# Control of flow and modularization

Chapter 5 – **Code for making decisions**

- pp. 65-76
- **Exercises 5A & 5B**

Chapter 6 – **Using subroutines and functions**

- pp. 77-102
- **Exercises 6A, 6B, 6C & 6D**

# Chapter 5 – Code for making decisions

- Making a Case for branching
- Coding an If Then statement

# Chapter 5 – Code for making decisions

- For really simple operations, we can imagine writing many lines of code that **simply run in order** from beginning to end
  - However, this would only work in a situation where there were **no variations** in what those of lines were going to do, or what data we would have to use
- It is far **more common** to have to deal with situations where there are some **uncertainties or variations or different situations** to deal with for our code
  - For example, suppose we write some code to do something with the **features in a shapefile**. Is it a **point, line, or polygon** shapefile? Depending on which it is, maybe the code has to do something different …

# Chapter 5 – Code for making decisions

- One way to think of this is that our code contains the ability to respond to a **multiple-choice question** about **what it should do next**, depending on different situations that might be encountered
- A nice theoretical example in the text is the **decision** we make when **reaching a traffic light** … depending on if it is **green**, **yellow**, or **red**, we would do different things:
  - **green**– keep driving
  - **yellow**– proceed with caution
  - **red**– stop

# Chapter 5 – Code for making decisions

- A more **geographic example** might be some code that returns **some statistics about a shapefile**, which might differ if the shapefile contains **points**, **lines**, or **polygons**:
  - **points** – return the centroid of all points
  - **lines** – return the total length of all lines combined
  - **polygons** – return the total area of all polygons combined
- You can imagine that the **code to do each** of these things would be **quite different**, and that we need a way both to **distinguish** which of the three situations we encounter, and then to **run the appropriate lines of code**
- ArcGIS VBA provides us two approaches: **Case** statements and **If Then** statements

# Making a Case for branching

- You can use the **Case statement** to deal with this sort of multiple-choice situation by **making a Case** for **every possible choice**
- The trick here is to able to enumerate (in your head, often before the fact) **every Case that could be encountered**:
  - One of the skills you will develop as a programmer is the ability to **imagine all the possible values** some code might encounter at a given line before the fact, but even so, very often you will have to go back and later fix your code to deal with a situation you had not anticipated
- A catch-all for the unexpected is **Case Else**, which is how a Case statement deals with the situation where **none of the enumerated Cases apply**

# Making a Case for branching

- The first line (Select Case) **specifies what variable** is going to be used to **choose which branch** to follow
- Subsequent lines that begin with **Case**, followed by the **possible values**, are used to **make the decision** through an **exact comparison of values**:

```
Select Case strUser
Case “Mark”
        msgbox “Welcome Mark!”
Case “Dana”
        msgbox “Welcome Dana!”
Case “Braden”
        msgbox “Welcome Braden!”
Case Else
        msgbox “You are not an authorized
user!”
End Select
```

# Coding an If Then statement

- Alternatively, you can use the **If Then** statement structure to deal with your multiple-choice situation (rather than Case statements)
- The **difference** between Case statements and (the appeal of) If Then statements is that the If Then structure can deal with **more complicated situations**
- Unlike a Case statement where each **Case** is checked to see if it is an **exact match to a specified value**, the way that **If** statements work is on the **basis of logic**:
  - If the **logical expressions specified is true**, then that particular choice is the one selected, and **that chunk of code runs**

# Coding an If Then statement

- Each **If** or **ElseIf** line is used to specify a **logical condition that could occur**
  - One **tricky thing** is to make **each If and ElseIf logically exclusive from one another** … because if they are not, only the first situation that is evaluated to be true is going to run (an analogy to understand this: imagine a professor makes a multiple choice question where multiple answers are correct, and you are only allowed to choose one answer …)
- An **Else** section can be included to deal with any time a situation is encountered when **none of the If and ElseIfs specified are applicable**

# Coding an If Then statement

- Structurally, there are **no other significant differences**
  - Case can have several Cases and a Case Else
  - If can have an If and several ElseIfs and an Else
- Just about **anything you could do with Case** statements, **you could do with If Then** statements, although **not vice-versa**
- The real **power** of If Then statements is the ability to **combine comparison operators, logical connectors and functions** to specify a range of **complex situations**:
  - **Comparison operators**: >, <, <>, =, >=, <=
  - **Logical connectors**: AND, OR
  - **Functions**: IsNumeric, IsDate, IsString, IsNull, etc.

# Chapter 6 – Using subroutines and functions

- Calling a subroutine
- Passing values to a subroutine
- Making several calls to a single subroutine
- Returning values with functions

# Chapter 6 – Using subroutines and functions

- You will recall that we previously worked with **events** and **event procedures**:
  - An **event** occurs when a **user does something** (performs an action), e.g. a user clicks on a button: Associated with that button's click event is a number of lines of code that performs some action
  - We refer to the code associated with a **given user action** as an **event procedure**, i.e. when a user clicks on a CommandButton, then the CommandButton's click event procedure runs
- We are now going to work with **other blocks of code** that do something (subroutines and functions) that **differ from event procedures** by the **circumstances under which they are triggered**

# Chapter 6 – Using subroutines and functions

- An **event procedure** is triggered by an event; that is what causes it to start running
- A **subroutine** can be thought of as a procedure that runs when it is **called by another procedure** (whether an event procedure, or another subroutine)
- You can think of a **function** as being much like a subroutine (in that it is several lines of code that are written to do something in particular, that it is triggered by another calling procedure, etc.) with **one key difference**:
  - A **function returns a specific value** (when it is finished) to the line of code that called it → you can think of this as functions having **some result that it gives back** when finished

# Chapter 6 – Using subroutines and functions

- The appeal of building code using subroutines and functions is that it is **modular**:
  - It makes it **easier to keep complicated code organized** because it is broken up into chunks, each with a discrete purpose
  - This, in turn, makes it **easier to re-use individual modular pieces** of a larger codebase in different places
  - It also has the desirable effect of **making your code easier to debug**: If there is an error, and some particular functionality does not work, it should be pretty easy to identify the particular subroutine or function at fault because they are modular and organized by their purpose
- This modularization of code is a **very good practice for promoting efficiency**, both in code writing and running

# Calling a subroutine

- Using the example from the text, you get a subroutine to run with the **Call statement**:

```
Public Sub GetMessages()
  Call Message
End Sub
```

- Here, GetMessages is **calling another subroutine** called Message:

```
Public Sub Message()
  MsgBox “Geography is terrific”
End Sub
```

- We can expand on this idea with **one procedure calling several others** in a row, or a whole **series of procedures calling other procedures** … whatever our task requires

# Passing values to subroutine

- One of the consequences of this modular approach is that subroutines sometimes need to **pass values** to one another
  - e.g. suppose I have a subroutine that changes a layer in a map from being visible to invisible, it might be **convenient** for me to **pass that layer** to the subroutine **when I call it**
- Subroutines are capable of accepting an **argument** when they are called, which facilitates this passing of a value to the subroutine; this is **not required but often useful**
- Arguments are defined with a **name** and **data type**:
  - e.g. `Public Sub PrintMap (aPageSize As String)`
- When the subroutine is called, the **value is specified**:
  - e.g. `Call PrintMap (“Letter”)`

# Making several calls to a single subroutine

- There is **no impediment to calling the same subroutine from different places**, or calling the same subroutine **many times** in the service of performing some particular computing task …
- In fact, one of the reasons that the **modular design** approach is desirable is specifically to make this possible to do while **minimizing the amount of effort required** to make the functionality work

# Returning values with functions

- A function provides the **other half of the capability to pass values/objects back and forth** between our modular chunks of code:
  - Subroutines **accept an argument as input**
  - Functions accept an argument as input **AND** **return a value as output**
- The **syntax looks a little different** because of this
  - For example, suppose we have a **function named InputBox**, we can make the function run, passing it the value "Enter a Parcel Value", AND assign its output value to a String called strValue using the line:

```
strValue = InputBox("Enter a Parcel Value")
```

# Next Topic:

Using loops and debugging code