Using loops and debugging code

Chapter 7 – **Looping your code**

- pp. 103-118
- **Exercises 7A & 7B**

Chapter 8 – **Fixing Bugs**

- pp. 119-132
- **Exercise 8**

Chapter 7 – Looping your code

- Coding a For loop
- Coding a Do loop

Chapter 7 – Looping your code

- It is **quite common** to have software perform some task **repeatedly**, whether it is:
  - Until it has been done **for each member of a set**, e.g. for each record in a shapefile, do the following …
  - Until **some condition is satisfied**, e.g. check the distance between a point of interest and all other points in a shapefile until one is found that is less than a specified threshold
- This **repeated execution** of a few lines of code is called **looping**, and VBA for ArcGIS provides **coding structures** for both of these situations:
  - A **For loop** can be used to execute code a given number of times
  - A **Do loop** can be used execute code until a specified condition is satisfied

Coding a For loop

- **For loops** begin with a line that **specifies a variable** that will keep track of its iterations:

```
For variable = StartValue To EndValue (Step StepValue)
```

- The StartValue and EndValue **specify the range** over which the variable should be iterated, e.g. in a basic example:

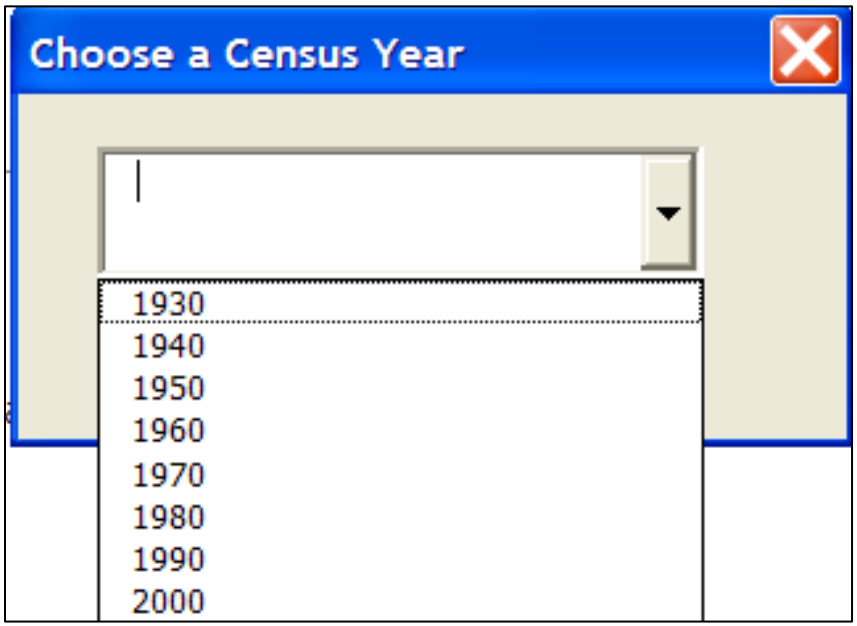
```
For i = 1 To 10
```

the loop **will be executed 10 times**, for each value between 1 and 10, with the **value of i being increased by 1** on each iteration {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Coding a For loop

- Optionally, we can also use the **Step** keyword to **change the increments**, as we will in the exercise when we will use a For loop to populate some choices in a pulldown:

```
For intYear = 1930 To 2000 (Step 10)
```



{1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000}

Coding a For loop

- **For loops** end with a **Next** statement, which simply indicates where the **body of the loop ends** (the body of the loop being everything between the For and the Next)
- Usually, the For loop will **run as many times as the iterator specifies** that it should, but there is **one other way** to write code to **exit a For loop**:
  - An **Exit For** statement can be placed **in the body of the loop**, usually within an **If Then** statement so that if a specified condition occurs, rather than completing the loop's usual number of iterations, we can **jump straight to the Next** statement
  - This is a useful approach when we plan to **search through a number of items** (say, all the layers in a map), until we **find the right one**; once we find it, there is no need to look at the rest

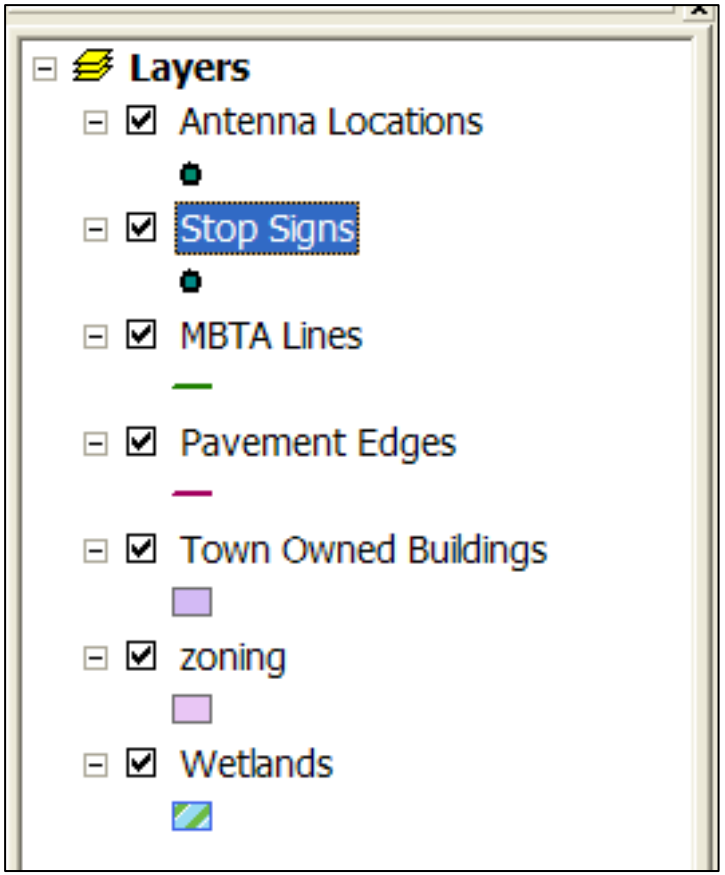
Coding a For loop

– This is a useful approach when we plan to **search through a number of items** (say, all the layers in a map), until we **find the right one**; once we find it, there is no need to look at the rest:

```
Dim pZMap as IMap
Dim x as Integer

For x = o to pMaps.Count – 1
      If pMap.Item(X).Name = “zoning” Then
            set pZMap = pMaps.Item(x)
            Exit For
      End If
Next x

…
```



Coding a Do loop

- **Do loops** are used in the other situation, when you want some code repeated **until some condition is satisfied**
- This can take **two forms**:

  `Do While` – Runs the loop while the specified expression is **true**

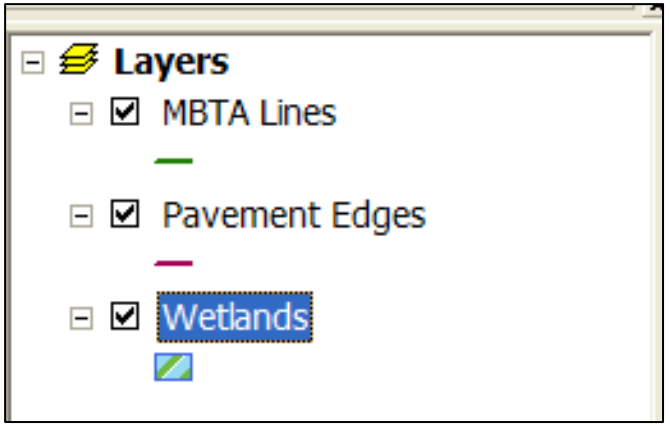
  `Do Until` – Runs the loop while the specified expression is **false**
- **Structurally**, Do loops look a lot like For loops:
  - They have an **opening line**, that in this case **specifies the expression to be checked** to see if the loop should run again:
    `Do While|Until Expression`
  - Rather than ending with a Next, **they end with a Loop** statement
    `Loop`
  - You can use an **Exit Do** to leave the loop from within its body
    `Exit Do`

Coding a Do loop

- For example, suppose we need to move through all the layers in a map, but we do not know how many there are; we can **use a Do loop** like so:

```
‘Layer enum example
Dim pLayer as ILayer
Dim pMapLayers as IEnumLayer
Set pMapLayers = pMap.Layers

Set pLayer = pMapLayers.next
Do Until pLayer is Nothing
    msgbox pLayer.Name
    set pLayer = pMapLayers.next
Loop
```



Chapter 8 – Fixing bugs

- Using the debug tools

Chapter 8 – Fixing bugs

- Now that you have completed a few of the exercises, you have almost certainly had the experience of **having your code not run properly**, and having had ArcGIS greet you with an **error message** instead
- It is very easy, through **small errors in syntax**, to get into this situation and create code with a **bug**
- Fortunately, the ArcGIS VBA environment provides us with some **tools** to make it easier to identify and correct any bugs in our code
- First, though, it is worthwhile to identify **three different kinds of errors** that we might encounter, what their causes are, and what we can do about them

Chapter 8 – Fixing bugs – Compile Errors

- When the code we write is **converted** into the form that the computer will execute, this is called **compiling**
  - We can distinguish between the **code we can read** (the VBA code) and the **code the computer's processor can read** (which is binary and called machine code or assembler language)
- As the VBA compiles, the software **can detect when something doesn't quite make sense** and the VBA cannot be compiled. Some **common reasons** this occurs:
  - You make a **syntax error** by **misspelling** something
  - You make an error by **misusing VBA** (forgetting an argument, not closing a loop, trying to use a method without an object)
- VBA will **highlight** where you made the error

Chapter 8 – Fixing bugs – Run-time Errors

- It is possible for your **code to compile successfully**, but still **cause errors when you try to run it**. These errors are known as **run-time errors**
- Unfortunately, these **cannot be detected before the fact**, because even though there is **nothing wrong with the syntax**, what your code asks the computer to do is **impossible or prohibited** in some way. Some common examples of this are:
  - **Illegal math**, such as a divide by zero error (syntactically valid, but mathematically impossible, e.g. `Acres = 40000 / 0`)
  - **Type mismatch errors**, where you try to use two kinds of objects together in a way that is not viable (e.g. a mathematical expression containing a string like `Acres = “SqFt” / 43650`)

Chapter 8 – Fixing bugs – Logic Errors

- It is possible for your **code to compile successfully and run successfully**, but when it does running, it **does not produce the desired result**. When this is the case, the programmer has usually committed an error known as a **logic error**
- Unfortunately, the **software itself cannot detect a logic error**: You, the programmer, have to know what your software is supposed to do, and when it doesn't do that, you have to be the one who figures out **what went wrong**
- The **key** to detecting logic errors is to **test your code**, usually **thoroughly**, trying to make it encounter **every possible condition** it is likely to encounter in regular use

Using the debug tools

- Regardless of which of the three types of errors you encounter, you can **use the VBA Debug toolbar** to help you **figure out what is wrong**, and **correct the problem**
- The **key capability** of the Debug toolbar is the ability to **control the rate at which your code runs**, so you can check and see what is going on:
  - Using the **Step Into** button, you can run your code **one line at a time** until you see an error occur
  - Using **Breakpoints**, you can allow the code to **run up until a certain point**, where you can have a closer look (very useful if you have a lot of lines of code, or loops with many iterations, such that stepping through every line would take a really long time)

Using the debug tools

- Other buttons on the Debug toolbar are useful:
  - The **Step Over** button is similar to Step Into, but will successfully execute a procedure call (and run the procedure in its entirety) before returning to the next line
  - The **Step Out** button will execute the remaining lines of the current procedure, and then stop after its closing line
  - The **Run Sub/Userform** button is particularly important: It will proceed to run the rest of the code, up until a breakpoint is encountered (if there is one)
- Another key when debugging is **checking on the values of variables** at various points during code execution, either by **hovering the mouse over them**, or using the **Locals Window** to see the values of all local variables

Next Topic:

Making objects and using interfaces