Map symbology and ArcCatalog

Chapter 15 – **Setting layer symbology**

- pp. 263-293
- **Exercises 15A, 15B & 15C**

Chapter 16 – **Using ArcCatalog Objects in ArcMap**

- pp. 295-314
- **Exercises 16A & 16B**

Chapter 15 – Setting layer symbology

- Setting layer color
- Setting layer symbols
- Creating a class breaks renderer

Chapter 15 – Setting layer symbology

- Let's take a moment to think about the **data files** that are used when we **make a map using ArcGIS**:
  - The **spatial data** itself is stored in a **variety of possible formats** (shapefiles, coverages, in geodatabases, etc.), but these just contain **information about location and attributes**
  - The **project file** stores information about **what spatial data to include** in the map and **how to symbolize it**
- You can think of a **project file** like a **recipe**, and the **spatial data files** it references as **ingredients** it uses
  - There is **nothing specific about how to symbolize** the data stored **in the spatial data files**
  - There is **no spatial data stored in the project** file itself

Chapter 15 – Setting layer symbology

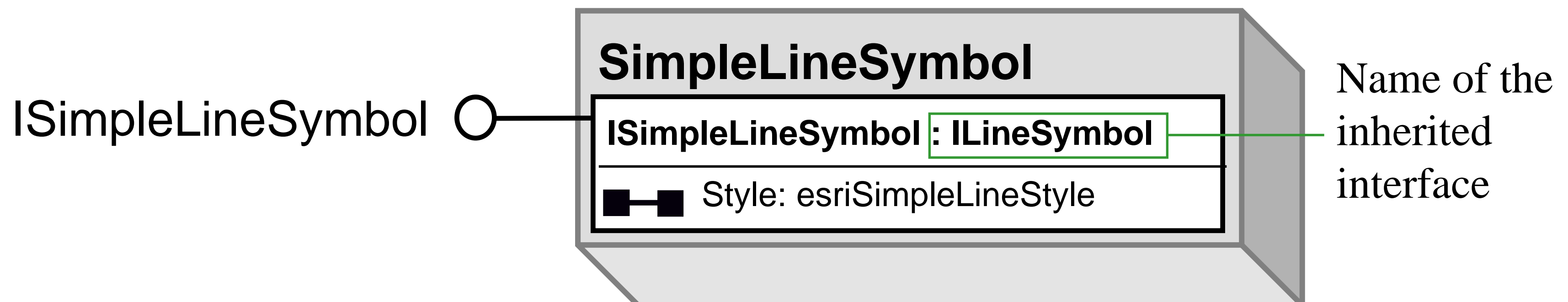
- This is a **very efficient** setup:
  - When we make a map, **we don't change the underlying spatial data files**
  - This means they can act as **'ingredients' for lots of maps**
- The **symbolization decisions** sit within the **project document**
- As we know from our experience in this course up to this point, we can extensively **customize** the functionality in projects by **changing the GUI elements** and **coding VBA**
- Thus, we can **use code to set the symbology** of layers in our maps, which in many cases allows us to **do things that would be difficult for the user to do manually** (?)

Chapter 15 – Setting layer symbology

- When we point ArcGIS towards a **spatial data file** to **add to a map**, it gets **added as a layer**
- Through the **GUI**, we use **legends** to **specify the symbology** that controls **how that layer is shown**
- Using VBA code, we use objects from the **renderer** class to control the symbology (when a user does this in the GUI, they are really manipulating an underlying renderer)

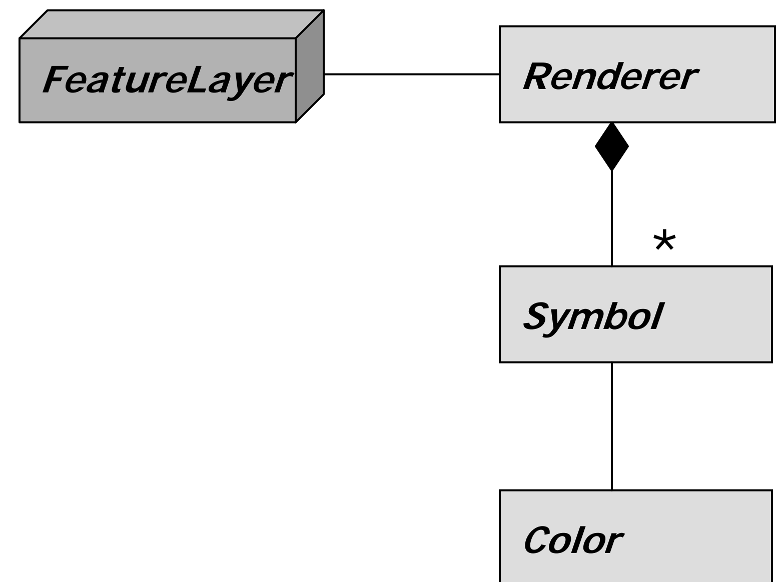
Chapter 15 – Setting layer symbology Interface inheritance

- Recall back in Chapter 10 when we learned of **class inheritance**: Derived classes can take over (or **inherit**) **properties, methods, and interfaces** of the pre-existing classes, which are referred to as base classes
- In this chapter, we look at a **form of inheritance** that is a **subset** of the above, called **interface inheritance**:
  - The properties and methods **associated with a particular interface** are inherited, but properties and methods from **other interfaces on the same class** **ARE NOT** inherited here



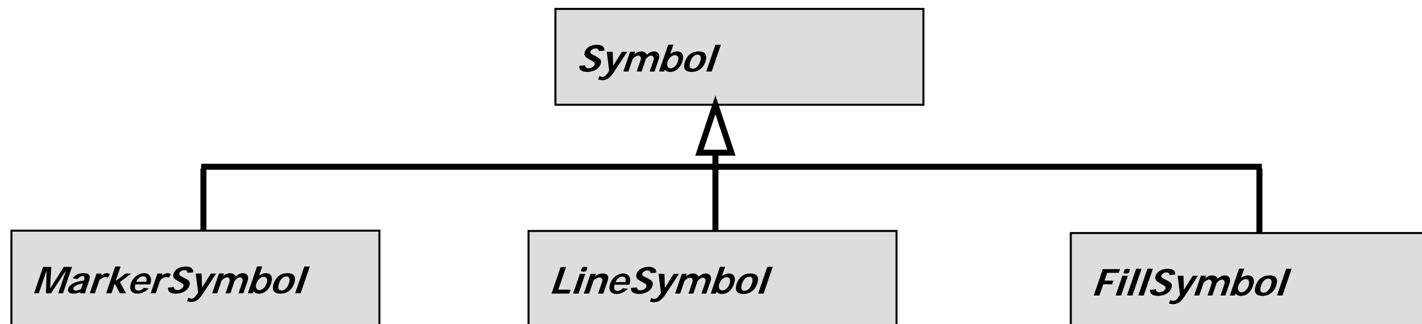
Setting layer color

- By **default**, when a layer is added to a map using the GUI, it is a symbolized with a **single random color**
- This is the **default renderer** assigned to the layer
- As an alternative, we can **write code** to make use of **another renderer**
  - Every **feature layer** has a renderer
  - Renderers are **composed of symbols**
  - Every symbol has a **color** (different kinds of symbols will have other sorts of characteristics as well)



Setting layer color

- The **Symbol abstract class** has many **subclasses**; the basic ones are:
  - The **MarkerSymbol** class for **points**
  - The **LineSymbol** class for **lines**
  - The **FillSymbol** class for **polygons**
- These, in turn, are abstract classes that each have their own **subclasses** (see page 266 of the text)



Setting layer color

- The **usual approach** applies here: **Symbols** and their **Colors** are **declared** with the `Dim` keyword, **created** with the `New` keyword, and **properties** are set with the **object.property** syntax
- Every **FeatureLayer** has one **FeatureRenderer**; FeatureRenderer is an abstract class with **eight subclasses** for the **various legend types**:
  - UniqueValueRenderer
  - DotDensityRenderer
  - SimpleRenderer
  - ClassBreaksRenderer
  - ScaleDependentRenderer
  - ChartRenderer
  - BitUniqueValueRenderer
  - ProportionalSymbolRenderer

Setting layer symbols

- In addition to **specifying the characteristics** of symbols yourself, you can also **draw upon pre-existing sets of symbols**
- ArcGIS symbols are stored in the **Style Manager**, grouped by style gallery classes that contain individual style gallery items
- These are **designed to be used for common thematic maps** of various types
- This is as simple as **finding the styles** you wish to use **in the Manager**, and then **navigating** the associated objects and classes (known as **Enums**, from enumerations) to **obtain those symbols for your use**

Creating a class breaks renderer

- A particularly **useful application** of manipulating legends / renderers by code is to **create them with particular ranges of associated attribute values**
- This kind of renderer is a **ClassBreaksRenderer**, and by working with these through VBA, you can **specify the exact ranges of attribute values** associated with particular symbols
- You might use this approach if you are **making many similar maps, and want to ensure they all have precisely the same legend** (and ranges of values associated with particular symbols)

Chapter 16 – Using ArcCatalog objects in ArcMap

- Adding layer files to ArcMap
- Making your own Add Data dialog box

Chapter 16 – Using ArcCatalog objects in ArcMap

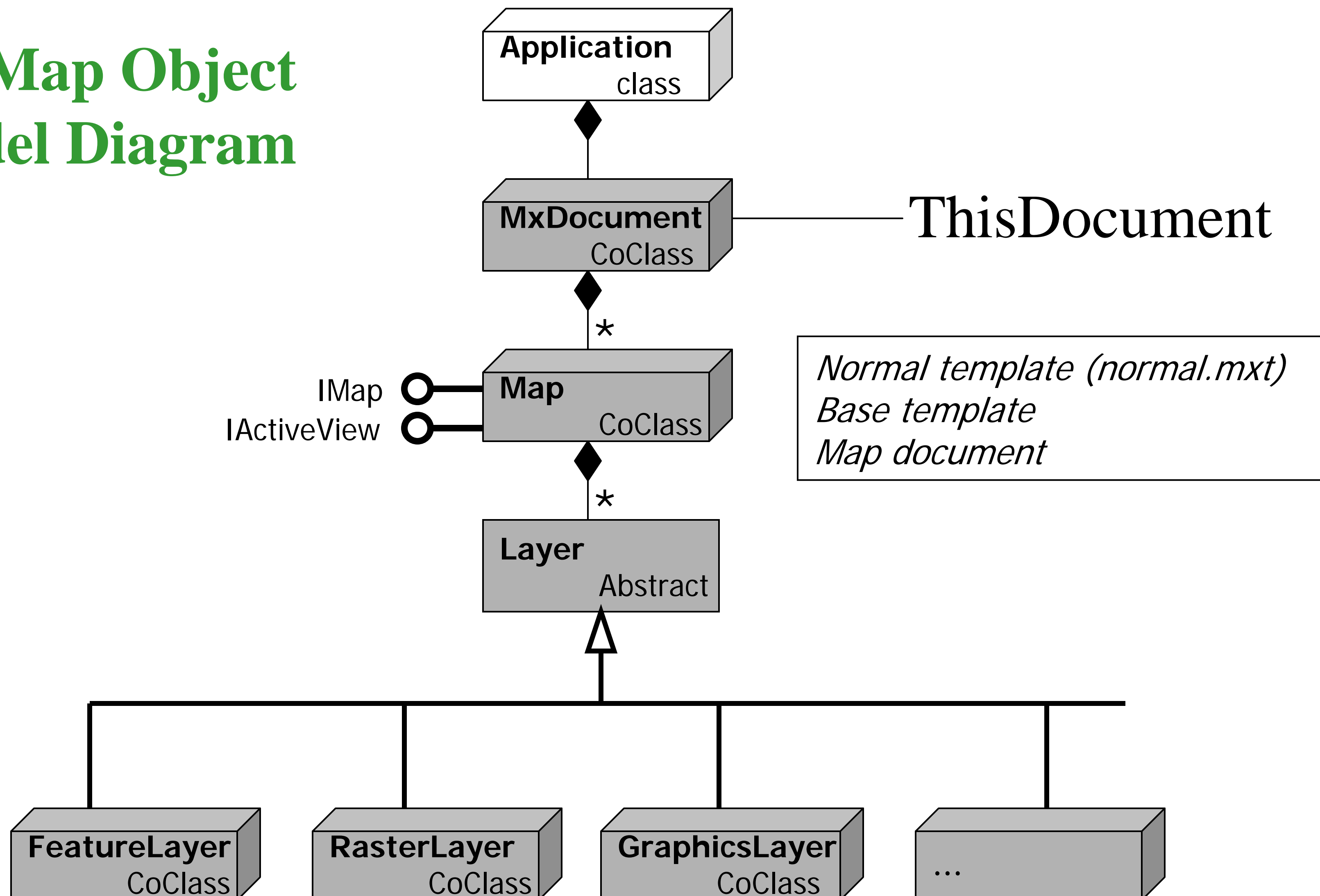
- You are familiar enough with ArcGIS to know that its functionality is **broken up into different applications**:
  - **Map-making** happens in **ArcMap**
  - **Management of data files** happens in **ArcCatalog**
- Even if you're going to **develop VBA code primarily for ArcMap**, you'll need to **work with some ArcCatalog classes and objects to manage data files** (that you might to a map, for example)
- To be totally accurate, **all ArcObjects** are **available** in **all ArcGIS applications**, although some are **associated** with the **object model diagrams** of **particular applications**

Chapter 16 – Using ArcCatalog objects in ArcMap

- The ArcCatalog object model has **similar starting points** to that of ArcMap
  - There is an ArcCatalog **Application object** named **Application**
  - There is a **GxDocument object** named **ThisDocument**
- One **key difference** is the location **where customizations can be stored**
  - **Unlike ArcMap** with its options (the project .mxds, base templates and the normal.mxt template), **ArcCatalog has only one place where customizations are stored**, its own **normal.gxt template** (this presents some problems in conveniently distributing ArcCatalog customizations)
- Just as many objects in **ArcMap** have the **Mx prefix** in their name, **Gx** is the **common prefix for ArcCatalog**

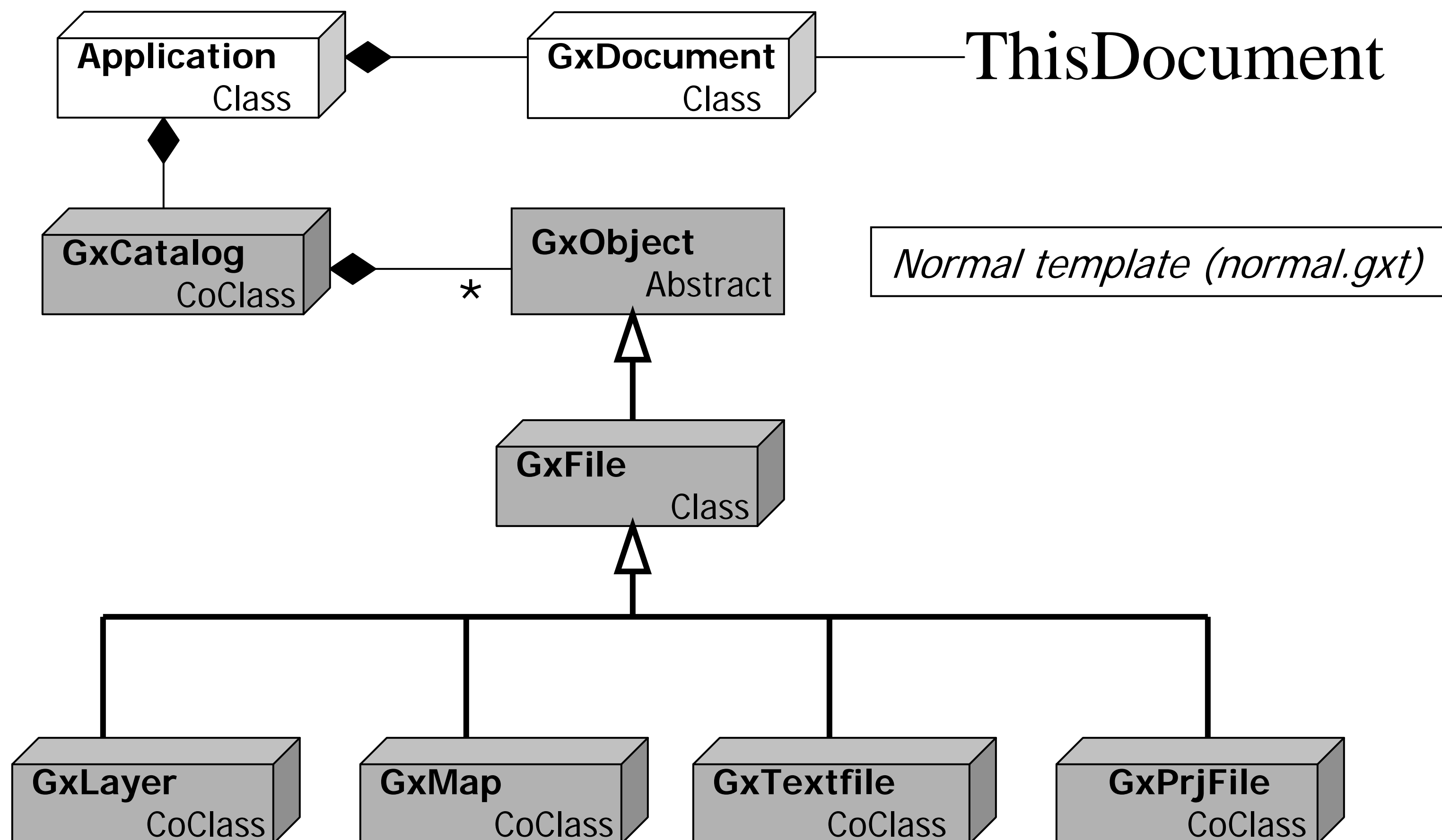
Chapter 16 – Using ArcCatalog objects in ArcMap

ArcMap Object Model Diagram



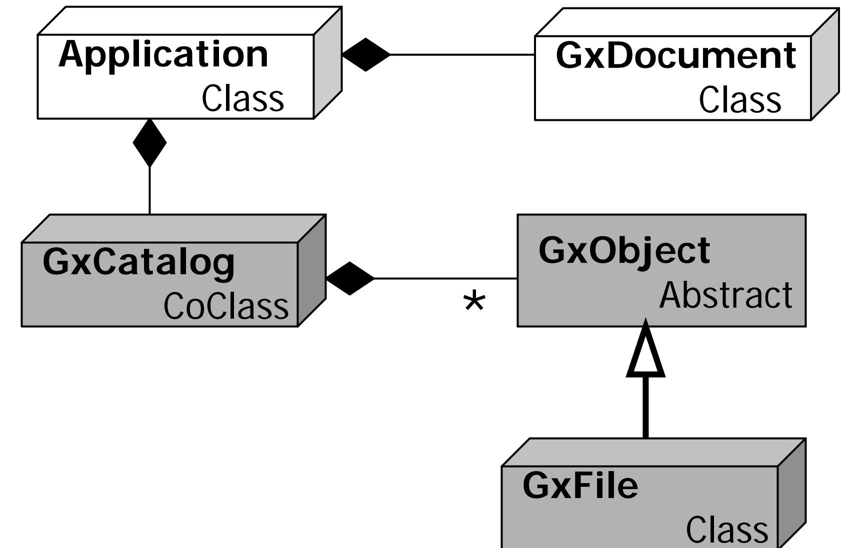
Chapter 16 – Using ArcCatalog objects in ArcMap

ArcCatalog Object Model Diagram

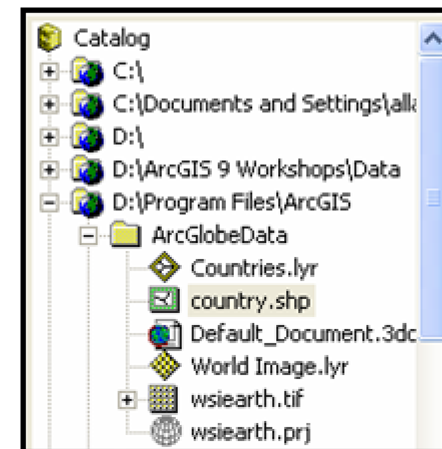


Chapter 16 – Using ArcCatalog objects in ArcMap

- The ArcCatalog Application is composed of **GxCatalog objects**, which in turn are composed of **GxObjects**
- A GxObject is **any file, folder, disk connection, or other object you can click on in the tree view** shown in the left-hand pane of ArcCatalog

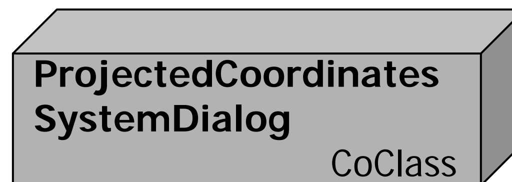


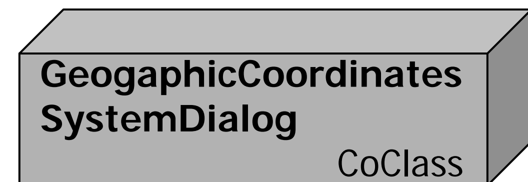
Several kinds of **GxObjects**, shown in the tree view

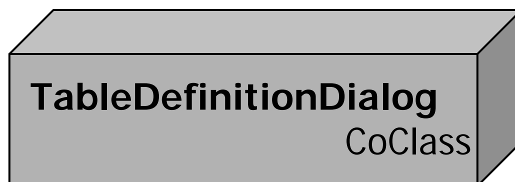


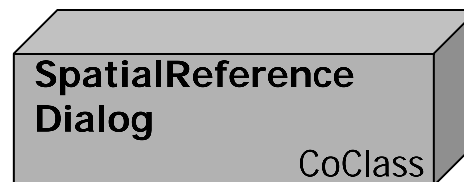
Chapter 16 – Using ArcCatalog objects in ArcMap

- There are **five further coclasses** in the ArcCatalog object model diagram that **represent dialog boxes**
- Each has its uses, but particularly important to us is the **GxDialog**, which gives us the **capability to make customized dialog boxes for specifying files to be opened or saved**



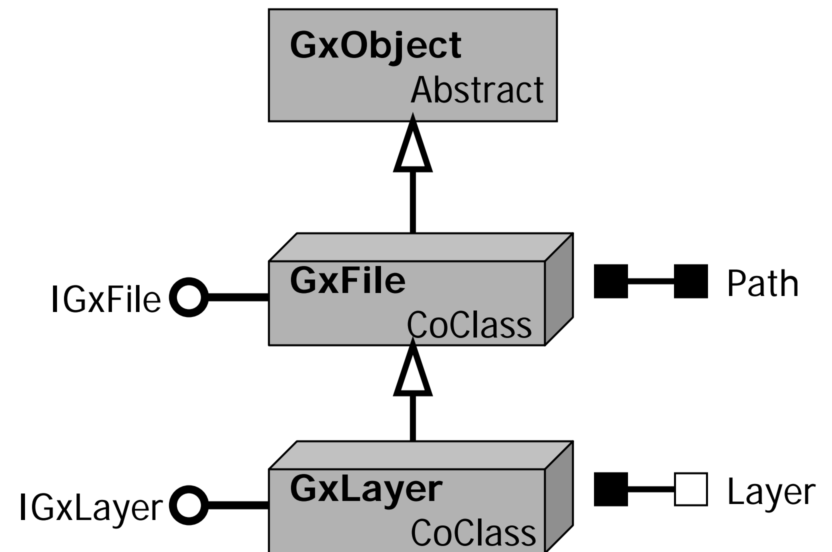






Adding layer files to ArcMap

- A **layer file (extension .lyr)** acts as an intermediate between a spatial data source and the Map document: It **stores information about symbology, the path to the data set** etc.
  - This **simplifies adding a layer to a Map with a particular symbolization**; it is all set up already
- A **GxLayer** is a **GxFile**, and both are **GxObjects**, and as they are **coclasses**, either can be **created directly**
- To **create one from a file**, use GxFile's **path property**:



Adding layer files to ArcMap

- Set up the GxLayer object by **declaring and creating** it:

```
Dim pGxLayer As IGxLayer
Set pGxLayer = New GxLayer
```

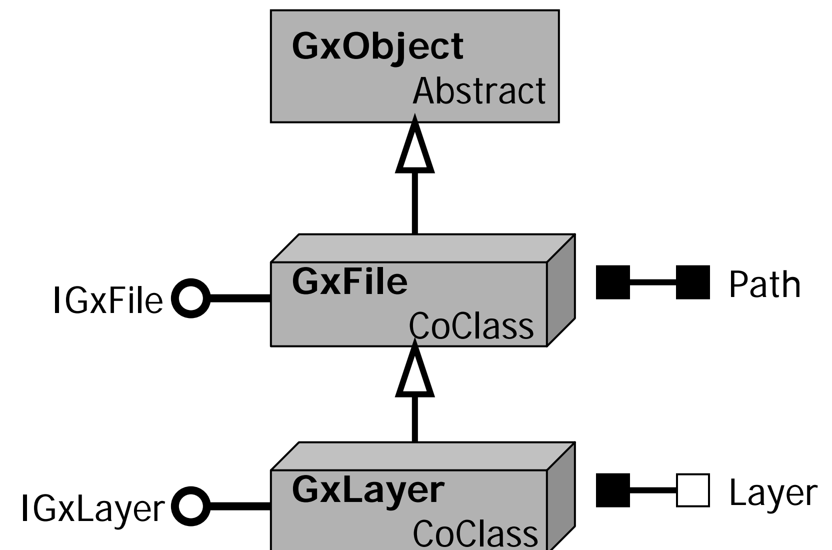
- Now get the **IGxFile interface** to use the **path property**:

```
Dim pGxFile as IGxFile
Set pGxFile = pGxLayer
```

- Now, with a **known path** to our layer file, **set the path property**:

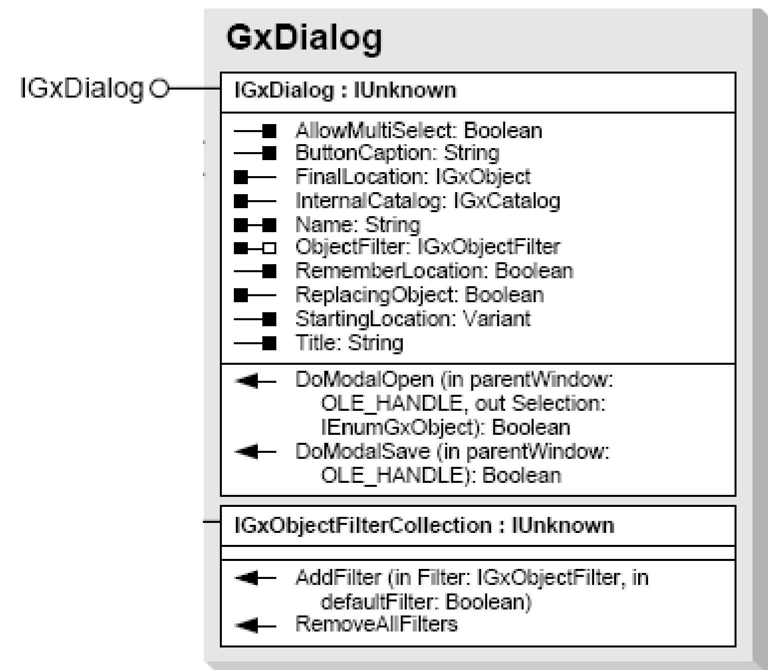
```
pGxFile.Path = "C:\the.lyr"
```

- The layer is now **ready to use** with our pMxDoc



Making your own Add Data dialog box

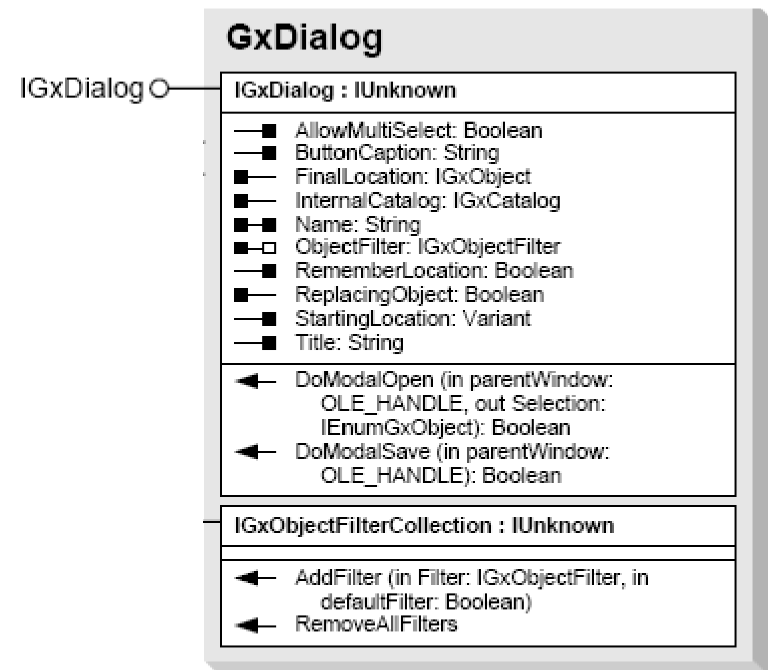
- In many cases, rather than having a known path to the data we want to add, instead we give the user the chance to **navigate to the correct directory** and **select the data source using a dialog box**
- The **GxDialog** is designed just for this purpose: It allows to create a **file selection dialog box** that we can **customize in various ways** (e.g. to allow specific file types to be selected, single or multiple files selected, what the title and buttons say, what directory it opens in etc.)



Making your own Add Data dialog box

- **For example**, to create a GxDialog titled “Add Data”, that starts in “Catalog”, with a Button that says “Add”, and only allows the selection of a single file:

```
Dim pGxDialog As IGxDialog
Set pGxDialog = New GxDialog
pGxDialog.ButtonCaption = "Add"
pGxDialog.StartingLocation = _
"Catalog"
pGxDialog.Title = "Add Data"
```



- We can further customize the GxDialog by **restricting the type of files** it can be used to open using an **ObjectFilter**

Making your own Add Data dialog box

- There are a **wide variety of types** of **GxObjectFilter** to suit whatever you need your GxDialog to get
- For example to allow our GxDialog to **just open layers**:

```
Dim pLFilter as IGxFilterLayers
Set pLFilter = New GxFilterLayers
```

- We then **set** our GxDialog's **ObjectFilter property** accordingly:

```
Set pGxDialog.ObjectFilter = _
pLFilter
```

Making your own Add Data dialog box

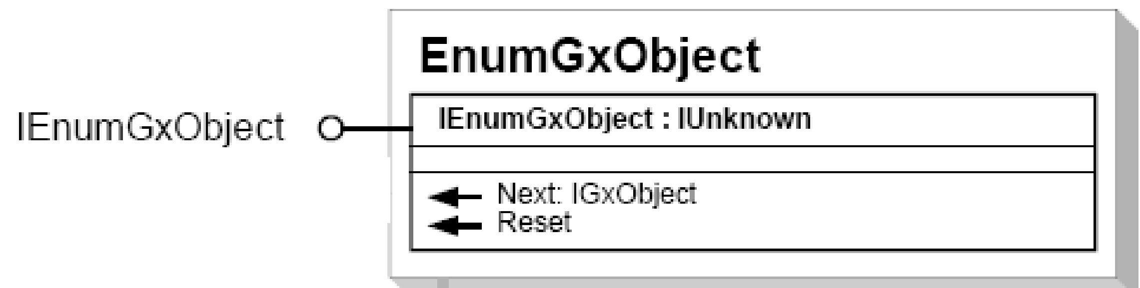
- We need to **create one more object** to use our GxDialog
  - An **EnumGxObject** gives **access to the members** that are **enumerated** through the **ArcCatalog tree view**
  - Essentially, this gives us **a way to get the files that the user chooses through** the dialog

```
Dim pLayerFiles As IEnumGxObject
```

- We can now **open** the GxDialog:

```
pGxDialog.DoModalOpen 0, pLayerFiles
```

- We can now **retrieve the files** from the **Enum object** by using its **Next method**

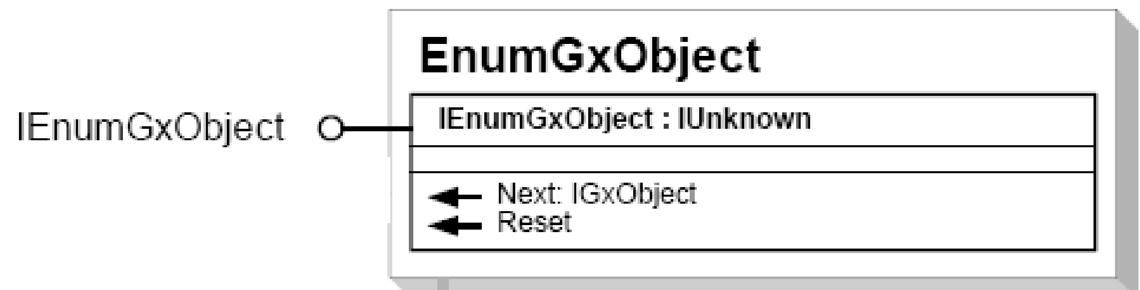


Making your own Add Data dialog box

- The **result** of the Next method will be a **GxObject**, so we need to declare it as such, and then we can **set its value** by **getting the value from the EnumGxObject** using the **Next method**:

```
Dim pLayerFile As IGxObject
Set pLayerFile = pLayerFiles.Next
```

- Since we used **AllowMultiSelect = False**, we know our EnumGxObject is going to **contain just a single value**
- Otherwise, we could **loop**, and apply the **Next method repeatedly** to get **multiple values**



Next Topic:

Displaying and selecting features