Introduction

Objectives

- To become familiar with what constitutes a valid loop.
- To learn the syntax for defining loops using a Do While block.
- To learn the syntax for defining loops using the For Next block.
- To understand how to define loops using recursive definitions.

Iteration

Back in the early days of computer programming, the question was raised about how many different kinds of control structure might be required by any program. By "control structures," programmers mean the ways in which the next statement to be executed in a program can be determined. Earlier in this Study Guide you have been exposed to two control structures:

- "Sequencing" is the name given to the control structure where the next statement to execute immediately follows the current one. Sequencing is the normal "state of affairs" where the order of our statements in a program specifies the order in which we want to execute them.
- "Conditional Branching" is the control structure used when what gets performed next depends upon the value of an expression. Two examples of conditional branching are the IF block and the Case block. As was indicated, programmers really don't need both, but having the choice does allow them to address the issue in different ways depending on the nature of the test, which can lead to more readable programs.

In the 1960's two pioneers in computer science, Corrado Bohm and Guiseppe Jacopini demonstrated that there are really only three different types of control structure needed to write any program and that any other control structures that might be proposed are actually either equivalent to one or a combination of the three.
They are:

1. Sequencing,
2. Conditional Branching
3. Iteration

"Iteration" is the control structure that is defined to permit the repetition of programming statements in what is commonly called a Loop.

You have already been using programming loops in your Visual Basic programming even though you may not have realized it. Event-driven programs necessarily employ iteration because they must repeatedly respond to events initiated by users on the same control objects. Each time a user performs the same event, such as clicking on a button object, the corresponding method that handles that event for that object must be re-evaluated. In effect, the body of the method is repeated each time the event occurs, and hence a loop is being performed.

However, not all loops are a consequence of a reaction to an event. In many cases, they occur because there is a need to perform a sequence of instructions for a fixed number of steps or until a certain condition arises. Visual Basic provides several other ways besides event handling to define loops, among them "While loops" and "For loops."

The DO WHILE Loop

A "While loop" is defined in Visual Basic by a segment of programming called a Do While block. The syntax of a Do While block is defined as follows:

Do While conditional expression
    Steps to be repeated (the body of the loop)
Loop
    Steps to perform following the loop

Rather than being terminated by an End statement, as has been the case with previous programming blocks, the Do While block is terminated by the keyword Loop. The meaning of the block should be clear from the choice of keywords:

"Do repeat, While the conditional expression is True, the statements before Loop"

To illustrate its application, suppose we wish to write a Function procedure that computes the product of the first n positive integers (called the factorial function). One way to do so is to begin with the number 1 and then to repeatedly multiply by the next positive integer, accumulating the growing product until we have multiplied the product by n. With this description as the basis of our algorithm, one way to define the body of a Function procedure to compute the product: \(1 * 2 * 3 * \ldots * n\) is as follows:
Function Factorial(n As Integer) As Integer

    Dim nextInt, nFactorial As Integer
    nextInt = 1
    nFactorial = 1
    Do While nextInt <= n
        nFactorial = nFactorial * nextInt
        nextInt = nextInt + 1
    Loop
    Return nFactorial
End Function

Inspection of this example reveals the use of a **Do While** block:

1. The block is introduced by the keyword phrase **Do While**
2. This is followed by the conditional expression **nextInt <= n**
3. The body of the loop is given by the sequence of assignment statements:
   4. **nFactorial = nFactorial * nextInt**
      **nextInt = nextInt + 1**
5. The While Block is terminated by the keyword **Loop**.
6. The statement to be executed following the loop is: **Return nFactorial**

Besides the obvious keywords, there are some other essential features that are part of every loop, and therefore the programmer must ensure that they are addressed. Specifically

1. There must be an appropriate test for termination of the loop.
2. The test must be based on some variable whose value is initially defined.
3. The test must be subject to change with each iteration.
4. The loop must terminate, and the test must be able to determine when it does.

Unlike previous control sequences, which do not adversely affect a program’s behavior, failure to define a loop properly can result in a program never terminating. Called, appropriately, an "infinite loop," such an adverse circumstance can result in the programmer having to terminate the program by a system service call in order to return control to the operating system. If you were running the program within the VS IDE and forgot to save your files, you may be unable to do so subsequently. It is therefore important for you to ensure that all four requirements listed are met to the best of your ability. Unfortunately, it has been proven mathematically that it is not possible to verify in
all cases that any given program will always terminate, but for the assignments in this course. You should be able to assure yourself that they will.

The FOR Loop

Visual Basic provides another type of block useful for defining iteration—the "For-Next" block. A "For-Next" block is applicable when the following conditions describe the nature of the iteration to be defined:

- The variable that controls the loop assumes a sequence of values that are either strictly increasing or strictly decreasing.
- The difference between two successive values is constant. This value is called the "increment amount."
- The initial and final values of the sequence are known and are not changed during the execution of the body of the loop.

The syntax of a "For-Next" block is as follows:

```
For control variable As type = initial value To final value Step increment amount
    Steps to be repeated (the loop body)
Next control variable
```

If you examine the variable, `nextInt`, that controls the loop in the previous `Do-While` example, you should make the following observations:

- The variable assumes the values 1, 2,..., n. That is, the sequence of values assigned to it is strictly increasing.
- The increment amount is 1. That is, the difference between any two successive values is the same, in this case 1.
- The initial value is 1 and the final value is n, and neither of these values is changed during the loop.

Therefore, the "Do-While" block of the previous example can also be implemented using a "For-Next" block as follows:

```vbnet
Function Factorial( n As Integer) As Integer
    Dim nFactorial as Integer
    nFactorial = 1
    For nextInt As Integer = 1 To n Step 1
        nFactorial = nFactorial * nextInt
    Next nextInt
    Factorial = nFactorial
End Function
```
Next nextInt
Return nFactorial
End Function

Compare the "For-Next" loop with the "Do-While" loop given earlier. You should note that the "Do-While" loop has an explicit initialization assignment statement to declare explicitly the type of nextInt and set it to 1. This is handled in the "For-Next" block as part of the declaration statement:

For nextInt As Integer = 1 To ...

Similarly, an explicit assignment statement is required to change the value of the control variable, in this case by incrementing nextInt by 1. This is handled in the "For-Next" block by the closing statement of the block:

Next nextInt

As suggested by the keyword "Next", this statement assigns the next value to nextInt by adding the increment amount specified in the "For loop" declaration to it, as long as the new value of nextInt is less than or equal to n. Otherwise, executions moves on to the first statement following the Next statement (i.e., Return nFactorial)

Programming with a FOR Loop

As a second example, suppose that you wished to write a function called HasLetter that determines if a given letter is in a word. If the letter belongs to the word, the function will return the value True; otherwise, it will return the value False. Since the word will be represented by a string of letters, you need to examine each letter of the word, one at a time, and compare it with the letter you are looking for. This describes an iterative process, and so your solution will require a loop. Further:

1. The sequence of values of the control variable will specify the positions of the letters in the word, from left to right. So this is an increasing sequence.
2. The increment amount is 1, assuming you wish to proceed from one letter in the word to the next, from left to right.
3. The initial value of the control variable should be 0 because it is the position of the left-most letter. The final value is given by the expression Len(word)-1.

"Len" is the Visual Basic built-in function that returns the number of characters in a string. Remember, if there are n characters in a string, they are located at positions 0, 1, 2,..., (n - 1) and not at positions 1, 2,..., n.
From this analysis, you can determine the appropriate values for the initial value, final value, and increment amount in order to define a suitable For-Next loop. The following Visual Basic Function provides one possible solution:

Function HasLetter (letter As String, word As String) As Boolean
    For posn As Integer = 0 To Len(word) - 1 Step 1
        If letter = word.Substring(posn, 1) Then
            Return True
        Else
            End If
        Next posn
    Return False
End Function

Notice that the Function returns "early" if it finds the letter in the word because it does not need to look at all the letters of the string. If the letter is not in the word, however, the function must examine all the letters to determine that this is the case. If the loop ends after looking at all the letters, no match could have been found, and so the value False is returned.

Of course, you could also have defined a solution to this problem using a Do While block:

Function HasLetter (letter As String, word As String) As Boolean
    Dim posn As Integer
    posn = 0
    Do While posn < Len(word)
        If letter = word.Substring(posn, 1) Then
            Return True
        Else
            posn = posn + 1
        End If
    Loop
    Return False
End Function
Other Step Increments

In the examples of For-Next blocks provided so far, the increment value is 1. When this is the case, a step need not be specified. For example,

```
For nextInt As Integer = 1 to n step 1
```

can be written more simply as

```
For nextInt As Integer = 1 To n
```

When no increment value is provided, the Visual Basic compiler assumes the value of the step is 1.

In the previous examples, the sequences are increasing. For a decreasing sequence, the increment value to provide should be a negative number. For example if we want to generate control values in descending order from n down to 1 for some control variable called index, the following For Statement would be required:

```
For index As Integer = n To 1 Step -1
```

A common error with a For-Next loop that is counting down is to make the initial value smaller than the final value.

Suppose you had inadvertently typed the last example as:

```
For index As Integer = 1 to n Step -1
```

Then, if n were some positive value greater than 1, you would be defining an infinite loop! The loop would attempt to count from 1 to n by decrementing 1 from each successive value starting with 1. In other words the successive values of index would be 1, 0, -1, -2, -3, ..., n where n is a non-negative number.

Finally, the sequence values and step value can be integer numbers or real numbers, but be sure you declare the control variable accordingly. For example, if you wish to have a control variable assume values from 0.0 to 9.5 in steps of 0.5, then the following For statement would be required:

```
For value As Double = 0.0 To 9.5 Step 0.5
```

There is one important concern that all programmers should be aware of when using real numbers in a loop. Remember that real numbers are represented by decimal fractions to a maximum number of places of decimal (refer back to Unit 4). Some real numbers cannot be represented exactly as decimal fractions. So repeatedly adding such an approximation will accumulate an error.

For example, the number of iterations performed by the following loop will be 5 and not 4, which might be what was expected:
For x As Double = 0.0 to 1.0 step 0.33
    ...
Next x

This problem is actually more serious because it is not always obvious that an approximate value is being used. Because numbers are represented in base 2 in the computer, many non-repeating decimal fractions are repeating binary fractions, and so such decimal fractions cannot be represented exactly as Double numbers.

Therefore, it is not good practice to use real numbers in loops when the number of iterations is important for determining the correct result. In such cases, use an Integer variable to count the loop iterations, and calculate the sequence of real values independently within the loop.

For example, if the intent of the previous loop is to compute the sequence of values: 0.0, 0.33, 0.66, 0.99, and assign these values successively to the variable x then the following loop will do this:

x = 0.0
For i As Integer = 1 to 4 step 1
    ...
    x = x + 0.33
Next i

Recursive Function Definitions

There is another way to define iteration that does not require any special syntax such as a "Do-While" or "For-Next" block. Consider once again computing the product of the first n positive integers, commonly called "factorial n" and introduced as an example earlier.

A common way to define how to compute factorial n (denoted by n! in mathematics) is by the following "rules":

1. Factorial(0) = 1
2. If n > 0 Then Factorial(n) = n * Factorial(n-1)
3. If n < 0 Then Factorial(n) is undefined.

This is an example of what is called a "recursive definition." The first rule is called a "basis rule" and tells you exactly the value of the factorial function for a particular number, in this case the value 1. Rule 3 is also a basis rule because it tells you immediately that the value of the factorial of a negative number is undefined.
Rule 2 is called a "recursive rule" because it tells you how to compute the factorial of any number greater than 1 by using the number and the value of some previous factorial.

It is possible to express recursive definitions as Visual Basic Function procedures. The factorial function, for example, can be defined recursively, using a **CASE block**, as follows:

```vbnet
Function Factorial( n As Integer) As Integer
    Select Case n
        Case 0 ' The Basis Rule
            Return 1
        Case Is > 1 ' The Recursive Rule
            Return n * Factorial(n-1)
        Case Else
            Return 0 ' indicating an invalid value for n.
    End Select
End Function
```

Notice how easily this Function procedure can be defined from the recursive definition. There is one Case clause for each rule, with the basis rule defined first. The body of each Case clause is a Return statement, returning the value of the expression given by each rule.