Chapter 3: Object Oriented Visual Basic programming

3-1: Custom Data Types

Sometimes you want to use a more complex data type. For example, in a 2-dimensional Cartesian plane, you want to store a pair of coordinates. That can be done by creating a custom data type using the `Type` statement.

```
[Private | Public] Type varname
  elementname [([subscripts])] As type
  [elementname [([subscripts])] As type]

End Type
```

The syntax should be comprehensible. Look at the following example:

```
Type Point
  x As Double
  y As Double
End Type
```

You can declare a variable with the custom type as you would do with other standard types. You can reference the sub-elements of your custom type with the period (.) operator. For example:

```
Dim p1 As Point
p1.x = 1
p1.y = 2
MsgBox "x = " & p1.x & "; y = " & p1.y
```

3-2: With statement

You can use With statement to simplify object and custom type variable references. For example, changing the above example,

```
Dim p1 As Point
With p1
  .x = 1
  .y = 2
  MsgBox "x = " & .x & "; y = " & .y
End With
```

This simplifies a lot of typing, especially later in the course when we want to reference several layers of objects.

3-3: Classes

Previously we developed a custom type called Point. We can treat a point as an object which we can manipulate with. In the same way, there are a lot of objects we can put into a program to help us develop computer applications. One other object we can think of is perhaps a line. A line is made up of two points, or a point and a slope. Perhaps we can represent a line as follows:

```
Type Line
  xy As Point
  slope As Double
End Type
```

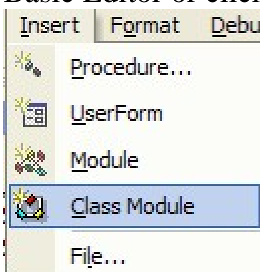
and we produce a line (or in computer terms, get an instance of a line) like this:

```
Dim aLine As Line
```

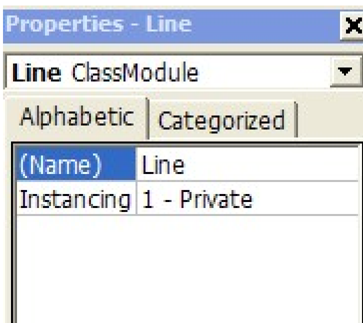
However, we want more than this. We want some operations to manipulate lines, for example, determining if a point lies in a line, or finding the point of intersection of two lines. We can define functions to do so. In a more organized way, we group the definitions of the data type and operations of a line together. These definitions are like templates or factories to generate some concrete objects. The group of definitions is called a class.

Let's try! Corner

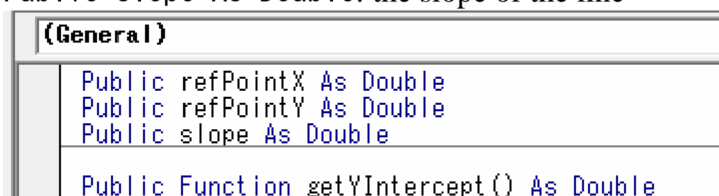
1 We define a class in a class module. You can add a class module in the upper left pane of the Visual Basic Editor or clicking Insert > Class Module from the menu bar.



2 We rename the class to Line in the lower left pane.



3 Add variables to the class: (they are called *properties* of the class)
`Public refPointX As Double`: the *x*-coordinate of a point to use with the Point-Slope form
`Public refPointY As Double`: the *y*-coordinate of a point to use with the Point-Slope form
`Public slope As Double`: the slope of the line



4 Add two functions to the class: (they are called *methods* of the class)
`Public Function getYIntercept() As Double`: get the *y*-intercept of the line
`Public Function getEquation() As String`: get the equation of the line (in Slope-Intercept form)

You will have a chance to test your new class in the next section.

3-4: Objects

Once you have designed a class (which is like a factory), you can create objects (which are like goods).

You can declare an object variable of a particular class with the following syntax:

```
Dim objName As className
```

However, the object variable has been declared only. The actual object has to be created before it can be used. You create an object with the `New` keyword. (This is also called "creating an *instance* of the class") After the object variable has been declared, you can use the object assignment statement to assign the newly created object to the declared variable. The object assignment statement has the following syntax:

```
Set objName = anotherObjName
```

The object assignment statement assigns the *reference* of the object at RHS to the object at LHS. Notice the `Set` keyword compared with the ordinary assignment statement. The following sets the newly created object to an object variable:

```
Set objName = New className
```

You can also create the object when you declare its variable:

```
Dim objName As New className
```

For example,

```
Dim myObj1 As SomeClass
Set myObj1 = New SomeClass
Dim myObj2 As New SomeClass
Dim myObj3 As SomeClass
Set myObj3 = myObj2
```

If you have declared an object variable, but have not created an instance of the class, the variable will have a value of `Nothing`.

You can access properties and methods of the object by using the period (.) operator:

```
objName.propertyName
objName.funcName arglist
```

For example,

```
myObj.name
myObj.doSomething var1, var2
returnVar = myObj.doSomething(var1, var2)
```

Let’s try! Corner

Test your new Line class by creating a new instance of the class. The object should represent a line passing through (1, 2) with slope 1. Call the two methods of the object and check the result. (y-intercept = 1, equation: $y = x + 1$)

If you have used the Set statement to bind two object variables together, say for the previous example,

```
Set myObj3 = myObj2
```

then the two variables are the same, in the sense that they are not two objects with the same properties, but one single object. It means, when you change the properties using one of the variables, you can access the changed properties with the other variable:

```
myObj3.name = "abc"
MsgBox myObj2.name 'Outputs "abc"
```

You can compare object variables with the `Is` operator. If the two variables refer to the same object, the operator will return true. For the previous case,

```
If myObj3 Is myObj2 Then
  MsgBox "myObj2 is myObj3" 'Will output
End If
```

If you want to check if the object variable has been assigned an instance of the class or not, you can use the following code:

```
If myObj Is Nothing Then
  'Has not been assigned an instance
Else
  'Has been assigned an instance
End If
```

You can also check the type of the object with the `TypeOf … Is …` Operator. For example,

```
If TypeOf myObj3 Is SomeClass Then
  MsgBox "myObj3 is of the type SomeClass" 'Will output
End If
```

To get the type name, use `TypeName(`*myObj*`)`.

3-5: Properties

Properties describe the characteristics of an object belonging to a class. You have seen the usage of public variables as properties. In fact VBA provides another mechanism to access the properties of a class. While accessing the properties, the programmer can impose some kind of error or access control in the meantime. This is done by using property procedures.

There are two types of property procedures: Property Get and Property Let/Set.

```
[Public | Private | Friend] [Static] Property Get propName [(arglist)] [As type]
  [statements]
  [propName = expression]
  [Exit Property]
  [statements]
  [propName = expression]
End Property
```

As you can see, the Property Get procedure is very much like a function. We will explain the difference between the scoop Public, Private and Friend in the next section.

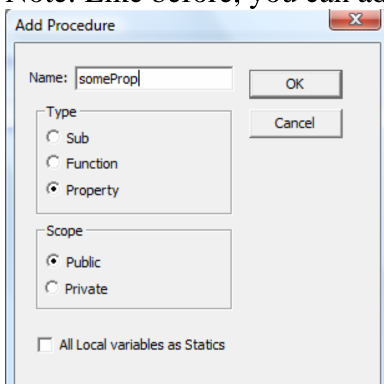
```
[Public | Private | Friend] [Static] Property Let propName ([arglist,] value)
  [statements]
  [Exit Property]
  [statements]
End Property

[Public | Private | Friend] [Static] Property Set propName ([arglist,] reference)
  [statements]
  [Exit Property]
  [statements]
End Property
```

The Property Let/Set is like a procedure, with a different in the argument list:

- `value` / `reference`: Required. Variable containing the value / object reference used on the right side of the assignment / object reference assignment statement.

Property Get is used while you are trying to read a property (when you put the property procedure at the right hand side of an assignment statement). Property Let/Set is used while you are overwriting a property (when you put it at the left hand side of an assignment statement). Property Let should be used when you are dealing with normal value variables, arrays and user-defined type variables. Property Set should be used when dealing with objects.

Note: Like before, you can add a pair of property procedures quickly by clicking Insert > Procedure…



You can invoke the Property Get procedure by the following statement:

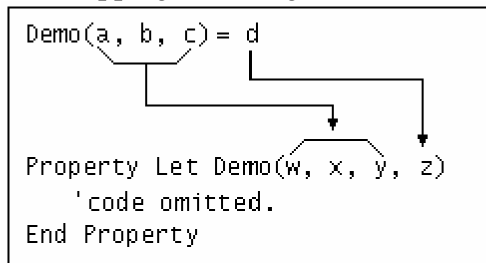
```
someValueVar = propName(arglist)
Set someObjectVar = propName(arglist)
```

depending on whether the return value is a value or an object.

And you can invoke the Property Let/Set procedure by:

```
propName(arglist) = someValueVar 'using Property Let
Set propName(arglist) = someObjectVar 'using Property Set
```

The mapping of the arguments when calling the Property procedures is as follows:



Let's try! Corner

1 We see that two properties, `refPointX` and `refPointY`, which we defined in the Line class is not good enough: they look better if we use the user-defined type Point to create the variable. We try to change the two properties into:

   ```
   Public mRefPoint As Point
   ```

   Change the codes in `getYIntercept()` and `getSlope()` if necessary to reflect the change.

2 Test your Line class by running the test procedure you have created last time (with minor correction of your code to reflect the change of properties). What happened?

3 Change the affected property into:

   ```
   Private mRefPoint As Point
   ```

   However you will not be able to directly modify the value of this property since you have declared it private.

4 Add a Property Let and a Property Get statement to facilitate the read/write of the property.

   ```
   Public Property Let refPoint(aPoint As Point)
     mRefPoint = aPoint
   End Property

   Public Property Get refPoint() As Point
     refPoint = mRefPoint
   End Property
   ```

5 You can test your class again by using the properties created.

3-6: Scoop of Properties and Methods (Advanced Topic)

We can declare a property variable, a property procedure or a method with Public, Private and Friend scoop.

- Public: accessible to all other procedures in all modules
- Private: accessible only to other procedures in the module where it is declared
- Friend: visible throughout the project, but not visible to a controller of an instance of an object

Normally you will use only Public or Private modifier. Public means it can be accessible by the object with the period (.) operator, while Private means it can only be accessible by methods in the class.

3-7: Constructors (Advanced Topic)

Constructors are methods which will be called automatically when an object is being created. You can insert a constructor into the class by inserting a Sub with the name Class_Initialize and with no parameters. For instance,

```
Private Sub Class_Initialize()
  MsgBox "Initializing"
End Sub
```

Usually initializing codes are stored in constructors.

3-8: Destructors (Advanced Topic)

Destructors contrasts with constructors in the way that they are called when an object is being destroyed. For example, objects are destroyed when the program ends, or they are no longer referenced by any object variable. Like constructors, you can insert a Sub with the name Class_Terminate and with no parameters to indicate that it is a destructor.

```
Private Sub Class_Terminate()
  MsgBox "Terminating"
End Sub
```

Let's try! Corner

Try inserting the above constructor and destructor into the Line class to see when objects are created and destroyed.

3-9: Inheritance with Interfaces (Advanced Topic)

The object-oriented programming is so powerful since it can provide *inheritance* to classes. Inheritance creates subclasses, which are more specialized versions of a class. They inherit properties and methods from their parent classes.

In VBA, we cannot achieve real inheritance. We can only pretend inheritance using *interfaces*. An Interface is a document to specify which kind of properties and methods a class needs to implement. It is like a blueprint for class construction. Many classes can follow the interface to implement, and they can have different implementation methods.

Follow the steps below to learn creating an interface and some inheriting classes.

Let's try! Corner

1 Create a Curve class. In the curve class, declare two empty functions (you don't need to write anything in the functions: they are just blueprints, so no implementations needed):
```
Public Function getYIntercept() As Double
Public Function getEquation() As String
```
```
(General)
Public Function getYIntercept() As Double

End Function

Public Function getEquation() As String

End Function
```
2 Change your Line class. At the beginning of your class, add a statement:
```
Implements Curve
```
3 At the top of the code editor, you will see two select boxes. Select Curve at the left box, and select the two functions, one at a time. You will notice two new functions appear somewhere in your code. They are called "Curve_getEquation" and "Curve_getYIntercept". They represent the functions in the interface which you need to implement.
```
Curve                                    getEquation
                                         getEquation
                                         getYIntercept
Implements Curve

Private mRefPoint As Point
Public slope As Double

Public Property Let refPoint(aPoint As Point)
  mRefPoint = aPoint
End Property
```
4 Implement the classes. In fact, you have done that earlier. So the method to implement is just call your old functions in these new ones, like this:
```
Private Function Curve_getEquation() As String
  Curve_getEquation = getEquation
End Function

Private Function Curve_getYIntercept() As Double
  Curve_getYIntercept = getYIntercept
End Function
```
5 We take the chance to add a function:
```
Public Sub initialize(aPoint As Point, aSlope As Double)
```
so that we can initialize the line with one step. Write your code.

6 Construct another class QuadraticCurve which implements Curve. QuadraticCurve takes in three numbers *a*, *b*, *c*, representing $y = ax^2 + bx + c$. Construct functions `getYIntercept()` and `getEquation()` like before.

7 Repeat step 3-4 to insert the functions required by the interface.

8 For the Quadratic Curve, add a function:
```
Public Function getExtremePoint() As Point
```
so that it can calculate the maximum or minimum point of the curve. (In case you have forgotten your Cert level mathematics, the formula for the point (x_0, y_0) is $x_0 = -\frac{b}{2a}$, and $y_0 = ax_0^2 + bx_0 + c$.)

9 Also, add a function:
```
Public Sub initialize(aA As Double, aB As Double, aC As Double)
```
to initialize the quadratic curve in one step.

You will be able to test your classes in the next section after some theories.

3-10: Polymorphism (Advanced Topic)

Polymorphism means that many classes can provide the same property or method, and a caller does not have to know what class an object belongs to before calling the property or method. In essence, each class provides a different type-specific code within a method. For example, a Rectangle class and a Circle class might each have a calculateArea method, where the method of calculating area is different. Polymorphism means that you can invoke calculateArea without knowing whether an object is a Rectangle or a Circle.

Let's try! Corner

You will be entering 4 lines or quadratic curves to your program:
1st: Line; point: (1, 2); slope: 1
2nd: Line; point: (2, 3); slope: 4
3rd: Quadratic; $a = 1$, $b = 2$, $c = 1$
4th: Quadratic; $a = 2$, $b = 5$, $c = 2$

You will output the equations and *y*-intercepts of the curves after data entry. If the curve is a quadratic one, also output its extreme point.

1 In a normal module, create a new Sub. Declare a Line variable, a QuadraticCurve variable, a Point variable and a Curve array variable with 4 cells.

```
Sub ch3_10_ex1()
  Dim aLine As Line, aQuadraticCurve As QuadraticCurve
  Dim aPoint As Point
  Dim aCurve(3) As Curve
```

2 Use the Line variable, create a new line object and input the first line into the Curve array.

```
  'data entry
  Set aLine = New Line
  aPoint.x = 1
  aPoint.y = 2
  aLine.initialize aPoint, 1
  Set aCurve(0) = aLine
```

3 Do this again for the second line.
4 Repeat step 2-3, but input the 3rd and 4th curve with the QuadraticCurve variable.
5 Start a For loop to output the equations and *y*-intercepts of the four curves. Notice that you are using an object variable declared as a Curve to do the output. VBA will notice the actual type of the object (a Line or a QuadraticCurve) and delegate the function call to the respective class.

```
  Dim i As Integer, str As String
  For i = 0 To 3
    str = "Equation of Curve " & i + 1 & ": " & aCurve(i).getEquation 'notice Polymorphism
    str = str & vbCrLf & "Y-intercept: " & aCurve(i).getYIntercept
```

6 Check if the curve is a QuadraticCurve. If so, output the extreme point as well. You can use a TypeOf … Is … to check the type.

```
If TypeOf aCurve(i) Is QuadraticCurve Then 'notice the use of TypeOf...Is to check type
  Set aQuadraticCurve = aCurve(i)
  str = str & vbCrLf & "Extreme point: (" & aQuadraticCurve.getExtremePoint.x & _
    ", " & aQuadraticCurve.getExtremePoint.y & ")"
End If
```

7 Remember to use a MsgBox to output your string. Test your Sub.

Let's try! Corner

I have created a Stock series of classes to simulate stock price with a binomial model. Stock class is an interface for ContDivStock and DiscDivStock. ContDivStock class represents the group of stocks who pay continuous dividends, and DiscDivStock class represents those who pay discrete dividends.

1 Download Stock.cls, ContDivStock.cls, DiscDivStock.cls from the course website. Import them into your Visual Basic Editor. If you have time, study the structure of the classes.
2 Create one continuous dividend stock and one discrete dividend stock. Enter some sample data in advance (using initialize method).
3 Ask the user to choose one kind of stock to simulate. You can use an input box to ask the use to input his choice.
4 Using polymorphism, advance 100 days using advance method of the Stock interface.
5 Using TypeOf … Is … operator, check the type of the object chosen. Pay dividend once (using payDividend method).
6 Show the percentage gain with a message box (using percentGain method).

3-11: Collections

We can use collections conveniently to replace one-dimensional arrays. Collection is a class already defined by VBA. A collection can be created the same way other objects are created. For example:

```
Dim X As New Collection
```

Once a collection is created, members can be added using the Add method and removed using the Remove method. Specific members can be returned from the collection using the Item method, while the entire collection can be iterated using the For Each...Next statement.

Properties:

- `Count`: Returns a Long (long integer) containing the number of objects in a collection. Read-only.

Methods:

- **Add Method**
  Adds a member to a Collection object.
  Syntax: *object*.Add *item*, *key*, *before*, *after*

Part	Description
item	Required. An expression of any type that specifies the member to add to the collection.
key	Optional. A unique string expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection.
before	Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the before argument. If a numeric expression, before must be a number from 1 to the value of the collection's Count property. If a string expression, before must correspond to the key specified when the member being referred to was added to the collection. You can specify a before position or an after position, but not both.
after	Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the after argument. If numeric, after must be a number from 1 to the value of the collection's Count property. If a string, after must correspond to the key specified when the member referred to was added to the collection. You can specify a before position or an after position, but not both.

  Remarks: Whether the before or after argument is a string expression or numeric expression, it must refer to an existing member of the collection, or an error occurs.
  An error also occurs if a specified key duplicates the key for an existing member of the collection.

- **Item Method**
  Returns a specific member of a Collection object either by position or by key.
  Syntax: *object*.Item(*index*)

Part	Description
index	Required. An expression that specifies the position of a member of the collection. If a numeric expression, index must be a number from 1 to the value of the collection's Count property. If a string expression, index must correspond to the key argument specified when the member referred to was added to the collection.

  Remarks: If the value provided as *index* doesn't match any existing member of the collection, an error occurs.
  The Item method is the default method for a collection. Therefore, the following lines of code are equivalent:

```
MsgBox MyCollection(1)
MsgBox MyCollection.Item(1)
```

- **Remove Method**

  Removes a member from a Collection object.

  Syntax: *object*.Remove *index*

Part	Description
index	Required. An expression that specifies the position of a member of the collection. If a numeric expression, index must be a number from 1 to the value of the collection's Count property. If a string expression, index must correspond to the key argument specified when the member referred to was added to the collection.

  Remarks: If the value provided as *index* doesn't match an existing member of the collection, an error occurs.

Let's try! Corner

Convert your Curve array variable in the last exercise with a Collection object. Notice that the starting index of a Collection object is 1, so make suitable change to the index in the For loop.

Exercise 3

1. Design a class called RegressionCalculator, which uses the methodology explained in 2-15-5 to calculate the regression equation. Add one more method to the class to calculate the error of the regression: $\|\hat{\mathbf{r}}\| = \|\mathbf{y} - \mathbf{A}\hat{\boldsymbol{\beta}}\| = \left\|\mathbf{y} - \mathbf{A}\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{y}\right\|$, where $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T\mathbf{x}}$. Of course you should test your class to see if it works. (Answer of error: 0.81650)
2. Design an interface called Shape. A shape object should be able to calculate area and perimeter, and get the type of the shape (rectangle, circle, etc.). Design classes which implement Shape: Triangle, Rectangle and Circle. Test polymorphism by creating different types of Shape objects and ask the user to choose which object and method to invoke (by means of input box, message box, or any other methods you like).
3. You need to take care of students' marks in a class. Design a class called StudentRecord to hold the student's name, student ID, marks of exam. Design another class called Course so that it can hold the course code, course title and an array (or a collection) of StudentRecord objects. After data entry, the Course class can calculate the students' average exam score and get the student who has the highest and lowest marks.
4. Try to make a portfolio of stocks with the Stock series by constructing a class called StockPortfolio and calculate the overall return of the portfolio.