**Visual Basic - controlling program flow**

**Introduction**

A procedure’s logic flows through statements left to right and from top to bottom. Only very simple procedures can follow this unidirectional route. The power and flexibility of a programming language comes from its ability to use program control. Program control entails making *decisions* based on *conditions*: *looping* – the repeated execution of a group of statements; and *nesting* – the placing of one control structure inside another.

**Decisions**

These are listed below.

<table>
<thead>
<tr>
<th>If</th>
<th>Then do this.</th>
<th>Applies a single condition,</th>
</tr>
</thead>
<tbody>
<tr>
<td>If</td>
<td>condition true</td>
<td>do this</td>
</tr>
<tr>
<td>Else</td>
<td>do this if condition false</td>
<td></td>
</tr>
<tr>
<td>If</td>
<td>condition 1 true</td>
<td>do this, then go to end</td>
</tr>
<tr>
<td>ElseIf</td>
<td>condition 2 false do this</td>
<td></td>
</tr>
<tr>
<td>Select Case</td>
<td>Several conditions testing the same quantity, runs one of several blocks of code</td>
<td></td>
</tr>
</tbody>
</table>

These controlling statements are discussed in turn below.

1) The first case comes in two versions. The forms are
   **If condition true Then** do this
   or
   **If condition true Then**
   statements to execute if condition true
   **End If**
   The first is a single line statement and has no ending **End If**.

2) **If condition true Then**
   do this
   **Else**
   do this if condition false
   **End If**

3) This case is an extension of the previous case to multiple conditions.
   **If condition 1 true Then**
   do this, then go to end
   **ElseIf condition 2 true Then**
   do this, then go to end
ElseIf condition 3 true Then
   do this, then go to end
.
   .(can have repeated ElseIf statements
.
Else
do this if ALL previous conditions are false. (This Else clause need not be present but is often there as a “catch-all”.
End If

Repeated ElseIf clauses are untidy if the same expression is being compared to a succession of different values. In this case the Select Case is preferable. Suppose a bonus depends on bonus rates which depend on the job classification, here represented by jobClass having values 1, 2, ... 10.
Select Case jobClass
   Case 1
      bonus = salary * 0.1
   Case 2
      bonus = salary * 0.09
   Case 3, 4  ` list can contain several values
      bonus = salary * 0.075
   Case 5 To 8  ` list can contain a range of values
      bonus = salary * 0.05
   Case Is > 8  ` list can use comparison
      bonus = salary * 0.02
   Case Else
      Bonus = 0
End Select

Loops
Do ...Loop structures repeated run a segment of code. There are 4 variants. Each one evaluates a condition to determine whether or not to continue running.

Do While condition
   statements run while condition is true

Loop
This form tests the condition before running the loop. It repeats the loop while the condition remains True. The statements must eventually cause the condition to be False or the loop will run forever. To stop an infinite loop press ESC or CTRL+BREAK. The following example counts the occurrences of the target string within the longstring.
Function CountStrings(longString As String, target As _) String
   Dim position As Integer, count As Integer
   position = 1
   Do While InStr(position, longString, target, 1) >0
      position = InStr(position, longString, _
                   target, 1) +1
      count = count + 1
Loop
  CountStrings = count
End Function

**Do Until** condition

Statements

**Loop**

This tests *condition* before looping and runs until the condition becomes **True**. If initially **True** the statements are not run at all. For example, the loop, below is not run if the response is No.

```vba
response = MsgBox(“Do you want to process data?”, _
vbYesNo)
Do Until response = vbNo
  Call ProcessData
  response = MsgBox(“Do you want to process more _
data”, vbYesNo)
Loop
```

**Do**

statements

**Loop While** condition

This form causes the loop to execute at least once and then tests the condition to see if it should be looped. It will be repeated while the condition is **True**. The following example loops over cells A1 to A100 setting the font to red if the cell contains a letter “t”.

```vba
Sub MakeRed()
  Dim rSearch As Object, c As Object, first As String
  Set rSearch = Worksheets(1).Range(“A1:A100”)
  Set c = rSearch.Find(“t”)
  If Not c Is Nothing Then
    first = c.Address
    Do
      c.Font.ColorIndex = 3
      Set c = rSearch.FindNext(c)
      Loop While (Not c is Nothing) And _
(c.Address <> first)
    Else
      MsgBox “Text not found”
    End If
  End If
End Sub
```

**Do**

statements

**Loop Until** condition

This form runs the loop at least once and stops when the condition becomes **True**.

You use a **Do** loop when you don’t know how many times the statements should be run. If you know that they should be run for a specific number of times use a **For...Next** loop. This uses a counter which increases or decreases on each repetition of the loop and ends when this counter reaches a set value. For example
For $i = \text{lowerBound}$ To $\text{upperBound} \ \text{Step} \ n$
   statements
Next $i$
starts with $i = \text{lowerBound}$ and increments $i$ by amount $n$ until it reaches $\text{upperBound}$. The bounds are integer quantities, and $\text{Steps}$ controls the increments (which may be negative).

For Each $\text{element In group}$
   statements
Next $\text{element}$
This form is similar to the For...Next loop but repeats the statements for each element of a collection of objects or in an array. (This is an advanced feature as we will not discuss objects in much detail in this introductory course). VB defines $\text{element}$ as naming the first element in the group, runs the statements, checks if $\text{element}$ is last one of the group. If it is not it defines $\text{element}$ as the second in the group and executes the statements on this $\text{element}$. This is repeated till all elements are processed. Note that $\text{element}$ is a Variant or Object variable. The example below examines all cells in the current region of cell A1 and deletes its contents if the value is negative. Each cell is an element of the group of cells.
For Each $c \text{ In }$
   Worksheets(1).Range("A1").CurrentRegion.Cells
   If $c.\text{Value} < 0$ Then $c.\text{Delete}$
Next $c$

Nesting
You can place one control structure within another. This is called nesting. The example searches a specified range of cells, $\text{rangeToSearch}$, for the $\text{searchValue}$ and counts how many matches occur.
Function CountValues($\text{rangeToSearch}$, $\text{searchValue}$)
   Dim $\text{counter} \text{ As Integer}$
   ' check that quantity passed in variable $\text{rangeToSearch}$ is
   ' indeed a range object
   If TypeName($\text{rangeToSearch}$) <> "Range" Then
      MsgBox "You didn’t specify a range of cells”
      CountValues = -1
   Else
      For Each $c \text{ In } \text{rangeToSearch.Cells}$
         If $c.\text{Value} = \text{searchValue}$ Then
            counter = counter + 1
         End If
      Next $c$
   End If
CountValues = counter
End Function
(Note this example distinguishes between 12 as a number and 12 as text).
Exiting loops

Use **Exit For** and **Exit Do** to exit **For** and **Do** loops prematurely. There are, however, better and more elegant ways to avoid portions of a macro, e.g.

```vba
i = LBound(searchArray)
ub = Ubound(searchArray)
foundIt = False
Do
    If search Array(i) = findThis Then foundIt = True
    i = i +1
    .
    other statements
    .
Loop While i <= ub And Not foundIt
```