_10, invl_11_to_25, invl_26_to_50, invl_51_to_99 and outside_Invls will be used as "frequency counters" and have been declared. Each variable will keep track of the number of times a value falls within a given interval. For example, the variable, inv_1_to_10 will track the number of times a value is provided that falls in the interval from 1 to 10.

The procedure for updating the appropriate frequency counter of occurrences is:

Sub Update_Freq_Count(value As Integer)
    If value < 1 Then
        outside_Invls = outside_Invls + 1
    ElseIf value <= 10 Then
        invl_1_to_10 = invl_1_to_10 + 1
    ElseIf value <= 25 Then
        invl_11_to_25 = invl_11_to_25 + 1
    ElseIf value <= 50 Then
        invl_26_to_50 = invl_26_to_50 + 1
    ElseIf value <= 99 Then
        invl_51_to_100 = invl_51_to_100 + 1
    Else
        outside_Invls = outside_Invls + 1
    End If
End Sub

While this example illustrates how to handle multiple options using an IF block containing ELSEIF clauses, many programming languages, including Visual Basic, provide another way to express this logic, called a "Case Block."
Case Blocks

The IF block is controlled by an expression that can assume only Boolean values. When there are more than two possibilities, nested IF blocks are required. Alternatively, ELSEIF clauses may be used, but these too require expressions that have Boolean values.

Case blocks are programming structures that permit the programmer to define the tests in a more compact way. The basic organization of a Case block is as follows:

Select Case expression
  Case First Set of possible values of the expression.
    Instructions placed here will be executed only if the expression's value is a member of the first set of values.
  Case Second Set of possible values.
    Instructions placed here will be executed only if the expression's value is a member of the second set of values
  Case . . .
  . . .
  Case Last Set of specified values.
    Instructions placed here will be executed only if the expression's value is member of the last set of values.
  Case Else
    Instructions placed here will be executed only if the expression's value is not a Member of any of the sets of values in the previous Case Clauses.
End Select

The expression provided in the Select Case statement can be any valid VB expression having a numeric or String data type. Each line of the CASE block following the declaration is called a "clause." Each "set of specified values" can be a single value or a range of values. The instructions that follow are only executed if the value of the expression matches one of the values in the set of specified values for that clause.

If a match occurs for a given clause, then the expression is said to "satisfy that clause." If more than one clause can be satisfied, then only the instructions following the first satisfiable clause are executed. It is possible that none of the clauses will be satisfied.
However good programming practice requires that there be a clause for all possible values of the expression.

Unit: Using a CASE Statement

One typical application for Case blocks arises when you want to handle different characters in a string in different ways. For example, suppose you wish to count the frequency of occurrence of each vowel in a word. This example can be implemented using an IF block:

```
If letter = "a" Then
    count_a = count_a + 1
ElseIf letter = "e" then
    count_e = count_e + 1
ElseIf letter = "i" Then
    count_i = count_i + 1
ElseIf letter = "o" Then
    count_o = count_o + 1
ElseIf letter = "u" Then
    count_u = count_u + 1
Else
    ' Not a vowel
End If
```

However, a more elegant implementation can be defined using a Case block:

```
Select Case letter
Case "a"
    count_a = count_a + 1
Case "e"
    count_e = count_e + 1
Case "i"
    count_i = count_i + 1
Case "o"
```

count_o = count_o + 1

Case "u"
    count_o = count_o + 1
Case Else
    ' Not a vowel
End Select

Specifying a Range of Values

In the previous example, each clause is defined by just one letter. However in a CASE block each clause can be defined by a set of values. When the number of members in a set is small, they can be listed explicitly, as illustrated in the following example to count the number of vowels:

Select Case letter
    Case "a", "e", "i", "o", "u"
        count_vowels = count_vowels + 1
    Case Else
        count_consonents = count_consonents + 1
End Select

Just to compare, the following IF block also performs the same task:

If letter = "a" Or letter = "e" Or letter = "i" _
    Or letter = "o" Or letter = "u" Then
        count_vowels = count_vowels + 1
Else
    count_consonents = count_consonents + 1
End If

Even when the number of members in one or more sets is larger, it is often possible to use a Case block. For example, the Sub procedure Update_Freq_Count can be written using a Case block as follows:
Sub Update_Freq_Count(value As Integer)
    Select Case value
    Case Is < 1
        outsideInvls = outsideInvls + 1
    Case 1 To 10
        invl_1_to_10 = invl_1_to_10 + 1
    Case 11 To 25
        invl_11_to_25 = invl_11_to_25 + 1
    Case 26 To 50
        invl_26_to_50 = invl_26_to_50 + 1
    Case 51 To 100
        invl_51_to_100 = invl_51_to_100 + 1
    Case Else
        outside_Invls = outside_Invls + 1
    End Select
End Sub

Note the use of the relation operator, "<", in the first Case clause. The possible ways to express values in Case clauses are:

- One or more values separated by commas;
- A range of values, using the keyword "To" between the minimum and maximum values;
- A relational expression beginning with the keyword "Is", followed by a relational operator, followed by a value.