Chapter 1 – Programming with objects

- Object-oriented programming
- Visual Basic for Applications (VBA)
- ArcObjects

Object-oriented programming

- **Object-oriented programming** (OOP) is a programming approach that uses "objects" to design applications and computer programs
  - This is contrast to **modular or procedural programming** that you might be familiar with, where **lines of code are numbered** and run in sequence
- Even though it **originated in the 1960s**, OOP was not commonly used in mainstream software application development until the 1990s
- Today, **many popular programming languages** (such as Java, JavaScript, C#, VB, .Net, C++, Python, PHP, Ruby and Objective-C) support OOP

Object-oriented programming

Important OOP terminology:

- Object - **Anything** that can be 'seen' or 'touched' **in the software programming environment** . Objects have attributes (properties) and behaviors (methods)
- Properties - Attributes are **characteristics that describe** objects
  - e.g. *Text.Font = Arial*
- Methods (behaviors) - An object's methods are operations that either the **object can perform** or that **can be performed upon** the object,
  - e.g. *Table.AddRecord*

Object-oriented programming

Important OOP terminology:

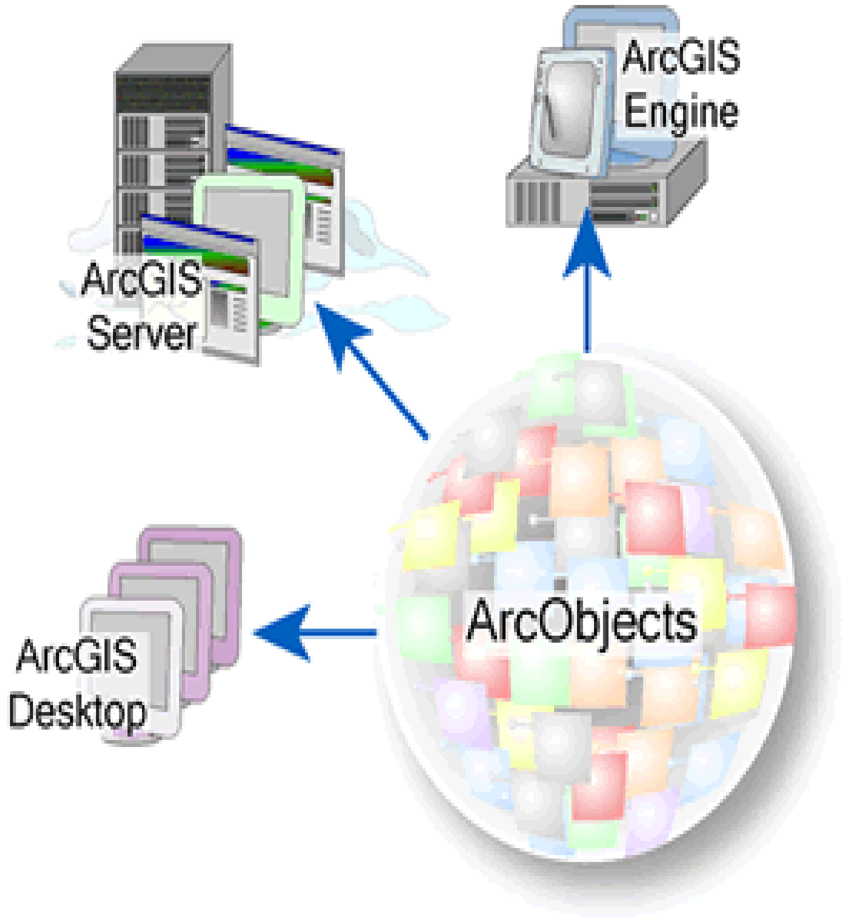
- Class - A **pattern or blueprint** for **creating an object**. It contains **all the properties and methods** that describe the object
- Instance - The **object you create** from a class is called an **instance** of a class
- Distinguishing an **object/instance vs. a class**, examples:
    - Cookie vs. a cookie-cutter
    - Car vs. the blueprint for manufacturing the car

Visual Basic for Applications (VBA)

- In ArcGIS, we will **interact** with VBA using the **Customize dialog box** and the **Visual Basic Editor**
  - Through the **Customize dialog box**, we can **change the set of controls** that appear in the interface (like menu items, buttons, and controls)
  - Through the **Visual Basic Editor**, we can **work on the VBA code** associated with the controls, so that when someone clicks on a control in the user interface … something happens!
  - For each **event** that might occur in the user interface (e.g. when a user clicks on a control) there is a **particular set of instructions** that are executed. These are **organized into procedures**, which break the code into modular chunks, that can call each other etc.

ArcObjects

- ArcObjects are **platform independent software components**, written in C++, that provide services to support GIS applications, either on the desktop in the form of thick and thin clients or on a server for Web and traditional client/server deployments
- Because this architecture **supports a number of unique ArcGIS products** with specialized requirements, all ArcObjects are designed and built to support a multi-use scenario

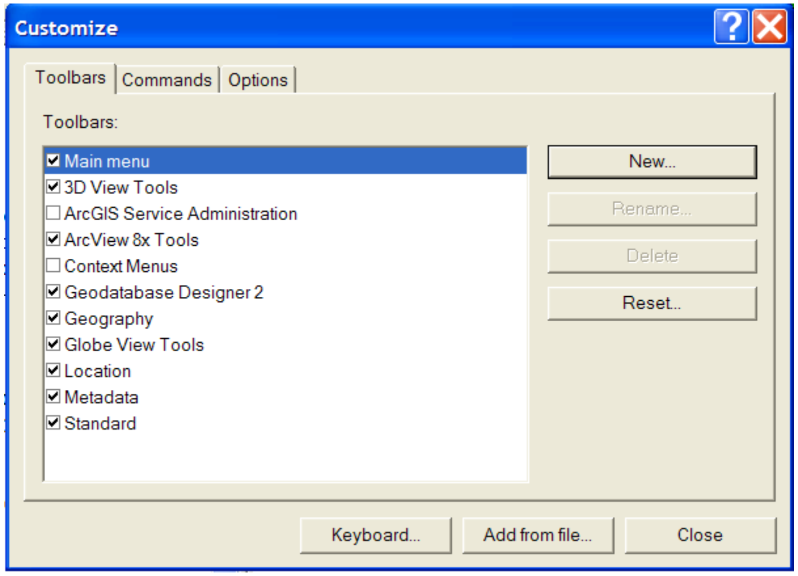


Chapter 2 – Building a custom application

- Organizing commands on a toolbar
- Making your own commands
- Storing values with variables

Organizing commands on a toolbar

- The **graphical user interface** (GUI) of ArcMap and ArcCatalog is made up of toolbars, menus, and commands
- Toolbars **contain** commands or menus
- Commands can come as **either buttons or tools**
  - We will learn the **distinction** between these later
- Menus provide **pull-down lists** of commands or of other menus
- To **change** the user interface, the Customize Dialog Box is used:



Toolbars:

- ☑ Main menu
- ☑ 3D View Tools
- ☐ ArcGIS Service Administration
- ☑ ArcView 8x Tools
- ☐ Context Menus
- ☑ Geodatabase Designer 2
- ☑ Geography
- ☑ Globe View Tools
- ☑ Location
- ☑ Metadata
- ☑ Standard

Making your own commands

- **Where** do these customizations get saved?
- **Projects** are files where your UIControls and VBA code are stored
- There are **three types** of projects:
  - Map documents (.mxd files)
  - Base templates (.mxt files)
  - Normal template (normal.mxt)
- When we save code in a **map document**, it is **only available** when we open that map document
- The **normal template** can be used to store customizations that are **available with any project** that the user opens

Storing values with variables

- What is a **variable**? A **named storage location** that **contains data** that **can be modified** during a program
- Each variable **has a name that uniquely identifies it** within its scope. Usually, a **data type** is specified
  - Some data types: byte, boolean, integer, long, currency, decimal, single, double, date, string, object …

Type (Bytes)	***Sample Value***
String(10+Length)	*Elm Street*
Boolean(2)	*True or False*
Date(8)	*1/1/200 to 12/31/9999*
Byte(1)	*0-255*
Integer(2)	*-21768 to 32767*
Long(4)	*-2,147,483,648 to 2,147,483,648*
Single(4)	*1.401298E-45 to 3.402823E38 positive*
Double(8)	*1.79769313486232E308 maximum*

Storing values with variables

A convention for naming variables:

- One way to **help keep track** of what **kind of variable** any given variable is makes use of **certain prefixes** for **particular data types**:

Type	*Prefix*	*Example*
String	*str*	*strAddress*
Boolean	*bln*	*blnStatus*
Date	*dat*	*datBirth*
Byte	*byt*	*bytAge*
Integer	*Int*	*intPopulation*
Double	*dbl*	*dblLatitude*

Storing values with variables

- Variables get ‘called into existence’ by **declaring** them:
- In many cases, you will **explicitly declare** them using the Dim statement:
  - *Dim txtGreet as String*
  - *Dim intCount as Integer*
- Another line can be used to **set their value**, once declared:
  - *txtGreet = “Hello”*
  - *intCount = 42*
- In some cases, you will **implicitly declare** them, and set their value simultaneously
  - *txtAlternateGreet = “Howdy”*

Using dialog boxes and objects

Chapter 3 – **Creating a dialog box**

- pp. 37-50
- **Exercise 3**

Chapter 4 – **Programming with objects**

- pp. 51-64
- **Exercises 4A& 4B**

Chapter 3 – Creating a dialog box

- Using controls to build a form

Using controls to build a form

- You can **build dialog boxes** to suit **whatever purpose** you have in mind, and add **whatever controls** are needed
- In VBA, these are referred to as **Forms**, or **UserForms**
- Within the **Visual Basic Editor**, we can craft a Form by **dragging and dropping controls** onto it
  - Choose from **elements/controls** such as Labels, Textboxes, ComboBoxes, Listboxes, Checkboxes, OptionButtons, ToggleButtons, CommandButtons, Images, and Frames
- Each and every element/control you add to your Form has **many properties**
  - You can set their **initial values** in the **Editor**, and control their state later through **VBA code**

Using controls to build a form

- Once you build the skeleton/appearance of a Form in the Editor, you need to **write code to make it do things** once a user clicks on controls (or performs some other action)
- This is organized into **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

Chapter 4 – Programming with objects

- Programming with methods
- Getting and setting an object’s properties

Chapter 4 – Programming with objects

- We work with objects by **setting their properties**, and **calling their methods**, using the **"object dot property"** syntax:
  - *Object.Property*
  - This is also known as **reverse Polish notation**
- We can think of properties a little like variables, in that **they describe an object**, and we can both **get (find out their current value)** and **set (change their value to something else)** them → Properties as **adjectives**, that describe the object
- We can think of methods as **things that objects can do** → Methods as **verbs**, that make the object do something

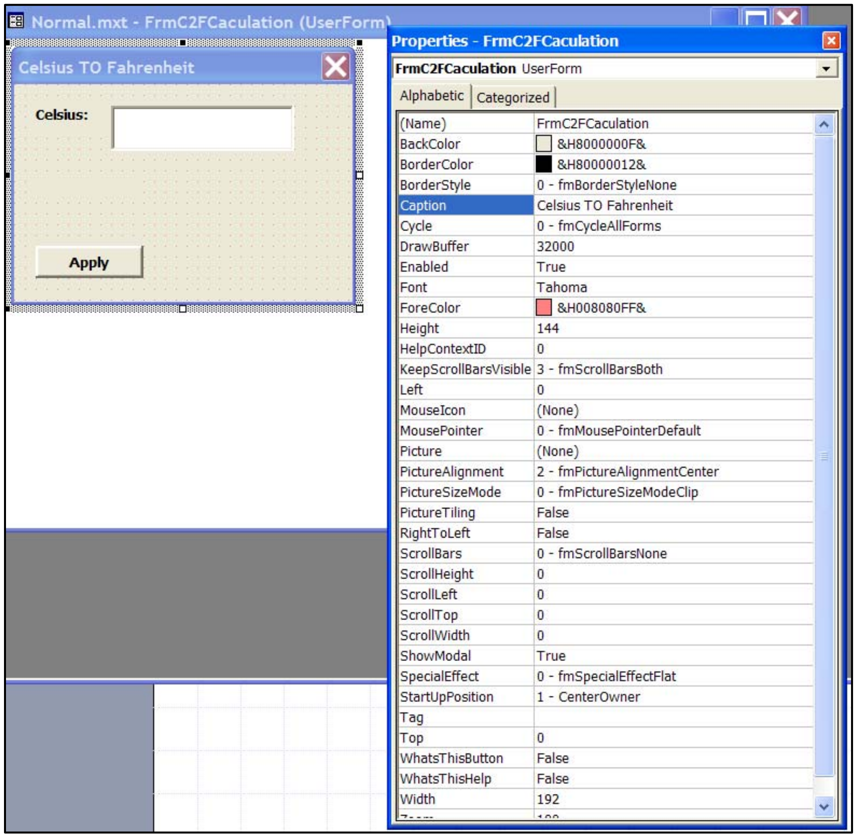
Programming with methods

- The textbook gives a series of **examples** with a spaceship, which boil down to *Object.Method*, where the **object** is the **spaceship**, and **method** is **something we expect it could do** (*Atlantis.WarpSpeed*, *Atlantis.Shields Down* etc.). Two key things to notice:

1. The methods in the fictitious example need to be things that the object can do. In real VBA code, the **methods** must be **defined** for an **object of that type**
2. Some methods have **arguments**, which **specify how** to perform the method *(Atlantis.Shields Down)*, and even multiple arguments, **separated by commas** (*Atlantis.BeamUp Andrew, Thad, Michael)*

Getting and setting an object’s properties

- In the Chapter 3 exercise, we will work with the properties of the controls we create, and **set them initially through the Editor interface**, where we can see all the properties of each control
- For example, here’s a Form with an InputBox that will convert a value from Celsius degrees to Fahrenheit degrees:



(Name)	FrmC2FCaculation
BackColor	□ &H8000000F&
BorderColor	■ &H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	Celsius TO Fahrenheit
Cycle	0 - fmCycleAllForms
DrawBuffer	32000
Enabled	True
Font	Tahoma
ForeColor	🟥 &H008080FF&
Height	144
HelpContextID	0
KeepScrollBarsVisible	3 - fmScrollBarsBoth
Left	0
MouseIcon	(None)
MousePointer	0 - fmMousePointerDefault
Picture	(None)
PictureAlignment	2 - fmPictureAlignmentCenter
PictureSizeMode	0 - fmPictureSizeModeClip
PictureTiling	False
RightToLeft	False
ScrollBars	0 - fmScrollBarsNone
ScrollHeight	0
ScrollLeft	0
ScrollTop	0
ScrollWidth	0
ShowModal	True
SpecialEffect	0 - fmSpecialEffectFlat
StartUpPosition	1 - CenterOwner
Tag	
Top	0
WhatsThisButton	False
WhatsThisHelp	False
Width	192

Chapter 5 – Code for making decisions

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

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

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”)
```

Chapter 7 – Looping your code

- Coding a For loop
- Coding a Do loop

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:

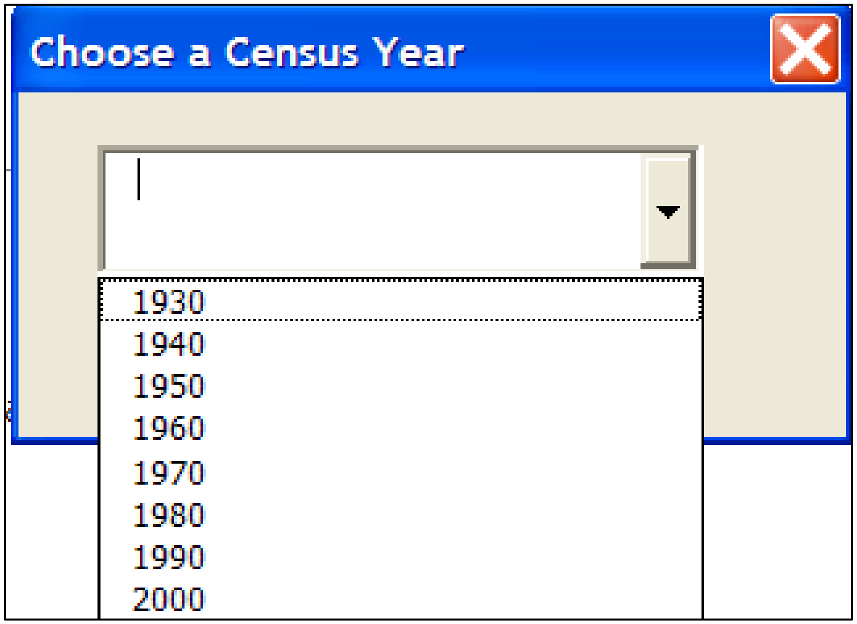
```
 For i = 1 To 10
```

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

{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 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

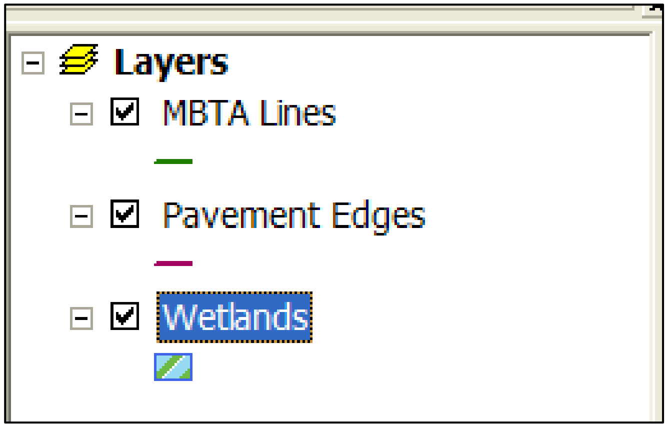
    `Exit Do`

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 – 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

Chapter 9 – Making your own objects

- Creating classes
- Creating objects

Creating classes

- Think of a class as a **container full of properties and methods**; that container has to be **stored somewhere**
- A class that you create gets **stored** in a particular kind of **code module** called a **class module**; once again, we have a decision to make about **where that class module will be stored** (like any customization we develop):
  - We could save it in a **map document**, or **normal.mxt**, or in a **base template**
- We create **properties** for our new class in its module by **declaring them as variables** (outside of any procedure)
- We create **methods** by **writing a subroutine or function** in the class module

Creating classes - properties

- We create **properties** for our new class in its module by **declaring them as variables** (outside of any procedure):

```
Public Value As Currency
Public Zoning As String
```

- Unlike all the code we have written so far, these are **not inside any particular procedure**, and we need to **use the Public keyword** (instead of the Dim keyword) to make them available to **any** procedure that is present in our class' module
- An alternative method for creating properties uses what is known as **property procedures**, but this is beyond the scope of what we will be doing

Creating classes - methods

- We create **methods** by **writing a subroutine or function** in the class module
  - Recall that **functions return a value**, so we would choose a function over a subroutine if we need to do so
- We **name** the subroutine or function according to the name we want to use to **call it in code later**, and again use the **Public** keyword to ensure that it is **available to any procedure in the class module**, for example:

```
Public Function CalculateTax() As Currency

End Function
```

makes a .CalculateTax method that returns a value of the type Currency

Creating objects

- With an **intrinsic variable** (like an integer), we can **declare and set** the variable using:

```
Dim X As Integer
X = 365
```

- For an **object variable**, the declaration line looks the same, but there is a **small difference** in the setting line:

```
Dim E As Elephant
Set E = New Elephant
```

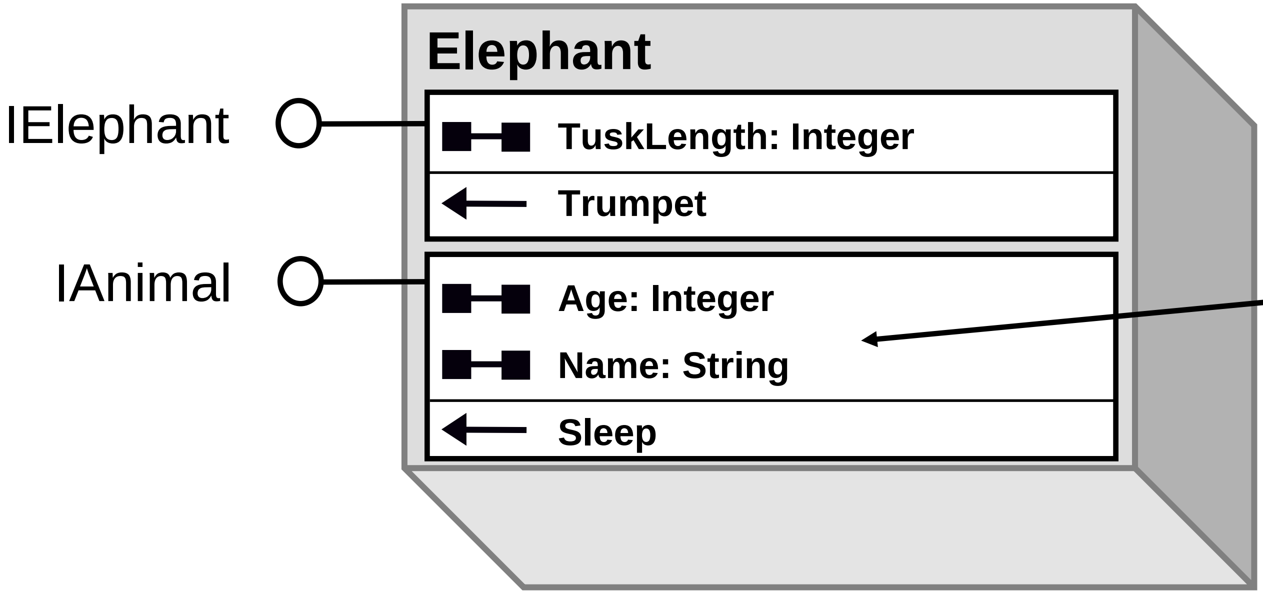
- The line used to set an object variable has to **begin with the Set keyword**, and must have the **New keyword after the equals sign** to denote the setting of a new object
- Getting/setting properties and using methods is the **same**

Chapter 10 – Programming with interfaces

- Using IApplication and IDocument
- Using multiple interfaces

Chapter 10 – Programming with interfaces

- In object-oriented programming, **inheritance** is a way to form new classes using the **characteristics of classes** that have **already been defined**
- The new classes, known as **derived classes**, take over (or **inherit**) properties and methods (and interfaces) of the pre-existing classes, which are referred to as **base classes**



Here, the Elephant class might **inherit from the Animal Class**, including the two properties, one method AND the **interface** (IAnimal) shown

Chapter 10 – Programming with interfaces

- When we **instantiate a COM object** with interfaces, we **specify what interface we will be using** right up front
- Recall when we were working with the **Elephant** class (and it was a simple object without interfaces), the declaration line looked like this:

  `Dim E As Elephant`
- Now that we have an Elephant class with **both an IElephant and IAnimal interface**, we have to **specify** which we are going to use:

  `Dim E As IElephant`
- The **naming convention** for interfaces is to name them ISomething (e.g. IAnimal, IApplication, IDocument)

Using IApplication and IDocument

- When we start ArcMap and open a map document, we can always count on there being **two objects that already exist**:
  - An **Application** object variable named **Application**
  - An **MxDocument** object variable named **ThisDocument**
- As ArcObjects developed in the COM framework, these objects naturally have interfaces, known as **IApplication and IDocument** respectively
- Often, you will write code that **begins with these objects** (and interfaces) and **navigate to other objects** (and interfaces) from these [more on this in the next section and in Chapter 11]

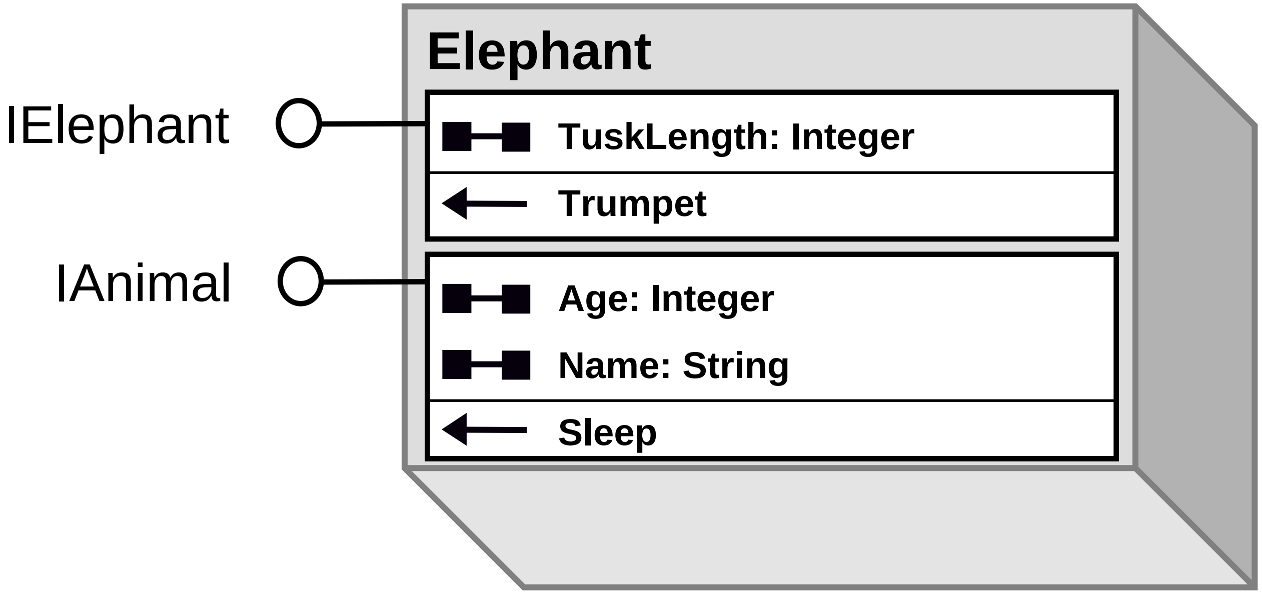
Using multiple interfaces

- Once you start working with **objects with multiple interfaces**, you have to keep track of what you are doing / **make sure you have the right interface** for the properties and methods you need
- Quite often, you will have a variable declared for an interface for an object and decide that you need another interface, and need to write the code to switch
- Suppose we created an **Elephant object** using the IElephant interface, and set its TuskLength:

```
Dim pElephant1 As IElephant
Set pElephant1 = New Elephant
pElephant1.TuskLength = 6
```

Using multiple interfaces

- This makes sense, because the **TuskLength** property is **located on the IElephant interface**:



- But what if we now want to **set our new Elephant's Name**; we cannot do so on the IElephant interface
  - We **need the IAnimal interface**, because that is where the Name property is located

Using multiple interfaces

- First, we must **declare a new variable** that points to the **IAnimal interface**

  `Dim pAnimal1 As IAnimal`

- Now, we can use **Set** keyword to set our new **pAnimal1 to be equal to be our existing pElephant1** to indicate it is the **same object** (but with a different interface):

  `Set pAnimal1 = pElephant1`

- Now, we have access to the interface we need to set our Elephant's **Name**:

  `pAnimal1.Name = “Dumbo”`