Making objects and using interfaces

Chapter 9 – **Making your own objects**

– pp. 133-145

– **Exercises 9A & 9B**

Chapter 10 – **Programming with interfaces**

– pp. 147-170

– **Exercises 10A & 10B**

Chapter 9 – Making your own objects

- Creating classes
- Creating objects

Chapter 9 – Making your own objects

Recall some of our 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

Chapter 9 – Making your own objects

- So far, we have used ArcGIS VBA to work with **classes** (and instantiate objects of those classes) **designed by someone else**
- Part of the power that developing for ArcGIS in VBA is the **ability to create your own classes**, and thus create instances of **objects with the characteristics you need**
- Think back to your introductory GIS classes: We use GIS to create **models of reality**:
  - With **spatial representations and attributes** (which can be **object properties** in this context), we can build these models
  - By creating objects with particular **methods**, we can represent how things in our reality **interact with one another** in a model

Chapter 9 – Making your own objects

Recall some of our 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*

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:

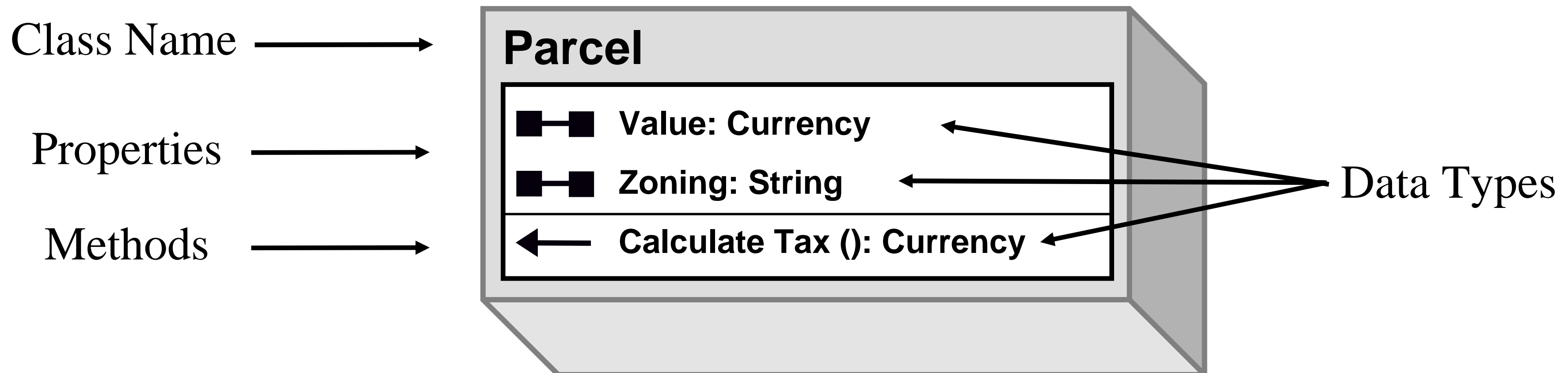
```
Public Function CalculateTax() As Currency

End Function
```

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

Creating classes – UML diagrams

- Just as it was useful to **diagram** our Form before we created it, diagrams are a useful way to **plan out the characteristics of a class** (and its relationships with other classes)
- We can make use of a **standard approach**, called the **Unified Modeling Language (UML)**, and create object model (or class) diagrams using its symbols, for example:



Creating objects

- **Creating (or instantiating) objects** is straightforward, once we have a **class defined** to describe them; there are a **couple of ways** we can do this:
  - We can (and have) created objects with the **user interface**, like our Form in Chapter 3
  - We can also create them **using code**, in the same fashion that we have been working with variables (declare them then set them)
- The **basic data types** we have worked with in the past (and used Dim to declare) are called **intrinsic variables**
- We can work with objects in ***nearly*** **the same way**; we refer to these as **object variables**, and still use the Dim keyword to declare them

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
```

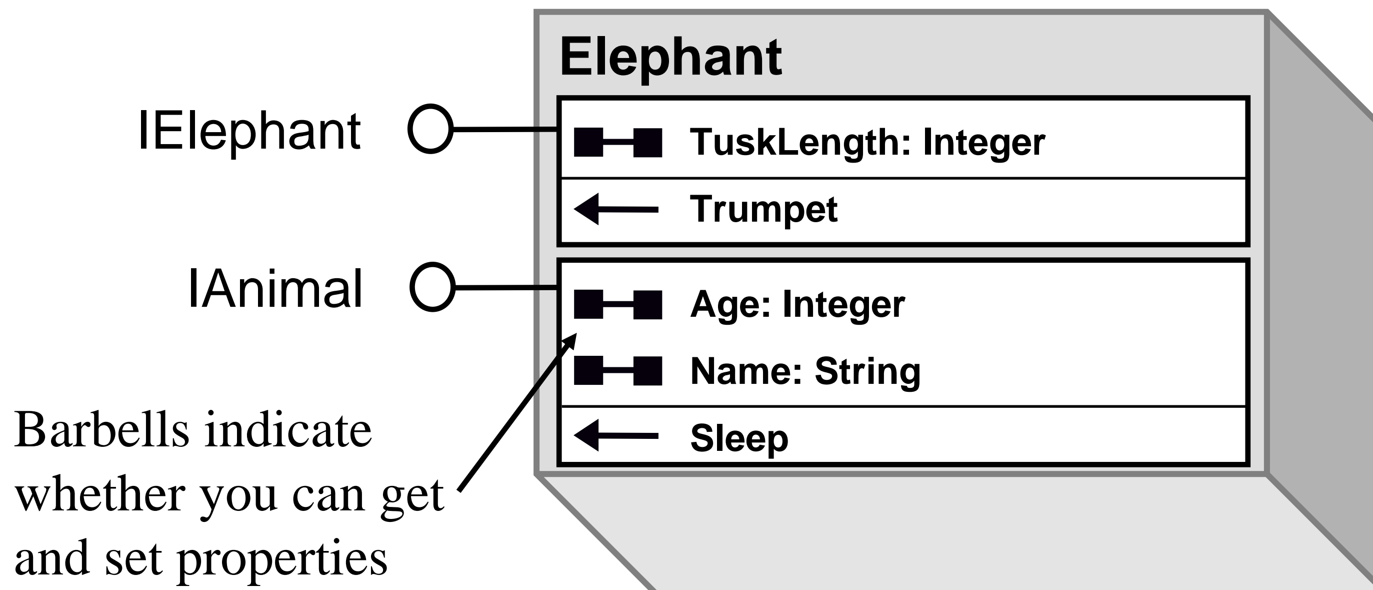
- The line used to set an object variable has to **begin with the Set keyword**, and must have the **New keyword after the equals sign** to denote the setting of a new object
- Getting/setting properties and using methods is the **same**

Chapter 10 – Programming with interfaces

- Using IApplication and IDocument
- Using multiple interfaces

Chapter 10 – Programming with interfaces

- For some classes, it is **useful to group** their properties and methods into **smaller subgroups**, based on their level of generality, or similarity, or their origin (more to come)
- **Interfaces** are logical groupings of properties and methods that are based on the criteria described above
  - E.g., the Elephant class from the text might have **two interfaces**:



Chapter 10 – Programming with interfaces

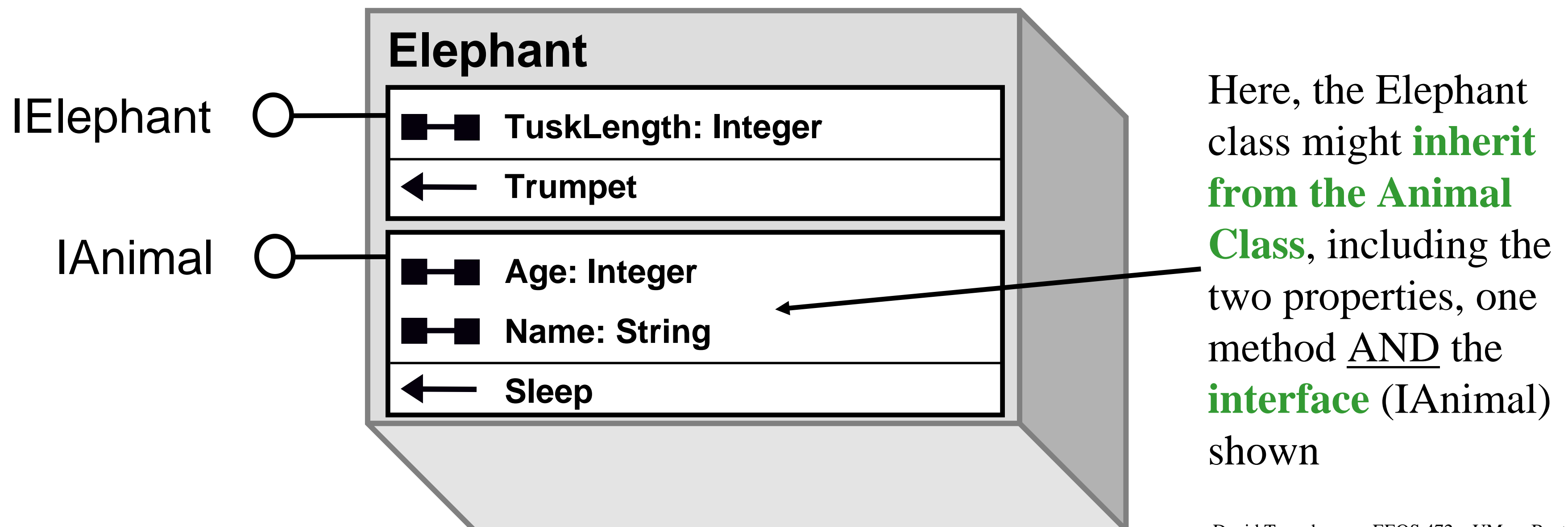
- To fully understand **why interfaces exist at all** (beyond the organizational reasons), you have to know a little more about the **software architecture** that underlies ArcGIS
- Interfaces are part of the **Component Object Model (COM)**, a set of programming standards developed by Microsoft that has many beneficial features:
  - Code written in one language **can work with code written in another language** (e.g. existing ArcObjects are C++, your new classes are VBA)
  - This allows the **reuse of classes**, between components and modules of one piece of software, or even between applications
  - This provides a **standard for creating classes and interacting and communicating with them**, through their interfaces

Chapter 10 – Programming with interfaces

- Once written, the code for an interface **never changes**
  - This way, once some **functionality is implemented, it persists**, and future programmers can always rely on it working in the same way, **even in newer versions** of the software
- But **more/multiple interfaces can be added** to add more functionality
  - e.g. perhaps we want to add something to the Elephant class lacking in IElephant, so we implement it in **IElephant2**
- It is important to understand that the **same interface can be available for use with multiple classes**, and to fully make sense of this, you need to be aware of two important concepts in OOP: **Inheritance** and **Polymorphism**

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

- A related concept is **polymorphism**, which in this context, is descriptive of the fact that **many classes can share the same interface**
- This should make more sense once we start examining **Chapter 11** which, amongst other things, will expand our understanding of the **relationships between classes**
- For our purposes now, we need to know that **multiple classes can have the same interface** AND that the classes you create **can use interfaces of existing ArcObjects classes**
  - Another way of putting this is **you can create variants of existing classes**, and take advantage of their existing interfaces

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]

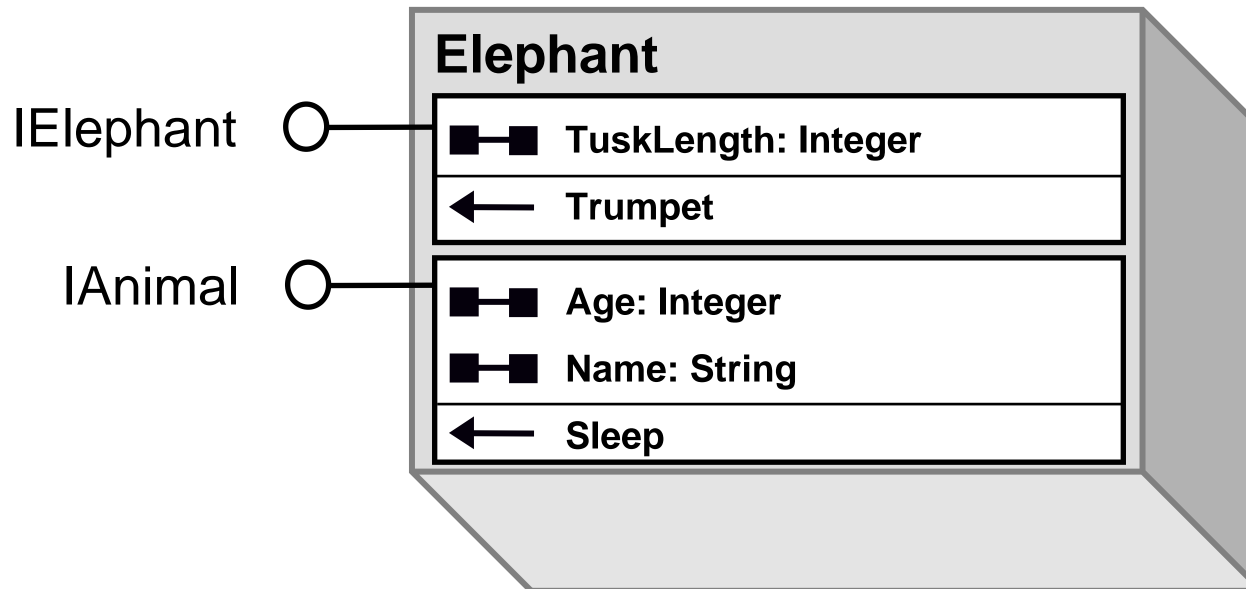
Using multiple interfaces

- 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"
```

Next Topic:

The object model, UML diagrams,
and making tools

(after the mid-term review & exam)