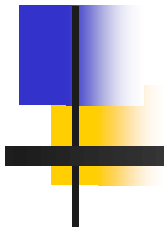
VBA
Data Types and Logic Constructs

IOE 373 Lecture 10



Topics

- Data Types
- Variables and Arrays
- Manipulating Objects
- Loops and Conditionals

Data Types

Data Type	Bytes Used	Range of Values
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	–32,768 to 32,767
Long	4 bytes	–2,147,483,648 to 2,147,483,647
Single	4 bytes	–3.402823E38 to –1.401298E-45 (for negative values); 1.401298E-45 to 3.402823E38 (for positive values)
Double	8 bytes	–1.79769313486232E308 to –4.94065645841247E-324 (negative values);4.94065645841247E-324to 1.79769313486232E308 (for positive values)
Currency	8 bytes	–922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point;+/–7.9228162514264337593543950335 with 28 places to the right of the decimal

More Data Types

Data Type	Bytes Used	Range of Values
Date	8 bytes	January 1, 0100 to December 31, 9999
Object	4 bytes	Any object reference
String (variable length)	10 bytes + string length	0 to approximately 2 billion characters
String (fixed length)	Length of string	1 to approximately 65,400 characters
Variant (with numbers)	16 bytes	Any numeric value up to the range of a double data type. It can also hold special values, such as Empty, Error, Nothing, and Null.
Variant (with characters)	22 bytes + string length	0 to approximately 2 billion
User-defined	Varies	Varies by element

Using Data Types

- Best to use the data type that uses the least bytes and can handle all the data assigned to it.
  - Execution speed is partially a function of the number of bytes that VBA has at its disposal.
- For worksheet calculation, default is Double data type,
  - Generally a good choice for processing numbers in VBA when worried about losing precision.
  - For integer calculations use Integer type (which is limited to values less than or equal to 32,767) or Long data type.

Explicit Variable Declaration

- To force yourself to declare all the variables that you use, include the following as the first instruction in your VBA module:

  Option Explicit

- When this statement is present, VBA won't execute a procedure if it contains an undeclared variable name.

Local Variables

- Declared within a procedure.
  - Local variables can only be used in the procedure in which they're declared. When the procedure ends, the variable no longer exists
  - If you need the variable to retain its value when the procedure ends, declare it as a Static variable.
- Most common way to declare a local variable is to place a Dim statement between a Sub statement and an End Sub statement.
  - Dim statements usually are placed right after the Sub statement, before the procedure's code.

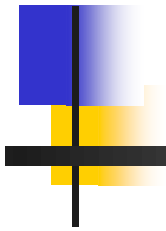
Local Variables

- Can also declare several variables with a single Dim statement. For example:
  - Dim x As Integer, y As Integer, z As Integer
  - Dim First As Long, Last As Double
- VBA doesn't allow declaration of a group of variables by separating the variables with commas.
  - The following statement, does not declare all the variables as integers: Dim i, j, k As Integer (only k)
  - Instead use: Dim i As Integer, j As Integer, k As Integer

Public Variables

- Public variables are available to all the procedures in all the VBA modules in a project
  - Declare by using the Public keyword rather than Dim:

    Public CurrentRate as Long
  - This makes CurrentRate variable available to any procedure in the VBA project,
  - Insert this statement before the first procedure in a module (any module).
  - This type of declaration must appear in a standard VBA module, not in a code module for a sheet or a UserForm.



Constants

- Constants are named values or strings that never change.
- Using constants instead of hard-coded values or strings is an excellent programming practice.
  - Better to declare the value as a constant and use the constant's name rather than its value in your expressions.
  - Code is more readable, and easier to change - change only one instruction rather than in several lines of code.

Constants

- Declare constants with the Const statement. Here are some examples:

  Const NumQuarters as Integer = 4

  Const Rate = .0725, Period = 12

  Const ModName as String = “Budget Macros”

  Public Const AppName as String = “Budget Application”

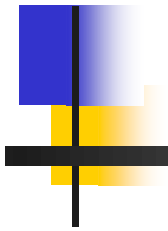
- If you attempt to change the value of a constant in your code, you will get the error “Assignment to constant not permitted”.

Predefined Constants

- Predefined constants can be used without declaring
- Example - Page orientation uses built-in constants: xlLandscape or xlPortrait

```
Sub SetToLandscape()
ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

- The Object Browser can display a list of all Excel and VBA constants. In the VBE, press F2 to bring up the Object Browser.



Arrays

- Group of elements of the same type that have a common name. You refer to a specific element in the array by using the array name and an index number. For example, you can define an array of 12 string variables so that each variable corresponds to the name of a month. If you name the array “MonthNames”, you can refer to the first element of the array as MonthNames(0), the second element as MonthNames(1), and so on, up to MonthNames(11).

Declaring arrays

- Use the Dim or Public statements (just as you declare a regular variable)
  - Specify the number of elements in the array.
    - First index number,
    - The keyword ”To”,
    - And the last index number — all inside parentheses.
  - For example, an array comprising exactly 150 integers:

```
Dim MyArray(1 To 150) As Integer
```

More on Array Declaration

- If specifying only the upper index, VBA assumes that 0 is the lower index. Example:

```
Dim MyArray(0 to 100) As Integer
Dim MyArray(100) As Integer
```

  - In both cases, the array consists of **101** elements.
- By default, VBA assumes zero-based arrays.
- If you would like VBA to assume that 1 is the lower index (for all arrays that declare only the upper index), use the following before any procedures in your module:

```
Option Base 1
```

Declaring multidimensional arrays

- VBA arrays can have up to 60 dimensions, although you'll rarely need more than three dimensions (a 3-D array). The following statement declares a 10x10 Array(Matrix) with two dimensions (2-D):

```
Dim MyArray(1 To 10, 1 To 10) As Integer
```

  - To refer to a specific element in a 2-D array, you need to specify two index numbers. We can assign a value to an element in the preceding array:

```
MyArray(3, 4) = 125
```

Multidimensional Arrays

- A 3-D array that contains 1,000 elements (visualize this array as a cube):

  ```
  Dim MyArray(1 To 10, 1 To 10, 1 To 10) As Integer
  ```

- Reference an item within the array by supplying three index numbers:
  - MyArray(4, 8, 2) = 0
- Can also define dimensions using ReDim

Object Variables

- An object variable represents an entire object, such as a range or a worksheet. Object variables are important for two reasons:
  - simplify your code significantly.
  - make your code execute more quickly.
- Object variables, like normal variables, are declared with the Dim or Public statement.

Object Variables

- For example, the following statement declares InputArea as a Rangeobject variable:

  Dim InputArea As Range, MyArea2 as Range

- Use the Set keyword to assign an object to the variable. For example:

  Set InputArea = Range("C16:E16")

  Set MYArea2=Range("J8:O15")

Simplifying Code with Object Variables

- Examine the following procedure (without object variables):

```
Sub NoObjVar()
Worksheets("Sheet1").Range("A1").Value = 124
Worksheets("Sheet1").Range("A1").Font.Bold = True
Worksheets("Sheet1").Range("A1").Font.Italic = True
Worksheets("Sheet1").Range("A1").Font.Size = 14
Worksheets("Sheet1").Range("A1").Font.Name = "Cambria"
End Sub
```

  - This routine enters a value into cell A1 of Sheet1 on the active workbook, applies some formatting, and changes the fonts and size.

Object Variables

- We can condense the routine with an object variable:

```
Sub ObjVar()
Dim MyCells As Range
Set MyCells = Worksheets("Sheet1").Range("A1:Z100")
MyCell.Value = 124
MyCell.Font.Bold = True
MyCell.Font.Italic = True
MyCell.Font.Size = 14
MyCell.Font.Name = Cambria
End Sub
```

  - Once the variable MyCell is declared as a Range object, the Set statement assigns an object to it. Any Subsequent statements can then use the simpler MyCell reference instead of: Worksheets("Sheet1").Range("A1")

Object Variables

- VBA can access object variables more quickly than lengthy range references that have to be resolved.
- So use object variables to speed up execution.
  - One way to think about code efficiency is in terms of dot processing.
  - Every time VBA encounters a dot, as in Sheets(1).Range("A1"), it takes time to resolve the reference. The fewer the dots, the faster the processing time.

Manipulating Objects and Collections

- Two important constructs that can simplify working with objects and collections:
  - With-End With constructs
  - For Each-Next constructs

With-End With

- With-End With constructs: enables to perform multiple operations on a single object with increased speed.
  - With-End With constructs are processed faster than equivalent procedures that explicitly reference the object in each statement.
- The following procedure, modifies six properties of a selection's formatting (selection is a Range object):

```
Sub ChangeFont1()
Selection.Font.Name = "Cambria"
Selection.Font.Bold = True
Selection.Font.Italic = True
Selection.Font.Size = 12
Selection.Font.Underline = xlUnderlineStyleSingle
Selection.Font.ThemeColor = xlThemeColorAccent1
End Sub
```

With-End With

- We can rewrite using the With-End With construct. The following procedure performs exactly like the preceding one:

```
Sub ChangeFont2()
With Selection.Font
    .Name = "Cambria"
    .Bold = True
    .Italic = True
    .Size = 12
    .Underline = xlUnderlineStyleSingle
    .ThemeColor = xlThemeColorAccent1
End With
End Sub
```

For Each-Next constructs

- When using For Each-Next construct we don't have to know how many elements are in a collection.
- The syntax of the For Each-Next construct is:

For Each [element] **In** [collection]
[instructions]
[Exit For]
[instructions]
Next [element]

For Each-Next Example

- VBA provides a way to exit a For-Next loop before all the elements in the collection are evaluated. Do this with an Exit For statement. The example that follows selects the first negative value in Row 1 of the active sheet:

```
Sub SelectNegative()
Dim Cell As Range
For Each Cell In Selection
    If Cell.Value < 0 Then
      Cell.Select
       Exit For
    End If
Next Cell
End Sub
```

  - If a cell is negative, it's selected, and then the loop ends when the Exit For statement is executed.

For-Next loops

- The simplest type of a good loop is a For-Next loop. Its syntax is:

```
For counter = start To end [Step stepval]
    [instructions]
    [Exit For]
    [instructions]
Next [counter]
```

Example

- This routine executes the Sum = Sum + Sqr(Count) statement 100 times and displays the result (e.g. the sum of the square roots of the first 100 integers.)

```
Sub SumSquareRoots()
Dim Sum As Double
Dim Count As Integer
Sum = 0
For Count = 1 To 100
    Sum = Sum + Sqr(Count)
Next Count
MsgBox Sum
End Sub
```

- Notice this For-Next loop doesn't use the optional Step value or the optional Exit For statement.

For-Next Loops

- Loop counter is a normal variable — nothing special. As a result, it's possible to change the value of the loop counter within the block of code executed between the For and Next statements.
  - However, changing the loop counter inside of a loop is a bad practice and can cause unpredictable results

Example

- You can also use a Step value to skip some values in the loop. Here's the same procedure rewritten to sum the square roots of the odd numbers between 1 and 100:

```
Sub SumOddSquareRoots()
Dim Sum As Double
Dim Count As Integer
Sum = 0
For Count = 1 To 100 Step 2
    Sum = Sum + Sqr(Count)
Next Count
MsgBox Sum
End Sub
```

In this procedure, Count starts out as 1 and then takes on values of 3, 5, 7, and so on. The final value of Count used within the loop is 99. When the loop ends, the value of Count is 101.

Example

- A Step value in a For-Next loop can also be negative. The procedure that follows is intended to delete Rows 2, 4, 6, 8, and 10 of the active worksheet:

```
Sub DeleteRows()
Dim RowNum As Long
For RowNum = 10 To 2 Step -2
    Rows(RowNum).Delete
Next RowNum
End Sub
```

- If we'd used a positive Step value in the previous sub, incorrect rows would have been deleted. Check the sub below:

```
Sub DeleteRows2()
Dim RowNum As Long
For RowNum = 2 To 10 Step 2
    Rows(RowNum).Delete
Next RowNum
End Sub
```

Row numbers below a deleted row get a new row number. (e.g. when Row 2 is deleted, Row 3 becomes the new Row 2.) Using a negative Step value ensures that the correct rows are deleted.

Example – Multiple Exit For

- We can include one or more Exit For statements within the loop. The following procedure finds the **cell with the largest value in Column A** of the active worksheet:

```
Sub ExitForDemo()
Dim MaxVal As Double
Dim Row As Long
MaxVal = Application.WorksheetFunction.Max(Range("A1:A10"))
For Row = 1 To 1048576
    If Cells(Row, 1).Value = MaxVal Then
        Exit For
    End If
Next Row
MsgBox "Max value is in Row " & Row
Cells(Row, 1).Activate
End Sub
```

The maximum value calculated by using the Excel MAX function, value is assigned to the MaxVal variable. The For-Next loop checks each cell in the column. If the cell being checked is equal to MaxVal, the Exit For terminates the loop and the statements following the Next statement are executed.

For Each Next

```
Sub ExitForDemo1()
Dim MaxVal As Double
Dim Cell as Range
MaxVal = Application.WorksheetFunction.Max(Range("A:A"))
For Each Cell in Range("1:100")
      If Cell.Value = MaxVal Then
         Cell.Activate
         Exit For
      End If
Next Cell
MsgBox "Max value is in Row " & Activecell.Row & "Column:" & Activecell.column
End Sub
```

Application.WorksheetFunction allows you to use many pre-defined excel functions directly in your VBA code:

https://docs.microsoft.com/en-us/office/vba/api/excel.worksheetfunction#methods

Do While loops

- This section describes another type of looping structure available in VBA. Unlike a For-Next loop, a Do While loop executes as long as a specified condition is met.
- A Do While loop can have either of two syntaxes:

```
Do [While condition]
[instructions]
[Exit Do]
[instructions]
Loop
```

```
Do
[instructions]
[Exit Do]
[instructions]
Loop [While
condition]
```

  - The difference between is the point in time when the condition is evaluated. In the syntax to the left, the contents of the loop may never be executed. In the syntax to the right, the statements inside the loop are executed at least one time.

Entering a value in the next empty cell/row

- A common requirement is to enter a value into the next empty cell in a column or row.

```
Sub GetData()
Dim NextRow As Long
Dim Entry1 As String, Entry2 As String
Do
'Determine next empty row
NextRow = Cells(rows.count, 1).End(xlUp).Row + 1
' Prompt for the data
Entry1 = InputBox("Enter the name")
If Entry1 = "" Then Exit Sub
Entry2 = InputBox("Enter the amount")
If Entry2 = "" Then Exit Sub
' Write the data
Cells(NextRow, 1) = Entry1
Cells(NextRow, 2) = Entry2
Loop
End Sub
```

Do Until loops

- The Do Until loop structure is very similar to the Do While structure. The difference is evident only when the condition is tested. In a Do While loop, the loop executes while the condition is True; in a Do Until loop, the loop executes until the condition is True.
- Do Until also has two syntaxes:

```
Do [Until condition]
[instructions]
[Exit Do]
[instructions]
Loop
```

```
Do
[instructions]
[Exit Do]
[instructions]
Loop [Until condition]
```

Controlling Code Execution

- Some VBA procedures start at the top and progress line by line to the bottom. (e.g. Macros).
- Often, we need to control the flow of your routines by skipping, executing multiple times, and testing conditions.
- Some additional ways of controlling the execution of your VBA procedures:
  - GoTo statements
  - If-Then constructs
  - Select Case constructs

GoTo statements

- The most straightforward way to change flow is to use a GoTo statement.
  - This statement simply transfers program execution to a new instruction, which must be preceded by a label (a text string followed by a colon, or a number with no colon).
  - VBA procedures can contain any number of labels, but a GoTo statement can't branch outside of a procedure.

GoTo – Example 1

- The following example uses the VBA InputBox function to get the user's name. If the name is not Howard, the procedure branches to the WrongName label and ends.

```
Sub GoToDemo()
UserName = InputBox("Enter Your Name:")
If UserName <> "Howard" Then GoTo ErrorHandler
MsgBox ("Welcome Howard...")
' -[More code here] -
Exit Sub
ErrorHandler:
MsgBox "Sorry. Only Howard can run this macro."
End Sub
```

- Use the GoTo statement only when you have no other way to perform an action. Experts tend to only use GoTo statements for error handling.

If-Then constructs

- The most commonly used instruction grouping in VBA
- Decision-making capability. Good decision-making is the key to writing successful programs.
- Basic syntax of the If-Then construct is

  If condition Then true_instructions [Else false_instructions]
- The Else clause is optional.

Example

- The time of day is expressed as a fractional value — for example, noon is represented as .5. The VBA Time function returns a value that represents the time of day, as reported by the system clock.
- A message is displayed if the time is before noon. If the current system time is greater than or equal to 0.5, the procedure ends, and nothing happens.

```
Sub GreetMe1()
If Time < 0.5 Then MsgBox "Good Morning"
End Sub
```

  - Another way to code this routine is to use multiple statements, as follows:

```
Sub GreetMe1a()
If Time < 0.5 Then
MsgBox "Good Morning"
End If
End Sub
```

If-Then Statements

- If statements have corresponding End If statement.
- We can place any number of statements between the If and End If statements.

```
Sub GreetMe2()
If Time < 0.5 Then
 MsgBox "Good Morning"
Endif
If Time >= 0.5 Then MsgBox "Good Afternoon"
End Sub
```

- We used >= (greater than or equal to) for the second If-Then statement. This covers the chance that the time is precisely 12:00 noon.

Else Clause

- Example:

```
Sub GreetMe3()
If Time < 0.5 Then MsgBox "Good Morning" Else _
MsgBox "Good Afternoon"
End Sub
```

- If-Then-Else can be a single statement, Or:

```
Sub GreetMe3a()
If Time < 0.5 Then
  MsgBox "Good Morning"
  MsgBox "It's a nice day"
` Other statements go here
Else
   MsgBox "Good Afternoon"
` Other statements go here
End If
End Sub
```

How about more than 2 conditions?

- We can use either three If-Then statements or a form that uses ElseIf. The first approach is simpler:

```
Sub GreetMe4()
If Time < 0.5 Then MsgBox "Good Morning"
If Time >= 0.5 And Time < 0.75 Then MsgBox "Good Afternoon"
If Time >= 0.75 Then MsgBox "Good Evening"
End Sub
```

  - The value 0.75 represents 6:00 p.m. — three-quarters of the way through the day and a good point at which to call it an evening.

If-Then

- In the previous example, every IF statement is evaluated (even if the first condition is satisfied) A more efficient procedure would end the routine when a condition is found to be True. This is the syntax:

```
If condition Then
[true_instructions]
[ElseIf condition-n Then
[alternate_instructions]]
[Else
[default_instructions]]
End If
```

If-Then

- For the previuos GreetMe procedure:

```
Sub GreetMe5()
If Time < 0.5 Then
MsgBox "Good Morning"
ElseIf Time >= 0.5 And Time < 0.75 Then
MsgBox "Good Afternoon"
Else
MsgBox "Good Evening"
End If
End Sub
```

- When a condition is True, the conditional statements are executed, and the If-Then construct ends.

If-Then

- This procedure prompts the user for a value for Quantity and then displays the appropriate discount based on that value.
  - Quantity is declared as Variant because it contains an empty string (not a numeric value if the InputBox is cancelled.)
  - This procedure doesn't perform any other error checking (e.g what happens if user enters a negative value?)

```
Sub Discount()
Dim Quantity As Variant
Dim Discount As Double
Quantity = InputBox("Enter Quantity: ")
If Quantity = "" Then
   Exit Sub
ElseIf Quantity < 25 Then
   Discount = 0.1
ElseIf Quantity < 50 Then
   Discount = 0.15
ElseIf Quantity < 75 Then
   Discount = 0.2
Else
   Discount = 0.25
End If
MsgBox "Discount: " & Discount
End Sub
```

VBA's IIf function

- Alternative to the If-Then construct: the IIf function.
- This function takes three arguments and works like Excel's IF worksheet function. The syntax is
- IIf(expr, truepart, falsepart)
  - expr: (Required) Expression you want to evaluate.
  - truepart: (Required) Value or expression returned if expr is True.
  - falsepart: (Required) Value or expression returned if expr is False.

Select Case constructs

- More useful for choosing among three or more options.
  - Also works with two options and is a good alternative to If-Then-Else.
- Syntax:

```
Select Case testexpression
[Case expressionlist-n
[instructions-n]]
[Case Else
[default_instructions]]
End Select
```

Example

- Discount example using a Select Case construct:

```
Sub Discount3()
Dim Quantity As Variant
Dim Discount As Double
Quantity = InputBox("Enter Quantity: ")
Select Case Quantity
Case Is Null
Exit Sub
Case 0 To 24
Discount = 0.1
Case 25 To 49
Discount = 0.15
Case 50 To 74
Discount = 0.2
Case Is >= 75
Discount = 0.25
End Select
MsgBox "Discount: " & Discount
End Sub
```