Introduction to Class Programming – Introduction

In application program development, one of the most important considerations is the program's ability to accurately represent our perception of the real world. This requires the program to incorporate a realistic data model. Such a program is certainly easier to understand than one that performs its task without using a data model.

In order to develop a data model, we begin with the contention that the real world consists of *entities* and *relationship*.

An entity is defined as a live being, object or abstraction that can be described in terms of certain characteristic, and is similar to, but distinct from, entities of the same type.

An *entity set* is a collection of entities of the same type. For example, the books in your personal library make up an entity set; this book that you are reading now is a member of the entity set. You are a student, and therefore, you are member of the entity students of your college.

Each member of an entity set is different from the other member of the same set. The difference stem from their individual characteristic, called attributes. For example, a book is different from all other book because of its subject matter, author, publisher, design characteristic and unique ISBN book number. As a member of entity set of students, you are different from the rest of the student in your college because of your characteristic, such as your social security number, name, and so on.

In addition to entity set, the real world teems with all kinds of relationship. A relationship is an association among member of one or more entity set. A relationship set is a set of similar but distinct relationships of the same type. For example, class enrollment can be regarded as a relationship set between the entities sets of students and courses. The fact that you own a book can be seen as a relationship between you, a member of the entity set of students, and your book, a member of the entity set of textbooks.

One way you can represent entity set in Visual Basic is by using Classes and Objects. Fore example, the structure of Students below can be use to represent the student entity set or any its member in terms of their attributes, such as `Student_ID`, `FirstName`, `LastName`, `MajorCode`, `YearLevel` **and** `BirthDate`.

In Visual Basic, you can represents relationship set by providing the former entity object with an object property to the latter. For example a typical `Enrollment` **entity might expose a** `Student` **entity. The relationship set enrollment can be modeled using this structure:**`Student_ID, CourseNumber, AcademicYear, Semester and FinalLetterGrade`

Many novice programmers when exposed for the first time in Object Oriented Programming tend to confuse classes and object. To aid the programmer about classes and object, this will give you some reminders. Your users will never see a class; rather, they will only see and act with objects created from your classes. As a programmer, your point of view is opposite because the thing you'll have in front of you while you're writing the application is the class, in the form of a class module. Until you run the application, an object isn't more real than a variable declared with a *Dim* statement in a code listing.

A class is a part of the program that defines the properties, methods and events of one or more object that will be created during execution. An object is an *entity* created at run time, which requires memory and some resources and is then destroyed when it's no longer needed or when the application end. In short, a class are design time only entities, while object are run-time entities. Or in more or less technical term

A class is a template or formal definitions that defines the *properties* of an object and the *methods* used to control that object's behavior. The description of these members is done only once, in the definition of the class. The object that belongs to a class, called instance of the class, share the code of the class to which they belong but contain only their particular setting for the properties of the class. *Everything an object knows is expressed in its properties and everything it can do is expressed in its methods*. Object interacts with each other by sending message requesting that

method be carried out, or that properties be set or returned. A message is simply the name of an object followed by the name of one of its members.

Classes can improve your productivity. Classes can organize your code into truly reusable modules and design your applications entirely using concepts derived from the Object-Oriented Design discipline. And the most important reason, objects are the base on which almost every feature of Visual Basic, without objects you can't do serious programming, you can't deliver Web based applications, and you can't write components for COM, DCOM, or MTS. To cut a long story short, you can do little or take advantage of them. If you are new to object-oriented programming, don't worry! The series of articles will help you to understand how objects can help you write better programs and plunge into object-oriented programming. But I assuming , you've already learned to master many advanced programming techniques concerned with, such as, events, database programming, and user interfaces.

Before diving into object-oriented programming let try to understand some concepts that will help you later in the articles.

Encapsulation

***Encapsulation* is the process of combining logically related procedure and data in one class/objects. This way, each object is insulated (separated, protected) from the rest of the program. Because the object is only using data contained within it or passed to it. And it executes only internal procedures. It does not contain any global or public variables, and does not require any external procedures to execute its members. The data and behaviors of an encapsulated object can only be accessed and manipulated through its properties and public methods. Thus encapsulation provides several advantages for you as a programmer. You can protect data from corruption by other objects or parts of the program. You can hide low-level, complex implementation details from the rest of the program, which results in the ability to implement a simple public interface to a more complex set of private member. It is also easier to maintain legacy code or add new members to the object without affecting any procedures that currently call the object. You can to debug individual object and ensure that a bug in one object will not affect some other part of the system in an apparently unconnected way. And lastly, you can reuse the object or by other programmer, improving the productivity.**

Polymorphism

***Polymorphism* is the ability of different classes to expose similar (or identical) interfaces to the outside. The most unmistakable kind of polymorphism in Visual Basic is forms and controls. For example, *TextBox* and *PictureBox* controls are completely different objects, but they have some properties and methods in common, such as *Left* property and *Move* method. As a programmer, you don't need to worry about how they implement its functionality, instead, all you have to do is to apply it to a supported object and pass a correct value as arguments.**

Inheritance

***Inheritance* is the ability to derive a new class (the derived or inherited class) from another class (the base class). The derived class automatically inherits the properties and methods of the base class. For example, you could define a generic Person class with properties such as `FirstName` and `Lastname` and then use it as a base for more specific classes (for example, Student, Faculty, and so on) that inherit all those generic properties. You could then add specific members, such as `BirthDate` and `StudentID` for the `Student` class and `FacultyID` for the `Faculty` class. Thus it reduce the amount of code on your class itself, therefore simplifies the job of the class author. Unfortunately, Visual Basic doesn't support inheritance, at least not in its more mature form of implementation inheritance.**

Creating a class in Visual Basic is very simple: just select the Add Class Module command from the Project menu. A new code editor window appears on an empty listing. Visual Basic automatically add a class module named Class1, so the very first thing you should do is change the Class name in the Project Properties window in a more appropriate name. The first version of our class includes only a few properties. These properties are exposed as Public members of the class module itself.

- **To create a class module**

  1. **Start Visual Basic.**
  2. **In the New Project dialog box, select Standard EXE, then click OK.**
  3. **On the Project menu, click Add Class Module.**
  4. **In the Add Class Module dialog box, select Class Module, then click Open.**
  5. **In the Properties window, set the Name property for the class module to Student.**

- **To create public property**

  1. **In the Code window, type the following:**

```
'In the declaration section of the Student class
module
Public        Student_ID        As        String
Public         FirstName        As        String
Public          LastName        As        String
Public         MajorCode        As        String
Public         YearLevel        As        String
Public BirthDate As Date
```

This is a very simple class, which consists of Public properties, so we are not distracted by OOP details (not yet), we will just examine the concept. Once you've created a class, you can create an instance of that class then you can use the properties of that class. The following example creates an instance of the Student class, and sets and retrieves its properties:

- **To Use the Student Class**

  1. **Place a command button on Form1.**
  2. **In the Click event for the command button, type the following:**

```
'Declare an object Student
Dim objStudent As Student

'Create an instance of the class
Set objStudent = New Student

'Use the object Student
objStudent.StudentID = "12345"
objStudent.FirstName = "Cathrina"
objStudent.LastName = "Aniversario"
objStudent.MajorCode = "C"
objStudent.YearLevel = "Freshmen"
objStudent.BirthDate = "Oct 10, 1980"

MsgBox "Student ID   : " & objStudent.StudentID & vbCrLf & _
       "Student Name : " & objStudent.FirstName & " " & _
                           objStudent.LastName & vbCrLf & _
       "Major Code   : " & objStudent.MajorCode & vbCrLf & _
       "Year         : " & objStudent.YearLevel & vbCrLf & _
       "BirthDate    : " & objStudent.BirthDate

Set objStudent = Nothing
```

3. On the **Run** menu in Visual Basic, click **Start**.

4. When the program is running, click the **Command1** button.

5. Click OK to close the message box.

6. On the **Run** menu in Visual Basic, click **Stop**.

Object Keyword

The *New* keyword

The *New* keyword (when used in a *Set* command) tells Visual Basic to create a brand-new instance of a given class. The keyword then returns the address of the instance data area just allocated.

The *Set* command

The *Set* command simply copies what it finds to the right of the equal sign into the object variable that appears to the left of it. This value can be, for example, the result of a *New* keyword, the contents of another variable that already exists, or the result of an expression that evaluates to an object. The only other tasks that the *Set* command performs are incrementing the reference counter of the corresponding instance data area and decrementing the reference counter of the object originally pointed to by the left-hand variable (if the variable didn't contain the Nothing value):

Unlike regular variable, which can be used as soon as they have been declared, an object variable must be explicitly assigned an object reference before you can invoke the object's properties and methods. When an object variable has not been assigned, it contains the special **Nothing** value, meaning it doesn't contain any valid reference to an actual object. For example try this code:

```
' Declare the variable
  Dim objStudent As Student

' Then comment out the next line
' Set objStudent = New Student

' Raises an error 91
' "Object variable or With block variable not set"
   MsgBox objStudent.FirstName
```

The code will give you an error, because we trying to use an object that doesn't exist. This behavior is favorable because it doesn't make much sense to use a property of an object that doesn't exist. One way to avoid the error is to test its contents of an object variable using the *Is Nothing*

```
' Use the variable only if it contains a valid object reference
If Not (objStudent Is Nothing) Then MsgBox objStudent.FirstName

MsgBox objStudent.FirstName
```

But in other cases, you may want to create an object and then assign its properties. You might find it useful to declare an auto-instancing object variable using the *As New* clause:

```
Dim objStudent As New Student ' Auto-instancing variable
```

At runtime, when Visual Basic encounters a reference to an auto-instancing variable, it first determines whether it's pointing to an existing object and creates a brand new instance of the class if necessary. But auto-instancing variables have an advantage and disadvantage:

- It reduce the amount of code you need to write to be up and running with your classes. This can be useful if you are prototyping an application.

- In some condition, you might declare a variable but never actually use it: which happens all the time with standard variables and with object variables too. In truth is, if you create an object with a Set command at the beginning of a procedure, you might be creating an object for no real purpose (thus taking both time and memory). On the other hand, if you delay the creation of an object until you actually need it, you could soon find yourself drowning in a sea of *Set* commands, each preceded by an *Is Nothing* test to avoid re-creating an object instanced previously. Auto-instancing variables are automatically created by Visual Basic only if and when they are referenced. This is probably the situation in which auto-instancing variables are most useful.

- Your object variable cannot be tested against the *Nothing* value. In fact, as soon as you use one in the *Is Nothing* test, Visual Basic insistently creates a new instance and the test always returns *False*

- It eliminate errors, which is sometimes this is specifically what you don't need especially during the development stage, because during this state, you want to see all the errors because this might be the symptoms of other serious deficiency in your code logic.

- Auto-instancing variables make the debugging step a little more difficult to understand because you can never be sure when and why an object was created.

- You can't declare an auto-instancing variable of a generic type, such as *Object*, or *Form* because Visual Basic must know in advance which kind of object should be created when it references that variable for the first time.

- Finally, each time Visual Basic references an auto-instancing variable, it incurs a small performance hit each time Visual Basic reference an auto-instancing, because Visual Basic has to check whether it's Nothing.

In short, auto-instancing variables are not the best choice for creating the object and you should stay away from it.

Object Keyword

The *Nothing* value

The *Nothing* keyword is the Visual Basic way of saying *Null* or *0* to an object variable.

The *Is* operator

The *Is* operator is used by Visual Basic to check whether two object variables are pointing to the same instance data block. At a lower level, Visual Basic does nothing but compare the actual addresses contained in the two operands and return True if they match. The only possible variant is when you use the *Is Nothing* test, in which case Visual Basic compares the contents of a variable with the value 0. You need this special operator because the standard equal symbol, which has a completely different meaning, would fire the evaluation of the objects' default properties:

Properties Of a Class

Now is the time to make our class to more robust class. A robust class is one that actively protects its internal data from tampering. So how can a class protect itself from invalid assignments, such as an empty string for its *FirstName* or *LastName* properties. To accomplish this purpose, you must change the internal implementation of the class module, because in its present form you have no means of trapping the assignment operation. Simply change all the Public member into Private members and encapsulate them in pairs of Property procedures.

- To change our Student class

    1. Double click the class Student.cls in the Project Explorer

    2. In the Student Class Module, change all word Public to Private and add a prefix m_ in front of all private variables, as shown below:

```
'In the declaration section of the Student class module
```

```
Private m_Student_ID As String
Private m_FirstName As String
Private m_LastName As String
Private m_YearLevel As String
Private m_BirthDate As Date
```

NOTE

You can also use Replace Dialog box. To do this, press Ctrl-H, the Replace Dialog box appears. On the Find What combo box, type *Public*. Next on the Replace With combo box, type *Private*, then click Replace All button.

Appending the prefix *m_* is just a personal style, this way it keeps my property name and private member variable synchronize and it is commonly used in programming. Feel free to use it or to create your own style.

3. **In the Student Class Module, type the following code:**

```
'In the declaration section of the Student class module
Private m_Student_ID As String
Private m_FirstName As String
Private m_LastName As String
Private m_MajorCode As String
Private m_YearLevel As String
Private m_BirthDate As Date

Property Get MajorCode() As String
    MajorCode = m_MajorCode
End Property

Property Let MajorCode(ByVal strNewValue As String)
    ' Raise an error if an invalid assignment is attempted.
    If Len(strNewValue) = 0 Or Len(strNewValue) > 1 Then Err.Raise 5
    m_MajorCode = strNewValue
End Property

Property Get FirstName() As String
   FirstName = m_FirstName
End Property

Property Let FirstName(ByVal strNewValue As String)
  ' Raise an error if an invalid assignment is attempted.
  If Len(strNewValue) = 0 Then Err.Raise 5 ' Invalid procedure argument
  ' Else store in the Private member variable.
  m_FirstName = strNewValue
End Property

Property Get LastName() As String
   LastName = m_LastName
End Property

Property Let LastName(ByVal strNewValue As String)
   ' Raise an error if an invalid assignment is attempted.
   If Len(strNewValue) = 0 Then Err.Raise 5        ' Invalid procedure
argument
   ' Else store in the Private member variable.
   m_LastName = strNewValue
End Property

Property Get StudentID() As String
   StudentID = m_Student_ID
End Property

Property Let StudentID(ByVal strNewValue As String)
   ' Raise an error if an invalid assignment is attempted.
   If Len(strNewValue) = 0 Then Err.Raise 5       ' Invalid procedure
argument
   ' Else store in the Private member variable.
   m_Student_ID = strNewValue
End Property

Property Get BirthDate() As Date
   BirthDate = m_BirthDate
End Property

Property Let BirthDate(ByVal datNewValue As Date)
   If datNewValue >= Now Then Err.Raise 1001, , "Future Date!"
     m_BirthDate = datNewValue
End Property

Property Get YearLevel() As String
   YearLevel = m_YearLevel
End Property

Property Let YearLevel(ByVal strNewValue As String)
  Dim varTemp As Variant
  Dim found As Boolean

  For Each varTemp In Array("Freshmen", "Sophomore", "Junior",
"Senior")
    If InStr(1, strNewValue, varTemp, vbTextCompare) Then
      found = True
      Exit For
    End If
  Next

  If Not found Then Err.Raise 5
  m_YearLevel = strNewValue
End Property
```

NOTE

Visual Basic can help you in typing Property Procedure by Add Procedure command from the Tools menu, which creates a templates for *Property Get* and *Let* procedures. But you should edit the result to a proper data type, because all properties created by this command is of type Variant.

4. **On the Run menu in Visual Basic, click Start.**

5. **When the program is running, click the Command1 button.**

6. **Click OK to close the message box.**

7. **On the Run menu in Visual Basic, click Stop.**

Everything works as before. What we have done, is make the class a bit more robust because it now refuses to assign invalid values to its properties. To see what I mean, just try to issue this command:

```
objStudent.FirstName = ""      'Raises an error 'Invalid Procedure call
```

Every time you assign a new value to a property, Visual Basic checks whether there's an associated *Property Let* procedure and passes the new value to its associated Private variable. If the code can't validate this new value, it raises an error and throws the execution back to the caller. Otherwise, the execution proceeds by assigning the value to the Private variable. And when the caller code requests the value of the property, Visual Basic executes the corresponding *Property Get* procedure, which simply returns the value of the Private variable. Try to trace your code by pressing F8, and see what actually those property procedure do.

NOTE

The type expected by the *Property Let* procedure must match the type of the value returned by the *Property Get* procedure.

You might be asking this question, "Why not just use a public variable to store property values? In a sense, we are just using a sort of indirection. Why not use function to return a value or sub to assign a value" In some cases that may work fine; however, if you want to make sure that the value assigned to the property is valid, you have to write validation code when the value is assigned. And usually you this by creating a Sub or Function to validate the public variable or worst you end up coding a lot in the client (form) for this validation routine. It end up, we did not benefit for the characteristic of OOP which is encapsulation, stated that "an object is the sole owner of its own data." And this is the benefit of using a property procedure.

If you look at Visual Basic how it handle its own object, such as form and controls, some properties can be both read and be written to. For example, you cannot modify the *Height* property of a ComboBox even at design time and you cannot modify the *MultiSelect* property of the ListBox at run time. You can also use this technique to limit the access to your class properties, thus making them read-only.

You can make a property to be read-only property by simply omitting its *Property Let* procedure. For example, we might add a *FullName* property to our Student class.

```
Public Property Get FullName() As String
  ' Raise an error if an FirstName or LastName is empty
   If (Len(m_FirstName) = 0) Or (Len(m_LastName) = 0) Then Err.Raise 5
  ' Else return the Student Fullname
   FullName = m_FirstName & " " & m_LastName
End Property
```

Now test your read-only property. Try to issue a command like as shown below:

```
'this raise an error Compile Error:  Cannot assign to read-only property
objStudent.FullName = "Samantha Aniversario"
```

Visual Basic raises a Compile Error "Cannot assign to read-only property", because you are trying to assign a value to a read only property. Your program won't even compile or run until you

delete this line of error. In addition, if we omit either the *FirstName* and *LastName* assignment statement (to be precise, omit the call of either *FirstName* or *LastName* Propert Get), Student class will raise an error when we try to execute the read-only property *FullName*. The trick is every time we use *FullName* property, the code will check the value stored in our Private member *m_FirstName* and *m_LastName*. If either of the two property does not contain any value, there is no reason to return the value of *FullName*. In fact, have you ever met a person with only have a FirstName or LastName?

This kind of property is nearly the most perplex property that Visual Basic has to offer. Because you cannot find anything useful for this property. And usually, programmer find it as unnatural of using this kind of property. For completeness, write-only property can be implemented by omitting the corresponding *Property Get*. A classical example is a *Password* property in a secured application usually with Login features. You can assign and validate the login process of the user, but your user cannot read it, because it compromise the security of the your application.

```
Private m_Password As String

Property Let Password(ByVal strNewValue As String)
    ' Raise error if password in invalid
    ' .. code omitted..
    ' Else assign to member variable.
    m_Password = strNewValue
End Property
```

Frequently, programmer prefer to create a method that accept the value as an argument, if the circumstances occurs like for our example.

In addition to class member (*property*) data. Your custom class can also include Sub and Function procedures, which are commonly known as *methods* of the class. A method of a class represent some standard operations on the class itself (properties). As you know, the difference between Sub and Function, is that Sub does not return any value, whereas Function method returns a value. But Visual Basic lets you invoke a function and discard its return value. In our example class, you could easily add a routine that calculate the Student *Age*.

```
Function Age() As Integer
' Returns the age in years between 2 dates.
' Doesn't handle negative date ranges i.e. BirthDate > Now()
    If Month(Now()) < Month(BirthDate) Or _
      (Month(Now()) = Month(BirthDate) And _
      Day(Now()) < Day(BirthDate)) Then
        Age = Year(Now()) - Year(BirthDate) - 1
    Else
        Age = Year(Now()) - Year(BirthDate)
    End If
End Function
```

As you can see in our example, if you are within the class module, you don't need the dot syntax to refer to the properties of the current instance. In addition, if you refer to a Public name for a property (*BirthDate*) instead of the corresponding Private member variable (*m_BirthDate*), Visual Basic executes the *Property Get* procedure as if the property were referenced from outside the class.

```
'In your client form
MsgBox "Student Age : " & objStudent.Age
```

Now let create another function, that checks the validity of *YearLevel*. We will make this function to be Private meaning that this procedure can only be called from within the module.

```
'In your Student class module
' Private method of a class, cannot be used outside
```

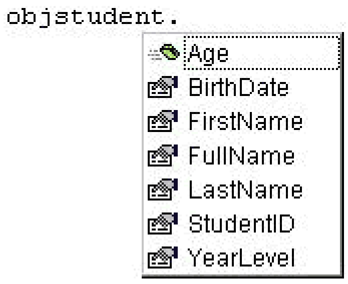
```
Private Function IsValidYearLevel(level As String) As Boolean
  Dim varTemp As Variant
  Dim found As Boolean

  For  Each  varTemp  In  Array("Freshmen",  "Sophomore",  "Junior",
"Senior")
    If InStr(1, level, varTemp, vbTextCompare) Then
        found = True
        Exit For
    End If
  Next

  IsValidYearLevel = found
End Function

'In Property Let YearLevel
Property Let YearLevel(ByVal strNewValue As String)
   If Not IsValidYearLevel(strNewValue) Then Err.Raise 5
   m_YearLevel = strNewValue
End Property
```

In other words, you cannot call this method in your client application. In fact, you cannot see a Private function in *IntelSense* technology of Visual Basic as shown below.



As you may pointed our earlier. Properties is like a function. So how can we know when we must implement property or a function. To tell you honestly, their is no universal rule concerning this scenario, but usually programmer implement a properties when a routine serves mostly to return a value stored inside the class and can be quickly and easily reevaluated. When the routine servers mostly to evaluate a complex value, they use *function*. If programmer thinks that in the future the value returned by the routine could be assigned to, they use Property Get procedure and gives them a chance to add a Property Let when its time to implement one.

Let's make an example. Earlier we implement a Property Get procedure for our class member *FullName*. How can we make our class more useable in a long run by providing a Property Let. This way we can have two way of assign a value to *FirstName* and *LastName* property. A possible solution might look like this:

```
Property Let FullName(ByVal strNewValue As String)
' Return the full name of Student object
    Dim aStrName() As String
    ' Split the argument pass (strNewValue)
    aStrName() = Split(strNewValue)
    ' Raise an error if an FirstName or LastName is empty
    If UBound(aStrName) = 0 Then Err.Raise 5
    FirstName = aStrName(0)
    LastName = aStrName(1)
End Property
```

You can directly assign a value to *FirstName* and *LastName* property as shown below:

```
objStudent.FullName = "Cathrina Anniversario"
'try getting the Student property FirstName and LastName
MsgBox objStudent.FirstName  'Invoke Property Get FirstName
MsgBox objStudent.LastName   'Invoke Property Get LastName
```

As you can see, even if we didn't assign a value to the Student *FirstName* and *LastName* property explicitly, our new *FullName* property does the job. This is other nice thing you can do with class property

In our previous discussion, we tackle about the basic concepts of class, including creating it's state (*property*) and behavior (*methods*). We also discuss about simple validation of its class property by including validation code in its *Property Let*. Thus making your class more robust. We also discuss about some property variation, such as *read-only* property as well as *write-only* property. We also discuss decision making of using function (method) and property procedure in your class.

In this series articles, we are going to discuss about: Enumeration, Properties that return an Object and other semi-advance topic pertain to in creating a class base program.

An enumeration allows you to define your own set of named constants. A named constant is an item that preserve a constant value throughout the execution of a program and can be used in place of literal values. In other words, an *Enum* is nothing but a group of related constant values that automatically take different values. You can use named constants as property values, method arguments, and as a function's return values. By using a named constant it makes your code easier to read and maintain. For example, some properties are intended to return a well-defined subset of integer numbers. In our Student class, we can implement a *YearLevel* property that can be assigned the values 1 (Freshmen), 2 (Sophomore), 3 (Junior), and 4 (Senior).

```
' In the declaration section of the class
Enum YearLevelEnum
    Freshmen = 1
    Sophomore
    Junior
    Senior
End Enum
```

You don't need to assign an explicit value to all the items in the *Enum* structure, Visual Basic increments the preceding value automatically by 1. But because 0 is the default value for any Integer property when the class is created, Visual Basic starts at 0. It is a good programming style that you should always stay clear in assigning value to the *enum list*, so that you can later trap any value that hasn't been properly initialized . But *Enum* values don't need to be in an increasing sequence. In fact, you can provide special values for *Enum* constant list, as shown below:

```
' In the declaration section of the class
Enum GradeEnum
    Falling = 50
    Passing = 75
    Probationary = 84
    Except = 90
End Enum
```

After you define an *Enum* structure, you can create a Public or Private property of the corresponding type:

```
Private m_YearLevel As YearLevelEnum

Property Get YearLevel() As YearLevelEnum
```

```
    YearLevel= m_YearLevel
End Property
Property Let YearLevel(ByVal enumValue As YearLevelEnum)
    ' Refuse invalid assignments.
    If enumValue <= 0 Or enumValue > Senior Then Err.Raise 5
    m_YearLevel= enumValue
End Property
```

You should never forget that *Enum*s are just shortcuts for creating constants. This means that all the enumerated constants defined within an *Enum* block should have unique names in their scope. Typically *Enums* type are made by programmer to be Public, so their scope is often the entire application.

It is a good programming practice, that you should devise a method for generating unique names for all your enumerated constants. If you fail to do that, the compiler refuses to compile your application and raises the *"Ambiguous name detected: <itemname>"* error. The easy way to avoid this problem is to add to all the enumerated constants a unique 2 or 3 letter prefix, for example:

```
' In the declaration section of the class
Enum YearLevelEnum
    lvlFreshmen = 1
    lvlSophomore
    lvlJunior
    lvlSenior
End Enum
```

Another way to avoid ambiguous name problem is use the complete *enumname.constantname* syntax whenever you refer to an ambiguous *Enum* member, as in the following code:

```
Student.YearLevel = YearLevelEnum.lvlSenior
```

While enumerated properties are very useful and allow you to store some descriptive information in just 4 bytes of memory, sooner or later you will have to extract and decode this information and sometimes even show it to your users. For this reason, It is a good programming practice to add a *read-only* property that returns the description of an enumerated property:

```
Property Get YearLevelDescription() As String
    Select Case m_YearLevel
        Case lvlFreshmen : YearLevelDescription = "Freshmen"
        Case lvlSophomore: YearLevelDescription = "Sophomore"
        Case lvlJunior:    YearLevelDescription = "Junior"
        Case lvlSenior:    YearLevelDescription = "Senior"
        Case Else:         Err.Raise 5
    End Select
End Property
```

Another issue that you should never forget that your class are often change its structure if you are still in developing stage. So it is possible that your validation code can become outdated. For example, what happens if you later add a fifth *YearLevel* constant such *Graduate* or *Masteral*? For this reason, you should always add new constants safely without modifying the validation code in the corresponding *Property Let* procedure, one way to do this is create a *enum* constant as the highest value in that block and assign the value that you wanted to be the last *enum* constant, as shown below:

```
' In the declaration section of the class
Enum YearLevelEnum
    lvlFreshmen = 1
    lvlSophomore
    lvlJunior
    lvlSenior
    lvlGraduate                        ' newly added enum constant
    lvlMasteral                        ' newly added enum constant
    YEAR_LEVEL_MAX = lvlMasteral       ' make lvlMasteral the last enum constant
End Enum
Property Let YearLevel(ByVal enumValue As YearLevelEnum)
    ' Refuse invalid assignments
    If enumValue <= 0 Or enumValue > YEAR_LEVEL_MAX Then Err.Raise 5
    m_YearLevel= enumValue
End Property
```

As you can see we can safely add a new *enum* constant without worrying about the validation code in our *Property Let* to become obsolete. And making our maximum value in *uppercase* and putting a *comment*, we can easily spot it in our source code. Of course, you should account your read-only property description, because adding new *enum* constant in our *enum* block without adding appropriate description our *YearLevelDescription* property will result to an error, but at least we are safely notified.

```
' Modified version
Property Get YearLevelDescription() As String
    Select Case m_YearLevel
        Case lvlFreshmen:   YearLevelDescription = "Freshmen"
        Case lvlSophomore:  YearLevelDescription = "Sophomore"
        Case lvlJunior:     YearLevelDescription = "Junior"
        Case lvlSenior:     YearLevelDescription = "Senior"
        Case lvlGraduate:   YearLevelDescription = "Graduate"
        Case lvlMasteral:   YearLevelDescription = "Masteral"
        Case Else:          Err.Raise 5
    End Select
End Property
```

The addition of highest value for your *enum* list might confuse your user. You might want to hide this or decided not to show it from your user when they started to use your class. I common technique that you can use is by placing an underscore at the start of the *enum* list identifier as shown below:

```
Enum GenderEnum
   Male = 1
   Female
   [_GENDER MAX] = Female
End Enum
```

The square brackets [] are necessary because, Visual Basic will complain by raising a compile error: Invalid character. Adding square brackets permit us to add an *underscore* (_)at the beginning of the *enum* list identifier and brought us another useful technique, you can now add space to your *enum* list identifier as shown above. Unfortunately, even you, the author of the class cannot see this! So you must remember this *enum* list identifier when you use your class, especially in the *Property procedures* validation code.

One last thing that I can add pertaining to *enumerated* type. Sometimes, you need this description to populate a control in you client form, such as ComboBox and ListBox control. One technique that I used frequently, I usually change the implementation of description property by adding an *optional* ByVal parameter and then use the textual description to populate the control:

```
' In the Student class module
Public Sub LoadYearLevelDescriptionTo(ctrl As Control)
    Dim i As Integer
    ctrl.Clear
    For i = Freshmen To [_YEAR LEVEL MAX]
        ctrl.AddItem YearLevelDescription(i)
    Next i
End Sub

Property  Get  YearLevelDescription(Optional  ByVal  level  As  YearLevelEnum)  As
String
    Dim templevel As Long
    ' If argument level contain a value, use it in the Select Case,
    ' otherwise, use the Private m_YearLevel variable
    tempLevel = IIf(level = 0, m_YearLevel, level)
    Select Case tempLevel
        Case Freshmen:   YearLevelDescription = "Freshmen"
        Case Sophomore:  YearLevelDescription = "Sophomore"
        Case Junior:     YearLevelDescription = "Junior"
        Case Senior:     YearLevelDescription = "Senior"
        Case Graduate:   YearLevelDescription = "Graduate"
        Case Masteral:   YearLevelDescription = "Masteral"
        Case Else:          Err.Raise 5
    End Select
End Property


' In your client form
' Load Year level description in ListBox Control
Student.LoadYearLevelDescriptionTo List1
or
' Load Year level description in ListBox Control
Student.LoadYearLevelDescriptionTo Combo1
```

See, how easy it would be in the client, I don't have to populate the *List* property of the Combo or ListBox control in the client form, all I have to do is to call LoadYearLevelDescriptionTo to do the work. Note, you can still use the *Property YearLevelDescription* without an argument:

```
' Return the Year Level Description of this particular Student
Debug.Print Student.YearLevelDescription
```

In addition to (*enumerated*) class properties, our class objects might expose properties that return object values. To give you an example, Visual Basic object such as forms and visible controls expose a *Font* property, which returns a Font object.

```
txtFirstName.Font.Name = "Tahoma"
txtFirstName.Size = 10
txtFirstName.Bold = True
frmStudent.Font.Bold = True
```

We can also do this in our classes. Taking our Student class, we might add a *Address* property, but string is not enough to point accurately where the student lives, and we usually need several pieces of related information, such *street*, *city*, *state* or *province*, *zip code*, as well as *country*. Instead of adding multiple properties to the Student object, create a new Address class:

- **To create an Address class**

    1. **On the Project menu, click Add Class Module.**

2. **In the Add Class Module dialog box, select Class Module, then click Open.**

3. **In the Properties window, set the Name property for the class module to Address.**

4. **In the Code window, type the following:**

```
Option Explicit

' In Address class module declaration
Private m_Street As String
Private m_City As String
Private m_State As String
Private m_Zip As String
Private m_Country As String

Public Property Let Street(ByVal strNewStreet As String)
    If Len(strNewStreet) = 0 Then Err.Raise 5
    m_Street = strNewStreet
End Property

Public Property Get Street() As String
    Street = m_Street
End Property

Public Property Let City(ByVal strNewCity As String)
    If Len(strNewCity) = 0 Then Err.Raise 5
    m_City = strNewCity
End Property

Public Property Get City() As String
    City = m_City
End Property

Public Property Let State(ByVal strNewState As String)
    If Len(strNewState) = 0 Then Err.Raise 5
    m_State = strNewState
End Property

Public Property Get State() As String
    State = m_State
End Property

Public Property Let Zip(ByVal strNewZip As String)
    If Len(strNewZip) = 0 Then Err.Raise 5
    m_Zip = strNewZip
End Property

Public Property Get Zip() As String
    Zip = m_Zip
End Property

Public Property Let Country(ByVal strNewCountry As String)
    If Len(strNewCountry) = 0 Then Err.Raise 5
    m_Country = strNewCountry
End Property
```

```
Public Property Get Country() As String
    Country = m_Country
End Property

Public Function CompleteAddress() As String
    CompleteAddress = Street & vbCrLf & _
                      City & ", " & State & " " & Country & " " & Zip
End Function
```

Now you can add our new *Address* property to our `Student` class in declaration section of the *Student* class module:

```
'In the declaration section of the Student class module
'Enum type declaration omitted

Private m_Student_ID As String
Private m_FirstName As String
Private m_LastName As String
Private m_MajorCode As MajorCodeEnum
Private m_YearLevel As YearLevelEnum
Private m_BirthDate As Date
Private m_Gender As GenderEnum
Private m_Address As Address          ' Student address
```

Property Set procedures

A `Property Set` procedure sets the value of a property that contains a reference to an *object*. When you assign a value to an object, you must use the Visual Basic *Set* statement. An example of a property which is an object itself would be the *Font* property of the *TextBox* control. Because you're dealing with object references, you must use the `Set` keyword in both procedures. Add the following property procedure and additional method, as well as the revised version of `StudentInfo` method in Student class module:

```
' Student Address property procedures
Property Get Address() As Address
   Set Address = m_Address
End Property

Property Set Address(ByVal strNewAddress As Address)
   Set m_Address = strNewAddress
End Property

' New Student method
Function StudentAddressInfo() As String
    If m_Address Is Nothing Then Err.Raise 5
    StudentAddressInfo = m_Address.CompleteAddress
End Function

' Student StudentInfo method revised
Function StudentInfo(Optional ByVal IncludedAddressInfo As Boolean = True) As
String
' Returns the Student information
    Dim info As String

    info = "Student # : " & StudentID & vbCrLf & _
           "Name : " & FullName & vbCrLf & _
           "Age : " & Age & vbCrLf & _
           "Gender : " & GenderDescription & vbCrLf & _
           "Major Code : " & MajorCode & vbCrLf & _
           "Major Description: " & MajorCodeDescription & vbCrLf & _
           "Year Level : " & YearLevelDescription

    If IncludedAddressInfo Then info = info & "Address : " & StudentAddressInfo()

    StudentInfo = info
End Function
```

In our new `StudentAddressInfo` method, it is a good programming practice that you check first the existence of an object (`Address`) with in an object (`Student`), because a call to that method will raise an error number 91.
Now you can create a `Address` object in client form, initialize its properties, and then assign it to the *Address* property of the `Student` object.

```
' In client form
' Declare Student object and Address Object
Dim Student As Student
Dim Address As Address

' Initiate the object Student
Set Student = New Student

' Initiate the object Address
Set Address = New Address

' Set up Address properties
With Address
   .Street = "Block 10 Lot 26, Molave Street, Calendola Village"
   .City = "San Pedro"
   .State = "Laguna"
   .Country = "Philippines"
   .Zip = "4023"
End With

' Set up Student pproperties
With Student
   .FullName = "Dante Salvador"
   ' Add the newly created Address object to Student Address property
   Set .Address = Address
   .StudentID = "102472"
   .BirthDate = #10/24/1972#
   .Gender = Male
   .YearLevel = Senior
   .Major = BSCS
End With

' Show Student information
MsgBox Student.StudentInfo
```

You can add flexibility to your class by including a Variant member. Assuming that you want to implement a *ProvincialAddress* property, but you want to keep it more flexible and capable of storing either a `Address` object or a string. Now let us add *ProvincialAddress* property to our Student class as shown below:

```
Private m_ProvincialAddress As Variant

Property Get ProvincialAddress() As Variant
    If IsObject(m_ProvincialAddress) Then
        Set ProvincialAddress = m_CurrentAddress   ' Return a Address object.
    Else
        ProvincialAddress = m_ProvincialAddress    ' Return a string.
    End If
End Property

Property Let ProvincialAddress(ByVal strNewProvincialAddress As Variant)
    m_ProvincialAddress = strNewProvincialAddress
End Property

Property Set ProvincialAddress(ByVal strNewProvincialAddress As Variant)
    Set m_ProvincialAddress = strNewProvincialAddress
End Property

' Revised StudentInfo method
Function StudentInfo(Optional ByVal IncludeAddressInfo As Boolean = True, _
                     Optional  ByVal  IncludeProvincialAddressInfo  As  Boolean  =
False) As String
' Returns the Student information
    Dim info As String

    info = "Student # : " & StudentID & vbCrLf & _
                  "Name : " & FullName & vbCrLf & _
                  "Age : " & Age & vbCrLf & _
                  "Gender : " & GenderDescription & vbCrLf & _
                  "Major Code : " & MajorCode & vbCrLf & _
                  "Major Description: " & MajorCodeDescription & vbCrLf & _
                  "Year Level : " & YearLevelDescription

    If  IncludeAddressInfo  Then  info  =  info  &  vbCrLf  &  "Address  :  "  &
StudentAddressInfo()
    If IncludeProvincialAddressInfo Then info = info & vbCrLf & "Provincial Add
: " & _
                                        StudentProvincialAddInfo()

    StudentInfo = info
End Function

' Newly added method for Student class
Function StudentProvincialAddInfo() As String
' Return Student Provincial address
    If IsObject(m_ProvincialAddress) Then
       ' invoke Address CompleteAddress method
        StudentProvincialAddInfo = m_ProvincialAddress.CompleteAddress
    Else
       ' simply return a string
       StudentProvincialAddInfo = m_ProvincialAddress
    End If
End Function
```

But things are a bit more complex if the property can receive either a regular value or an object value. While this sort of flexibility adds a lot of power to your class, it also reduces its robustness

because nothing keeps a programmer from adding a nonstring value or an object of a class other than Address:

```
'In the client form

With Student
   .FullName = "Dante Salvador"
   Set .Address = 12345                ' an Integer value
   .StudentID = "102472"
   'etc
End With
```

Because *ProvincialAddress* property is declared as Variant type, meaning you can assign any value, including numeric type. To have more control of what is actually assigned to this property, you need to arbitrate all accesses to it through *Property* procedures:

```
' In Student class module
' Revised property procedures
Property Let ProvincialAddress(ByVal strNewProvincialAddress As Variant)
    ' Check if it is a string value.
    If VarType(strNewProvincialAddress) <> vbString Then Err.Raise 5
    m_ProvincialAddress = strNewProvincialAddress
End Property

Property Set ProvincialAddress(ByVal strNewProvincialAddress As Variant)
    ' Check if it is a Address object.
    If TypeName(strNewProvincialAddress) <> "Address" Then Err.Raise 5
    Set m_ProvincialAddress = strNewProvincialAddress
End Property

'In the client form
With Student
   .FullName = "Dante Salvador"
   Set .Address = 12345                ' this raises an error
   .StudentID = "102472"
   'etc
End With
```

Another technique that you can use that give slightly improve run-time performances and you save some code is to declare the type of the object you're expecting right in the parameter list of the *Property Set* procedure:

```
Property Set ProvincialAddress(ByVal strNewProvincialAddress As Address)
    ' Check if it is a Address object.
    If TypeName(strNewProvincialAddress) <> "Address" Then Err.Raise 5
    Set m_ProvincialAddress = strNewProvincialAddress
End Property
```

But you can't use it when if your class accept two or more objects of different types. One solution is use *As Object* parameter:

```
Property Set ProvincialAddress(ByVal strNewProvincialAddress As Object)
    If TypeName(strNewProvincialAddress) <> "Address" And _
       TypeName(strNewProvincialAddress) <> "OtherAddressType" Then Err.Raise 5
    Set m_CurrentAddress = newValue
End Property
```

Object Keyword

The *TypeName* function

The *TypeName* function returns the name of an object's class in the form of a string. This means that you can find the type of an object in a more concise form

In many situations, testing an object's type using the *TypeName* function is preferable to using the *TypeOf...Is* statement because it doesn't require that the object class be present in the current application or in the References dialog box. For information about *TypeOf...Is* function, consult your Visual Basic documentation.

The *VarType* function

The *VarType* function returns the type name of an object's class in the form of a string. Variant variables can also host special values that don't correspond to any data values described so far. The *Empty* value is the state of a Variant variable when nothing has been assigned to it yet. You can test this special value using the *IsEmpty* function, or you can test the *VarType* function for the value 0-vbEmpty.

The Null value is useful in database programming to mark fields that don't contain a value. You can explicitly assign the Null value to a Variant using the Null constant, test for a Null value using the *IsNull* function, or compare the return value of the *VarType* function with the value 1-vbNull. For more information about this function, consult your Visual Basic documentation.

In our previous example, our *Address* property of the `Student` class is responsible for proper format of the Student address. For instance, let us pretend that it takes a lot of processing time to evaluate its result and return the value. If you trace the execution of the program it executes all the `Property Get` in order to construct the correct address format, which we're surely like that the class would do. But add the `Debug.Print` statement below:

```
' Other client form code omitted
' Show Student information in the client form
MsgBox Student.StudentInfo(, True)

' Call the Student StudentAddressInfo method
Debug.Print Student.StudentAddressInfo        ' Add this code
```

The code above demonstrate the overhead of calling Student address even if we don't change address information of the student. In other words, the class (`Address`) always evaluate (run) the `Property Get` even if it is not necessary to do so because it highly dependent on the independet (such as *Street*, *City*, *State*, *Zip* and *Country*) *Property* value. So how can we modify this function to keep the overhead to a minimum without modifying the interface that the class exposes to the outside. A common sense solution we case use is don't to reevaluate it (all independent property, such as Street, City, State, Zip and Country) each time the client code makes a request. Right! But how? We can store the return value in a `Private Variant` variable before returning to the client and reuse that value if possible in all subsequent calls. The trick works because each time either *Street*, *City*, *State*, *Zip* or *Country* are assigned a new value, the `Private` variable is cleared, which forces it to be reevaluated the next time the *CompleteAddress* function is invoked.

```
' In Address class module declaration
' Other Private member variable omitted
Private m_CompleteAddress As Variant
Public Property Let Street(ByVal strNewStreet As String)
    If Len(strNewStreet) = 0 Then Err.Raise 5
    m_Street = strNewStreet
    m_CompleteAddress = Empty  ' add this line in every Property Let Procedures
End Property
' Other Property Let procedure ommited
' Revised CompleteAddress Method
Public Function CompleteAddress() As String
    If IsEmpty(m_CompleteAddress) Then
        m_CompleteAddress = Street & vbCrLf & City & ", " & _
                            State & " " & Country & " " & Zip
    End If
    CompleteAddress = m_CompleteAddress
End Function
```

If trace again the program execution (Pressing F8), the second time you invoke the *CompleteAddress* method (via *StudentAddressInfo* method of Student class), the class smartly save the previous result and return it. You might ask "We implement this technique to the Address object, why we did not implement this to Student class, is it possible?" Yes of course, but you must understand that a class should be robust, and to be a robust class, the class should be responsible for himself! As you can see, even if the *CompleteAddress* method are highly dependent on independent properties, it access it dependent property in Address class in which *CompleteAddress* is also included. Now lets go back to the topic, one last note, don't underestimate the advantage of this technique, because in a real-world application, this difference might involve unnecessarily opening a database, reestablishing a remote connection, and so on.

Initialization Method

We already explained that the class to be robust, it must always contains a valid value. And to achieve this objectives, we provide our class a *Property procedures* and *methods* to transform the internal data of the class to a valid state by providing validation code inside this procedures. However, if you are familiar with C++ and Java, you might be asking, what if an object is used immediately after creation of the object or during the creation of the object in the client side and how can we provide the user or client a initial valid values? We can provide the client some useful initial valid value in the *Class_Initialize* event procedure, without having to specify it in the client code. Visual Basic offers a neat way by writing some statements in the *Class_Initialize* event of the class module. To have the editor create a template for this event procedure, you select the Class item in the leftmost combo box in the code editor. Visual Basic automatically selects the *Initialize* item from the rightmost combo box control and inserts the template into the code window.
Because we are dealing with the Student class object, you can provide a reasonable value for client to expect in its *Country* property of the *Address* object property of the Student. For example, "Philippines" or whatever nationality appropriate where you live. In this, you would like for these default values to be assigned when you create an object, rather than, having assign them manually in the code that uses the class.

```
' In Address class module
Private Sub Class_Initialize()
    m_Country = "Philippines"
End Sub
```

If you trace the program, you will see that as soon as Visual Basic creates the object (the *Set* command in the form module), the *Class_Initialize* event fires. The object is returned to the caller with all the properties correctly initialized, and you don't have to assign them in an explicit way.
But this solution might be not enough for us. We just solve the second problem stated above. What happens if an object is used immediately after its creation. Consider this example:

```
' In the Client form
Set Student = New Student
Debug.Print Student.FirstName ' << this will display nothing
Debug.Print Student.FullName  ' << this will raise an error
```

In other programming language, this problem is solved by defining a special procedure that is defined in the class module and executed whenever a new instance is create, just like C++ and Java constructor. Because Visual Basic completely lack of constructor method, you can't prevent the user of your class from using the object as soon as they create it. The best solution that you can do is create simulated constructor method that correctly initialize all (if you desire) the properties and let the user know that they can initialize the object in a short way.

```
' In the Address class module
Public Sub InitAddress(Optional ByVal Street As Variant, _
                       Optional ByVal City As Variant, _
                       Optional ByVal State As Variant, _
                       Optional ByVal Zip As Variant, _
                       Optional ByVal Country As Variant)

    If Not IsMissing(Street) Then Me.Street = Street
    If Not IsMissing(City) Then Me.City = City
    If Not IsMissing(State) Then Me.State = State
    If Not IsMissing(Zip) Then Me.Zip = Zip
    If Not IsMissing(Country) Then Me.Country = Country

End Sub

' In the Student class module
Public Sub InitStudent(Optional ByVal StudentID As Variant, _
                       Optional ByVal FirstName As Variant, _
                       Optional ByVal LastName As Variant, _
                       Optional ByVal Major As MajorCodeEnum = Freshmen, _
                       Optional ByVal YearLevel As YearLevelEnum = BSCS, _
                       Optional ByVal BirthDate As Variant, _
                       Optional ByVal Gender As GenderEnum = Male, _
                       Optional ByVal Address As Variant, _
                       Optional ByVal ProvincialAddress As Variant)

    If Not IsMissing(StudentID) Then Me.StudentID = StudentID
    If Not IsMissing(FirstName) Then Me.FirstName = FirstName
    If Not IsMissing(LastName) Then Me.LastName = LastName
    If Not IsMissing(Major) Then Me.Major = Major
    If Not IsMissing(YearLevel) Then Me.YearLevel = YearLevel
    If Not IsMissing(BirthDate) Then Me.BirthDate = BirthDate
    If Not IsMissing(Gender) Then Me.Gender = Gender
    If Not IsMissing(Address) Then _
       Set Me.Address = Address     ' Set command is necessary
    If Not IsMissing(ProvincialAddress) Then _
       Set Me.ProvincialAddress = ProvincialAddress ' also here

End Sub
```

Now you can tell the user of your class, to use your newly created simulated constructor:

```
' In the Client form
' Initiate the object Student
Set Student = New Student
Set Address = New Address
Set ProvincialAdd = New Address

' Set up Address
Address.InitAddress "Block 10 Lot 26, Molave Street, Calendola Village", _
                    "San Pedro", _
                    "Laguna", _
                    "4023"

' Set up Provincial address
ProvincialAdd.InitAddress "Block 10 Lot 26, Molave Street, Calendola Village", _
                          "San Pedro", _
                          "Laguna", _
                          "4023"

' Set up Student
Student.InitStudent "12345", "Dante", "Salvador", _
                    BSCS, Senior, #10/24/1972#, Male, _
                    Address, ProvincialAdd

' Other code omitted
```

As you can see, we adopt optional arguments of type Variant because it is essential that you use the IsMissing function and bypass the assignment of values that were never provided by the client. The good consequence of this approach is that, we can use default value to the parameter list as shown in InitStudent method. We also use the same names of the properties they refer to, this makes the method easier to use and to avoid name conflict inside the procedure, we use Me keyword to refer to the real properties of the class.
Now, to add more usability of your class, you can provide a function in a BAS module in your application that return a newly created object of your class:

```
' In the Standard module of your application
Public Function New_Student(Optional ByVal StudentID As Variant, _
                            Optional ByVal FirstName As Variant, _
                            Optional ByVal LastName As Variant, _
                            Optional ByVal Major As MajorCodeEnum = Freshmen, _
                            Optional ByVal YearLevel As YearLevelEnum = BSCS, _
                            Optional ByVal BirthDate As Variant, _
                            Optional ByVal Gender As GenderEnum = Male, _
                            Optional ByVal Address As Variant, _
                            Optional  ByVal  ProvincialAddress  As  Variant)  As
Student

    ' Initiate an object Student
    Set New_Student = New Student

    ' Call InitStudent method
    New_Student.InitStudent StudentID, FirstName, LastName, MAJOR_CODE_MAX, _
                            YearLevel, BirthDate, Gender, _
                            Address, ProvincialAddress
End Function

Public Function New_Address(Optional ByVal Street As Variant, _
                            Optional ByVal City As Variant, _
                            Optional ByVal State As Variant, _
                            Optional ByVal Zip As Variant, _
                            Optional ByVal Country As Variant) As Address

    ' Initiate an Address object
    Set New_Address = New Address

    ' Call InitAddress method
    New_Address.InitAddress Street, City, State, Zip, Country

End Function
```

See how concise your code in the client form:

```
' In client form
' Declare Student object
Dim Student As Student
Dim Address As Address
Dim ProvincialAdd As Address

' Initiate and create Address object
Set Address = New_Address("Block 10 Lot 26, Molave Street, " & _
                          "Calendola Village", _
                          "San Pedro", _
                          "Laguna", _
                          "4023")

' Initiate and create Provincial Address object
Set ProvincialAdd = New_Address("Block 10 Lot 26, Molave Street, " & _
                                "Calendola Village", _
                                "San Pedro", _
                                "Laguna", _
                                "4023")

' Initiate and create Student object
Set Student = New_Student("12345", "Dante", "Salvador", _
                           BSCS, Senior, #10/24/1972#, Male, _
                           Address, ProvincialAdd)
' Show Student information
MsgBox Student.StudentInfo(, True)
```

You can add a little spice to your function, by assigning the *Address* property value to the *ProvincialAddress* property, if the `Student` lives in the same address.

```
' In the Standard module of your application
Public Function New_Student(Optional ByVal StudentID As Variant, _
                            Optional ByVal FirstName As Variant, _
                            Optional ByVal LastName As Variant, _
                            Optional ByVal Major As MajorCodeEnum = Freshmen, _
                            Optional ByVal YearLevel As YearLevelEnum = BSCS, _
                            Optional ByVal BirthDate As Variant, _
                            Optional ByVal Gender As GenderEnum = Male, _
                            Optional ByVal Address As Variant, _
                            Optional  ByVal  ProvincialAddress  As  Variant)  As
Student
    ' Initiate an object Student
    Set New_Student = New Student

    ' Assign the same adddress if ProvincialAddress is not set
    If IsMissing(ProvincialAddress) Then Set ProvincialAddress = Address

    ' Call InitStudent method
    New_Student.InitStudent StudentID, FirstName, LastName, MAJOR_CODE_MAX, _
                            YearLevel, BirthDate, Gender, _
                            Address, ProvincialAddress
End Function

' In client form
Dim Student As Student
Dim Address As Address

' Initiate and create Address object
Set Address = New_Address("Block 10 Lot 26, Molave Street, " & _
                          "Calendola Village", _
                          "San Pedro", _
                          "Laguna", _
                          "4023")

' Initiate and create Student object
Set Student = New_Student("12345", "Dante", "Salvador", _
                           BSCS, Senior, #10/24/1972#, Male, _
                           Address)
' Show Student information
MsgBox Student.StudentInfo(, True)

' Change the provincial address if you will
With Student
  With .ProvincialAddress
     .Street = "830 Euclid Avenue"
     .City = "Cleveland"
     .State = "Ohio"
     .Zip = "44114"
     .Country = "USA"
  End With
End With

' Show Student provincial address info
MsgBox Student.StudentProvincialAddInfo
```

In this article, we talk about enumerated properties, how we can use it to make our class more readable and can be easily maintain. We also discusses about properties that return an object and learn the use of another property procedure, Property Set. We also talk about properties that return a variant type and how to make our class more flexible. Lastly, we talk about some advance useful method to add functionality to our class.

The next article tackles about Collection in general and how to incorporate collection to make class mimic the functionality of Collection object.