Chapter 16 – Object-Oriented Programming in VB

# Objectives

* Understanding Object-Orientation in VB
* Overview of Classes in VB
* Simple Class Creation
* Inheritance Types
* A Polymorphic Example
* Summary on Basic Object-Orientation Keywords
* Firing Events and Exceptions from your Own Class
* Understanding Interfaces
* The DoEvents Statement
* Delegates
* Namespaces

This chapter will introduce the Object-Oriented Programming (OOP) aspects of Visual Basic programming. The original version of VB.NET was the first version of the language to have the full repertoire of OOP constructs available to VB developers. Prior versions of the language supported basic concepts such as classes, but many other key OOP pieces were not present at all or only partially implemented.

This chapter assumes that you understand what Object-Orientation is all about. It is not the purpose of this chapter to teach you detailed object-oriented design nor software construction. Rather, this chapter focuses on showing OOP developers how VB implements the constructs that are common in other OOP languages.

If you haven't worked with object-oriented languages before there are many excellent books on the topic. In my opinion, if you haven't been programming in an OOP language, you are really missing a vital development philosophy that can save you time and effort in design and coding, not to mention drive the reusability of your code. If you are serious about evolving your programming abilities and you’re wondering where to go next after this class, design patterns and software architecture would be the next stop. Both are highly reliant on your ability to effectively wield object-orientation.

# Understanding Object-Orientation in VB

It ought to be obvious to you that objects and classes are everywhere; even if you didn't realize it, most of the controls you have been working with exhibit object-orientation since they have properties and methods. Create a form and look at the code behind the scenes and you’ll see it starts out with the word Class. Therefore, classes in VB can consist of both methods and attributes as well as having events attached to them – this shouldn't be that strange since you’ll note that all of your forms have events and are classes.

Anything that you have learned about, or done object-orientedwise, in other programming languages can also be implemented in Visual Basic. This chapter will examine the various object-oriented aspects that exist. The assumption is that you have seen at least one OOPL before, so you should just need to see syntactically how things work in VB land.

## 

# Overview of classes in VB

Classes in VB are usually created in class modules (remember, though, that a form is a class too, so classes can live pretty much anywhere where you can type code in). Classes in VB are very similar in concept to classes in other high-level OOP languages.

Here's a quick summary that discusses the various OO aspects of a VB.NET class:

* There is constructor and destructor subroutines available which are called New and Finalize. .NET does automatic garbage collection of unreferenced objects just like Java does.
* Methods in a class can be made private (only the class can see these), public (everyone can see these), or friend (only the module that the class is in can see these) accessible. There are a couple of nuances to this, but we’ll talk those in a bit.
* All data members/attributes in a class should be kept private. Therefore, you will write associated accessor methods/properties to put data into and pull data from the private backing data members– remember that this enforces the concept of data hiding. The language doesn’t require it: good object-oriented design does!
* Objects need to be instantiated before they can be used. This is done with the New keyword – you should be really good with this one by now since we've been using it right along.
* Objects can be destroyed at any time programmatically. This is done in VB by setting the object = Nothing.
* Child classes can be derived, or inherited, from a parent class. Each child class can only be derived from a single parent.
* Methods in a class may be overloaded (more than one version with different parameter signatures and the same purpose) and overridden (same parameter signatures but a different purpose).
* Abstract classes can be created. You cannot create instances of abstract classes; instead you create child class implementations of the abstract class and instantiate those child classes.
* Interfaces can be created in a class. Interfaces are a guaranteed set of methods to be realized by a class when it agrees to implement a particular interface.
* There is no multiple inheritance in VB, which is identical to Java and unlike C++. Multiple interfaces may be implemented however.

# Simple Class Creation

Let’s jump right into a VB object-oriented code example. Let's look at creating an employee object – this will be in a class module called clsEmployee. You begin by adding a new class module under Project🡪Add Class. You should spend some time examining this code since you will see the implementation of several key object-oriented features.

'Chapter 16 - Program 1

'This indicates that we are creating a new class called clsEmployee

'which will be publically accessible

Public Class clsEmployee

'These are the internal data members that make up the

'class -- we will write a Get and Set method to actually

'put and pull data from these members. Note the keyword

'Private...

Private txtName As String

Private intAge As Integer

Private sngWage As Single

Private intHoursWkd As Integer

Private sngGrossPay As Single

'This is the default constructor -- notice we will use the

'methods we wrote for this class on itself --> this is one

'of the key OO concepts

Public Sub New() 'Instantiate to our "known" defaults

setName("")

setAge(0)

setWage(0.0)

setHoursWkd(0)

setGrossPay(0)

End Sub

'This is a named constructor; i.e. the user called an instantiation

'of this object with parameters provided. Notice that there are

'some private values that we still instantiate to their defaults.

'

'Also notice that this method is called New, just as the previous

'one was -- this is overloading a method: same method name, but

'different parameter signatures are used.

Public Sub New(ByVal newName As String, ByVal newAge As Integer,

ByVal newWage As Single)

setName(newName)

setAge(newAge)

setWage(newWage)

setHoursWkd(0)

setGrossPay(0)

End Sub

'So to work with a private data member, we write accessors.

'We can choose to handle them through manual gets/sets.

'In a future example, we learn that we can do this through

'Properties and make VB write most of the code...

'

'Set is used to assign a value to one of the members

'Notice that setName is a method (subroutine) of the class

Public Sub setName(ByVal newName As String)

txtName = newName

End Sub

'Get will be used to return the value from one of the members

'In this case, again, getName is a method (but afunction in

'this case) of our class.

Public Function getName() As String

Return (txtName)

End Function

'Here’s the routines for the Age member

Public Sub setAge(ByVal newAge As Integer)

intAge = newAge

End Sub

Public Function getAge() As Integer

Return (intAge)

End Function

'Here’s the routines for the Wage member

Public Sub setWage(ByVal newWage As Single)

sngWage = newWage

End Sub

Public Function getWage() As Single

Return (sngWage)

End Function

'Here’s the routines for the HoursWkd member

Public Sub setHoursWkd(ByVal newHoursWkd As Integer)

intHoursWkd = newHoursWkd

End Sub

Public Function getHoursWkd() As Integer

Return (intHoursWkd)

End Function

'Notice that we provide a method called GrossPay which can be

'queried from the outside, but only set inside the class...

'This is information hiding: only letting the programmer see

'what he/she needs to see. Once we get to Properties, we'll

'see that there is a different way to implement this type

'of information hiding.

Private Sub setGrossPay(ByVal newGrossPay As Single)

sngGrossPay = newGrossPay

End Sub

Public Function getGrossPay() As Single

Return (sngGrossPay)

End Function

'Finally let’s write a method that computes the employee’s wage

'We will get our data values through the method interfaces, not by

'directly referencing the private members! Think OO!!!!!!!

Public Sub ComputeWage()

If (getHoursWkd() > 40.0#) Then

setGrossPay((40 \* getWage()) + ((getHoursWkd() - 40) \* getWage() \*

1.5))

Else

setGrossPay(getHoursWkd() \* getWage())

End If

End Sub

End Class

Now, let’s exercise the class we created through a plain old code module that creates some instances of our clsEmployee class:

Module Module1

'Chapter 16 -- Program 1

Sub Main()

'Create two instances of clsEmployee

'The first is instantiated using the default constructor

Dim aCust As New clsEmployee()

'The second one is instantiated using a named constructor

Dim anotherCust As New clsEmployee("Scott", 34, 15.0)

'Use some of the set methods to put values into the private

'data members

aCust.setHoursWkd(10)

anotherCust.setHoursWkd(10)

'Call a method in each class

aCust.ComputeWage()

anotherCust.ComputeWage()

'Show that the ComputeWage method actually did some work --

'remember that there is no way to publically set the GrossPay.

'It only returns a numeric value for us.

Debug.WriteLine("aCust makes " & aCust.getGrossPay())

Debug.WriteLine("anotherCust named " & anotherCust.getName() &

" makes " & anotherCust.getGrossPay())

End Sub

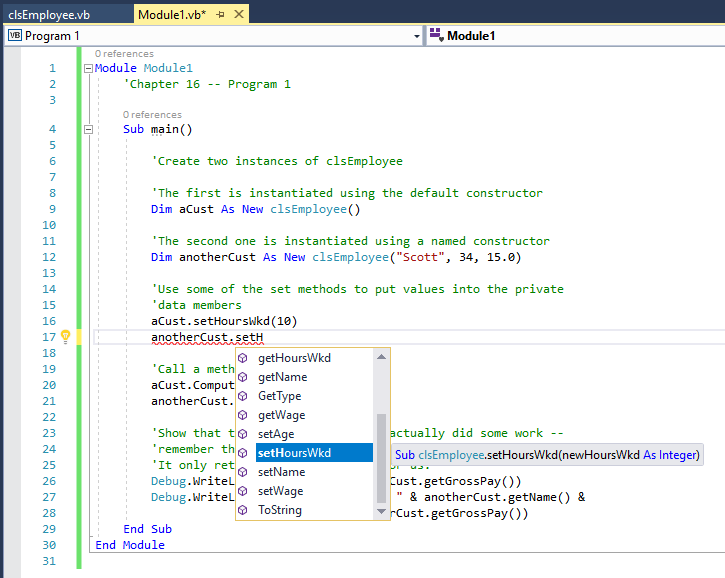
End Module

Here’s the program’s output, which is what we’d expect to see:

aCust makes 0

anotherCust named Scott makes 150

One of the really cool features of the Visual Studio IDE is that it learns your classes’ methods and properties as soon as you code them. The IntelliSense syntax auto completion functionality becomes available and will show you the properties and methods that are available inside of your own classes as you’re writing code:



Now, let’s try to be a bad programmer and attempt to put a value in via the GrossPay accessor. As you recall, this member is not publicly accessible for setting values: aCust.setGrossPay(200.00)

We can’t even compile the project. In the Error List window, we are greeted with a nice message: 

This means that the class works as we would expect it should and that we have protected our class from people trying to externally put values in where they shouldn't. Only our class can internally set the GrossPay amount…

Here is a list of the various scope levels that you can define when working with methods and data members/properties in classes. As promised there’s a bit more than the public, private and friend levels:

|  |  |
| --- | --- |
| *Scope Level* | *Purpose* |
| Friend | The member/method is only available inside of the current project so anyone deriving from your project cannot see it |
| Private | The member/method is available only inside of the current class |
| Protected | The member/method is only available within the current class and classes that derive from the class |
| Protected Friend | The member/method is available within the current project and within anything that derives from the project |
| Public | The member/method is available outside of the class |
| Shared | There is only one copy of the member/method created and it is shared among all instances of a particular class |

# Inheritance Types

Now that we've seen how to create a simple class, let's build an example that illustrates both kinds of inheritance. If you don't remember the idea of inheritance, there are technically two types: Is-a and Has-a. Is-a deals with creating a new class that is derived from an existing class. The new class will have everything that the original class had plus some features that differentiate the new class from the original class. Has-a on the other hand is where one class is referencing another class. Has-a comes in a couple of forms:

* Aggregation – our parent class points to a child class instance that lives independently of the parent. In other words, think of this as where a copy of some child object type was passed in to our parent object. Our parent object may be destroyed, but the passed in child object will live on after our parent object’s termination.
* Composition – our parent class points to a child class whose existence was created by the parent. This means that when our parent object is destroyed, so is the child object.
* Association – all this means is that one object is using another. For example, your class called MessageBox.Show – you’re using MessageBox but you didn’t create it nor did you get a copy of it passed in. There’s no “ownership” at play in Association as there is in Aggregation and Composition.

I do not expect you to be an expert on the fine distinctions of the Has-a type inheritance. There are conflicting definitions of these items in textbooks and online. What I do care about is that you understand the difference between Is-a and Has-a. You will end up using both types in your object-oriented designs. As far as what kind of Has-a a design implements, worry about that another day (if it’s even worth worrying about). I kind of chuck this into the same category as database normalization: Is being able to delineate the various forms *really* that critical to being able to build good software? Probably not to beginners! Probably even questionable for seasoned developers…

What we will do in our second programming example is create a new class called clsValuableCustomer that is derived from our clsCustomer class. Our clsValuableCustomer objects will have everything that our clsCustomer objects have, but because they are Valuable Customers, they will also have a credit line and a bank name where they keep their accounts. This is an example of Is-a inheritance.

We will illustrate Has-a inheritance by creating an Address class. Every customer, regardless of what type of customer they are will have an Address – this is Has-a at play.

Here's the code for all three of our classes and the main module that will create instances and exercise the classes. We’ll begin by looking at the Customer class:

'Chapter 16 - Program 2

'There are only a couple of changes to clsCustomer (which is just

'our clsEmployee class from the last program) -- we added

'a Finalize method and a TellAboutMe method. I also illustrated how

'to get VB to write the Gets/Sets by changing from manually writing

'them to using Properties

Public Class clsCustomer

'Private data members of the class

Private strName As String

Private intAge As Integer

Private sngWage As Single

Private intHoursWkd As Integer

Private sngGrossPay As Single

'We will define this class in a separate file in a minute...

'By the way since we are holding another class type in our

'class, this is a Has-a inheritance

Private CustomerAddress As clsAddress

'The default constructor

Public Sub New()

Name = ""

Age = 0

Wage = 0.0

HoursWkd = 0

setGrossPay(0)

'So our Has-a is technically composition since the

'address instance gets created here and will be destroyed

'when the customer instance gets destroyed

CustomerAddress = New clsAddress

End Sub

'A named constructor

Public Sub New(ByVal newName As String, ByVal newAge As Integer,

ByVal newWage As Single, ByVal newAddress As String,

ByVal newCity As String, ByVal NewState As String,

ByVal newZip As String)

Name = newName

Age = newAge

Wage = newWage

HoursWkd = 0

setGrossPay(0)

CustomerAddress = New clsAddress(newAddress, newCity, NewState, newZip)

End Sub

Protected Overrides Sub Finalize()

'This is VB's version of a destructor...

'Notice the Overrides, this says that we are going to rewrite a

'local version of a method that belongs to our object's parent,

'Object.

'Print out a message that we are in the process of destroying

'a clsCustomer instance

Debug.WriteLine("An instance of clsCustomer is being destroyed")

Try

'Call the parent object's finalizer, in this case it would

'be Object.Finalize -- remember Object is the parent of

'all objects. MyBase is a convenient way to refer to

'the parent without having to explicitly name the parent.

MyBase.Finalize()

Catch Ex As Exception

'we're leaving so don't worry about it

End Try

End Sub

'To see the effectiveness of this, I typed

'Public Property Name As String (return) Get

'and then VB types in the rest of the construct

'-- huge timesaver for the humdrum work of

'writing accessors!

Public Property Name As String

Get

Return strName

End Get

Set(ByVal value As String)

strName = value

End Set

End Property

Public Property Age As Integer

Get

Return intAge

End Get

Set(ByVal value As Integer)

intAge = value

End Set

End Property

Public Property Wage As Single

Get

Return sngWage

End Get

Set(ByVal value As Single)

sngWage = value

End Set

End Property

Public Property HoursWkd As Integer

Get

Return intHoursWkd

End Get

Set(ByVal value As Integer)

intHoursWkd = value

End Set

End Property

'We only can "get" GrossPay so notice the ReadOnly

'on this Property

Public ReadOnly Property GrossPay As Single

Get

Return sngGrossPay

End Get

End Property

'We will still need to write a private method to set

'the gross pay

Private Sub setGrossPay(ByVal newGrossPay As Single)

sngGrossPay = newGrossPay

End Sub

Public Sub ComputeWage()

If (HoursWkd > 40.0#) Then

setGrossPay((40 \* Wage) + ((HoursWkd - 40) \* Wage \* 1.5))

Else

setGrossPay(HoursWkd \* Wage)

End If

End Sub

Public Overridable Sub TellAboutMe()

'This routine is called to get the object to tell about

'itself...we will see this in play in our Is-a child object

'clsValuableCustomer

Debug.WriteLine(Me.Name & " is " & Me.Age & " and makes " & Me.Wage)

Debug.WriteLine(Me.Name & "'s address: " & Me.CustomerAddress.Address &

" - " &

Me.CustomerAddress.City & ", " &

Me.CustomerAddress.State & " " & Me.CustomerAddress.Zip)

End Sub

End Class

The Address class is simple, but shows another part of the language…

'Chapter 16 - Program 2

'I put this in its own class file, but it could have been placed

'in a single file along with the other classes. Which is better?

Public Class clsAddress

'Note: I am violating the tenets of OO by doing this next piece!

'Since I declared these as public properties and didn't specify

'the get/set stuff, VB automatically writes gets, sets and creates

'a class specific private backing variable for me called \_varname

'where varname is replaced with the actual public name

'Really cool, but not necessarily really OO and again we're sort

'of using magic and you saw what that bought us in databases!

Public Property Address As String

Public Property City As String

Public Property State As String

Public Property Zip As String

'Default Constructor

Public Sub New()

Address = ""

City = ""

State = ""

Zip = ""

End Sub

'Named Constructor

Public Sub New(ByVal strAddress As String, ByVal strCity As String,

ByVal strState As String, ByVal strZip As String)

Address = strAddress

City = strCity

State = strState

Zip = strZip

End Sub

End Class

Remember the clsCustomer object is responsible for creating and storing an instance of the clsAddress type – that’s Has-a inheritance. We now want to turn our view to the Is-a hierarchy with the code for our new inherited class clsValuableCustomer:

'Chapter 16 - Program 2

Public Class clsValuableCustomer

'This line says that clsValuableCustomer inherits

'everything that clsCustomer has (including the Has-a address -- cool!).

'We add some things that make this class unique

'from its parent class too. The Inherits keyword is our friend

'the Is-a inheritance at work.

Inherits clsCustomer

'Private data members we are adding to this class

Private sngCreditLine As Single

Private strBankName As String

'Default constructor

Public Sub New()

'Call parent's (clsCustomer) default constructor

MyBase.New()

'Set this class' unique members to defaults

'(See the section on properties below)

Me.CreditLine = 0.0

Me.BankName = ""

End Sub

'Named constructor

Public Sub New(ByVal newName As String, ByVal newAge As Integer,

ByVal newWage As Single, ByVal newAddress As String,

ByVal newCity As String, ByVal newState As String,

ByVal newZip As String, ByVal newCreditLine As Single,

ByVal newBankName As String)

'Call parent's named constructor

MyBase.New(newName, newAge, newWage, newAddress, newCity, newState,

newZip)

'Initialize this class' private members to parameter values

Me.CreditLine = newCreditLine

Me.BankName = newBankName

End Sub

Property CreditLine As Single

Get

Return sngCreditLine

End Get

Set(ByVal newCreditLine As Single)

sngCreditLine = newCreditLine

End Set

End Property

Property BankName As String

Get

Return strBankName

End Get

Set(ByVal newBankName As String)

strBankName = newBankName

End Set

End Property

'A destructor that doesn't really do anything useful

Protected Overrides Sub Finalize()

Debug.WriteLine("An instance of clsValuableCustomer is being destroyed")

Try

'Call parent (clsCustomer) finalizer

MyBase.Finalize()

Catch Ex As Exception

'we're leaving so don't worry about it

End Try

End Sub

'Let's override the parent's (clsCustomer) TellAboutMe method

'with a local version

Public Overrides Sub TellAboutMe()

Debug.WriteLine(Me.Name & " is a Valuable Customer " &

"with a credit line of " & Me.CreditLine)

'Now let's look at how to call the parent's (clsCustomer)

'TellAboutMe

MyBase.TellAboutMe()

End Sub

End Class

Finally, here's the code for the modMain module that drives our classes:

Module Module1

'Chapter 16 -- Program 2

Sub Main()

'The first customer is instantiated using the default constructor

Dim aCust As New clsCustomer()

'The second customer is instantiated using a named constructor

Dim anotherCust As New clsCustomer("Scott", 34, 15.0, "123 Elm",

"Saginaw", "MI", "48604")

'The third customer is a valued customer

Dim aValuedCust As New clsValuableCustomer("Moe", 20, 20.0,

"456 Dollar Lane",

"Saginaw", "MI", "48604",

75000.0, "BigBucksBank")

'Use some of the properties to put values into the private

'data members

aCust.HoursWkd = 10

anotherCust.HoursWkd = 10

aValuedCust.HoursWkd = 10

'Call a method in each of the classes

aCust.ComputeWage()

anotherCust.ComputeWage()

aValuedCust.ComputeWage()

'Show that the ComputeWage method actually did some work --

'remember that there is no way to publically set the GrossPay...

'We can only publically get the GrossPay through the property

Debug.WriteLine("aCust makes " & aCust.GrossPay)

Debug.WriteLine(" ")

Debug.WriteLine("anotherCust named " & anotherCust.Name &

" makes " & anotherCust.GrossPay)

Debug.WriteLine(" ")

'Use the new overriden method

anotherCust.TellAboutMe()

Debug.WriteLine(" ")

aValuedCust.TellAboutMe()

End Sub

End Module

When we execute the program, here’s what we get as program output:

aCust makes 0

anotherCust named Scott makes 150

Scott is 34 and makes 15

Scott's address: 123 Elm - Saginaw, MI 48604

Moe is a Valuable Customer with a credit line of 75000

Moe is 20 and makes 20

Moe's address: 456 Dollar Lane - Saginaw, MI 48604

An instance of clsValuableCustomer is being destroyed

An instance of clsCustomer is being destroyed

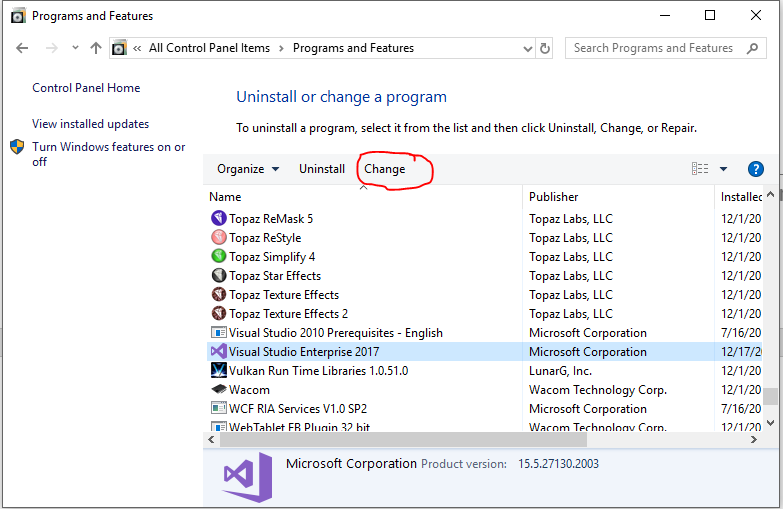
An instance of clsCustomer is being destroyed

An instance of clsCustomer is being destroyed

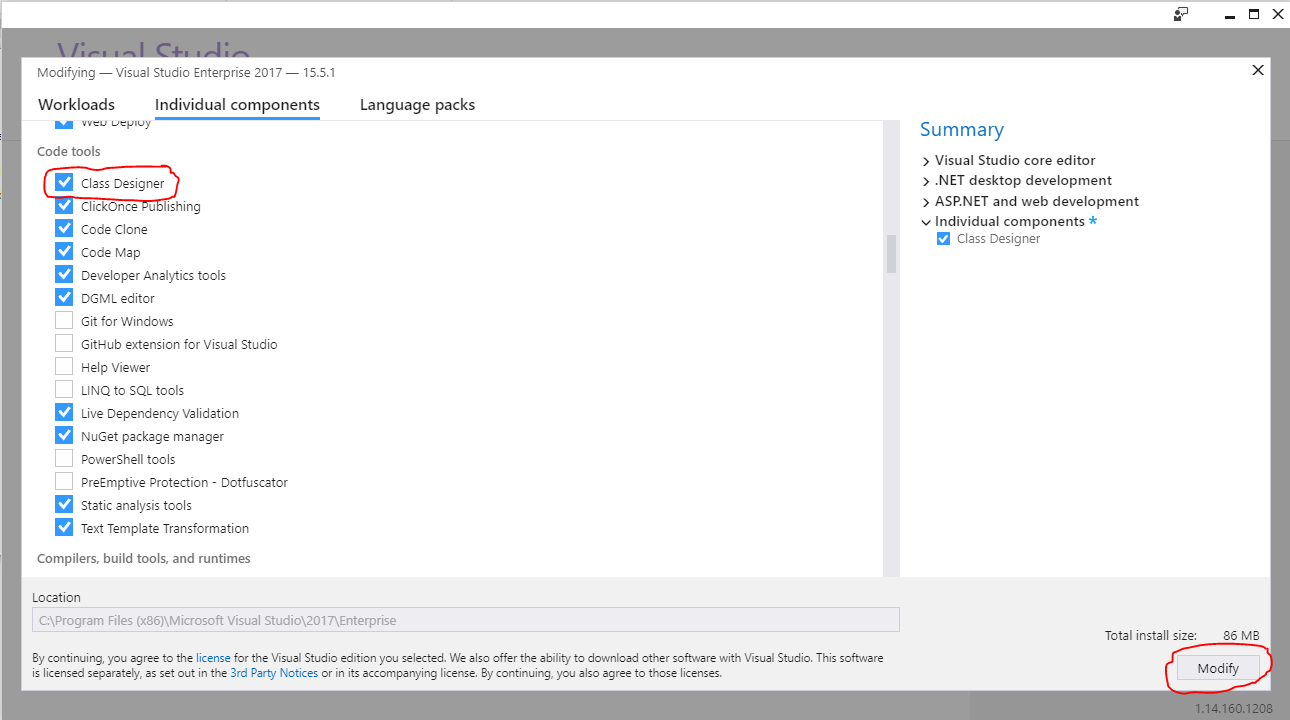
Why 3 instances of clsCustomer? Well, we actually had three if you’re counting correctly… Do you “see” all three of them?

So, this last program was a pretty good review of the basic tenets of object-orientation. Maybe you’ve got a headache after all that Is-a/Has-a mumbo-jumbo, so I’m going to have you take a couple minutes and do something else. We’re going to make an installation modification to our Visual Studio install. If you’re running Visual Studio, make sure you exit it before beginning this process.

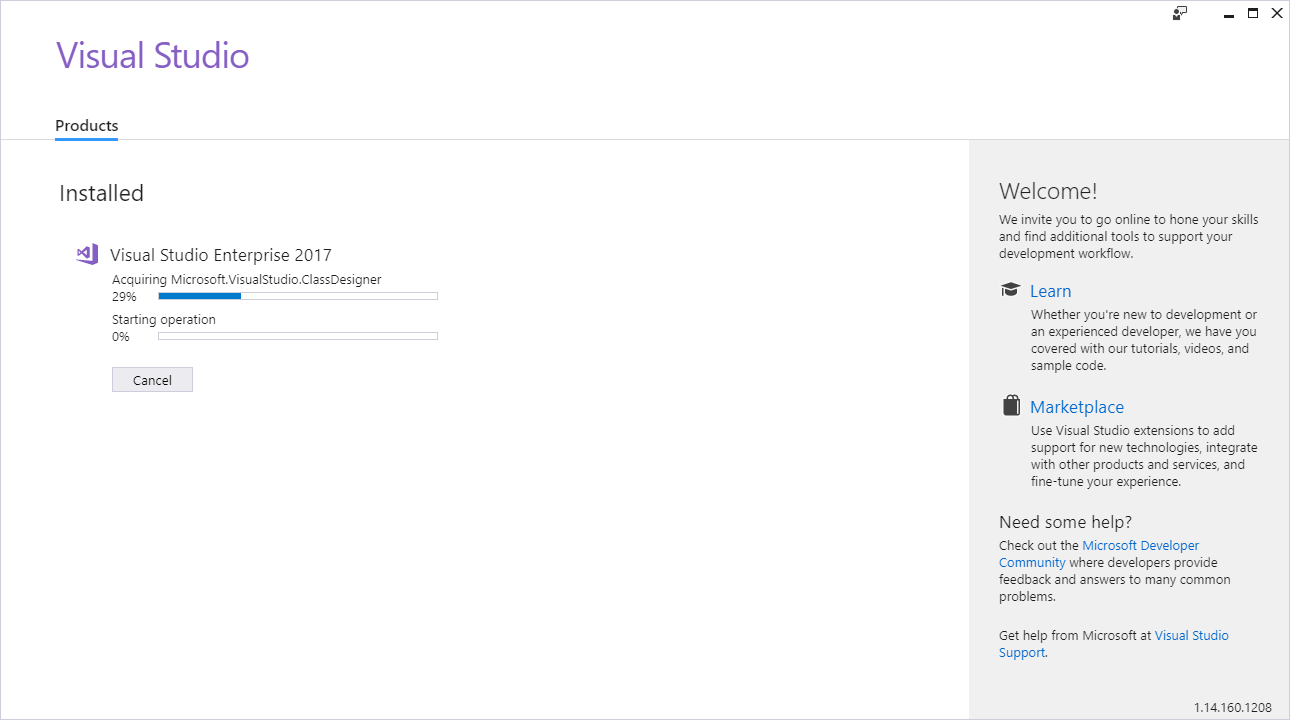
Once VS is shut down, begin the installation modification by going under the Control Panel🡪Program and Features. Scroll down until you find your Visual Studio Enterprise 2017 program in the list of installed programs. You did install things the way I suggested through the installation guide, right? Once you’ve clicked on the program name in the list of installed programs, click on the Change link toward the middle of the screen:



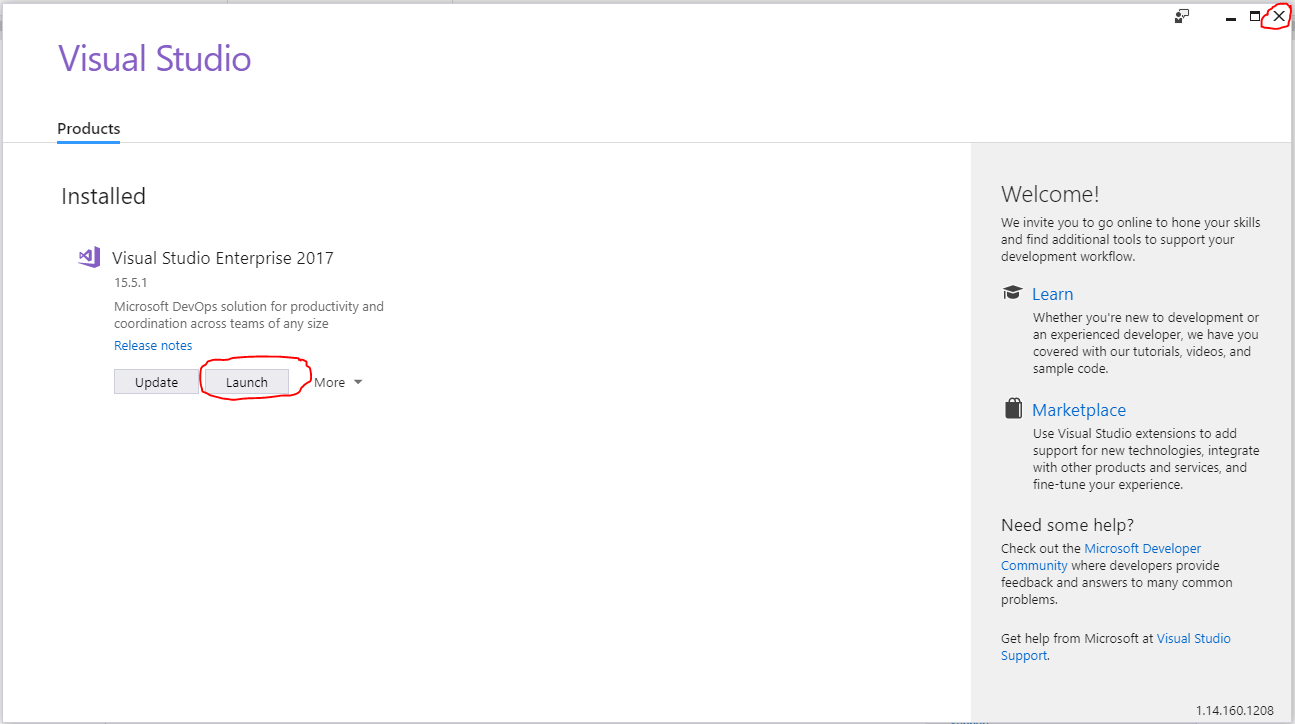
This will launch the Visual Studio installer again. It may want you to update the installer, and if that’s the case, go ahead and do so. I’ll wait for you…OK, glad you’re back! Once the installer opens, you’ll have three options at the top of the screen asking what you want to modify. Click on the Individual components item. You’ll then have a checkable list of items shown to you. Scroll down until you find the “Code tools” section and check the Class Designer checkbox. Once you’ve done that, click the Modify button at the bottom of the screen:



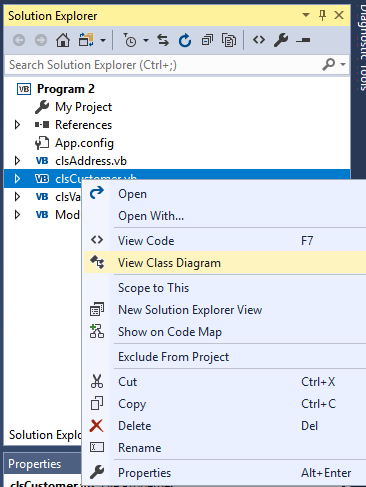
Sit back and let VS do its thing as it installs the software you just requested:



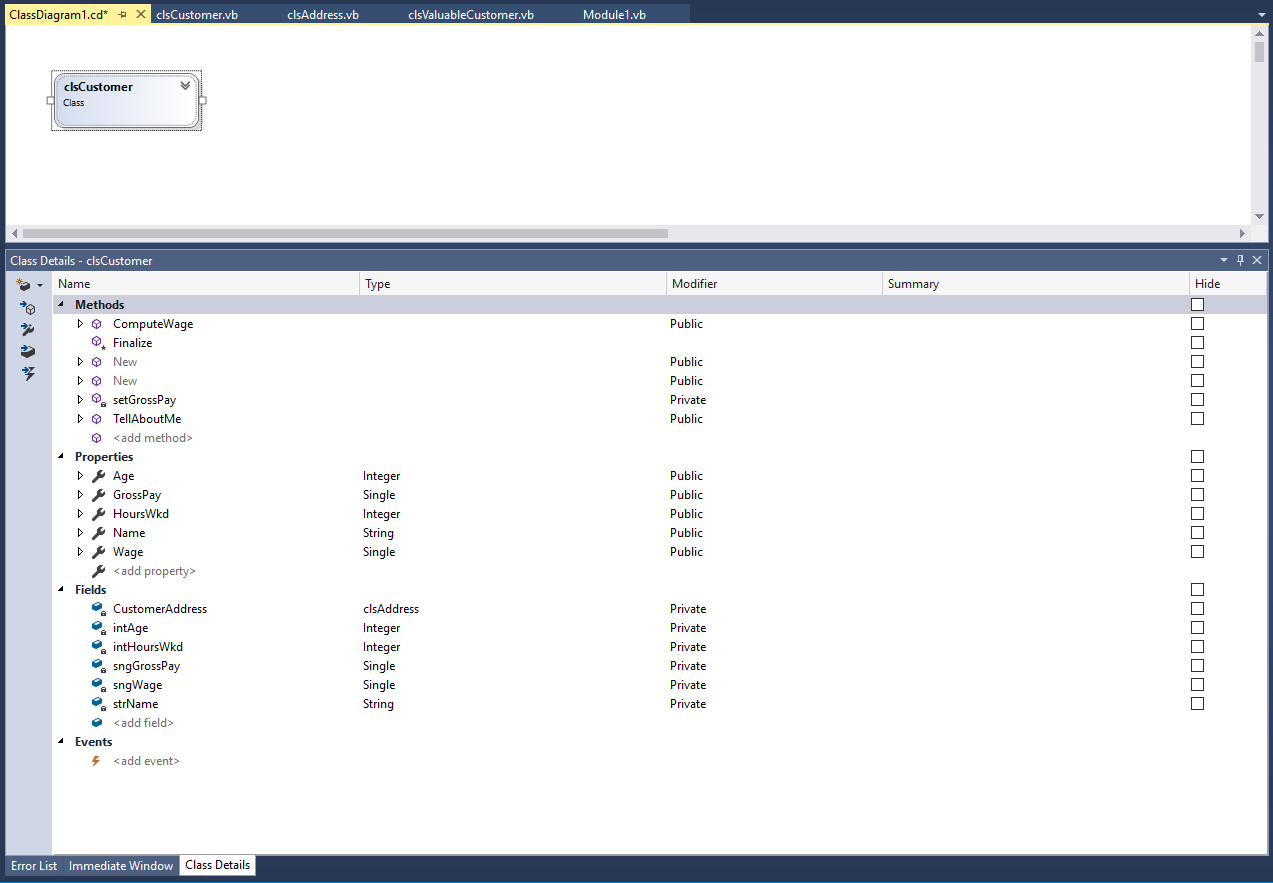
Eventually the installation will wrap up and you’ll be taken to the “Installed” screen. You can now either click Launch to run Visual Studio again from the installer or click the X in the upper right corner of the installer to exit the installer. In any case you’re going to need to load Visual Studio up again.



Now back to our code example in progress. Try this slick little trick: right click on clsCustomer in the Solution Explorer and select View Class Diagram. This option will only be available to you if you modified your installation as outlined above:

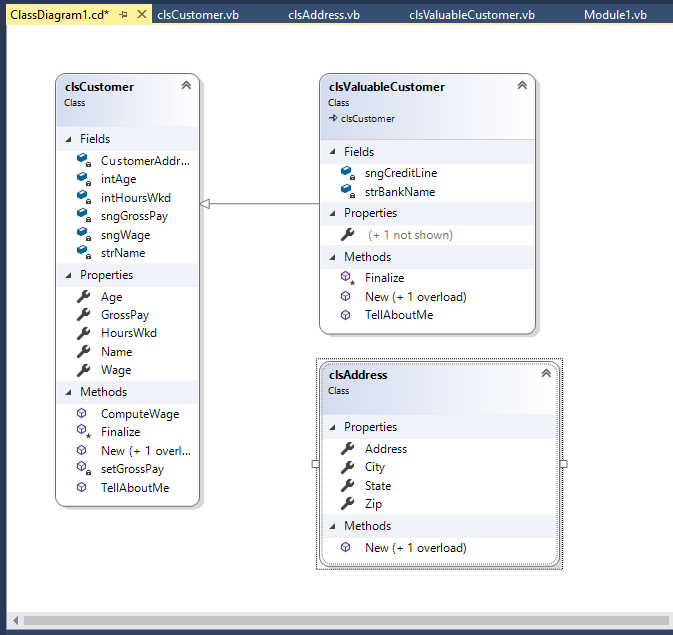


Here’s what we get:



Notice that a Class Details tab shows up in the same area where our Error List and Immediate Window are located. We can see all of the methods, properties, fields and events. We can even modify and add new ones right here.

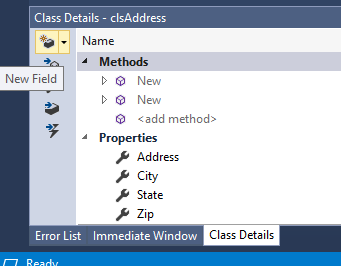
By itself, the diagram is not all that impressive (yet!), but left click and drag the other two classes you built from the Solution Explorer over to the Class Diagram Canvas and then click on the chevron at the top right of each class to expand them. With a little rearranging, you’ll get something that looks like this:



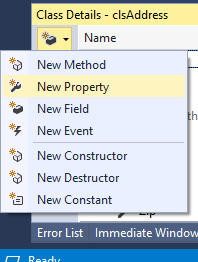
Now that’s pretty neat. Think about how handy this would be if you had a project with a whole bunch of class files that you weren’t familiar with. How do they relate to one another? Well, a picture is worth a thousand words. But that’s not where the Class Diagram power ends. Remember that lower section of the screen which shows the various variables, properties, methods and so forth? You can actually point and click and build up the skeletal pieces of your code here!

Obviously the Class Diagram can’t write the guts of a method, but it will write all of the header junk and let you start working by concentrating on the real programmatic content rather than the grunt work. Can you really say that you aren’t you tired of writing get/set accessors?

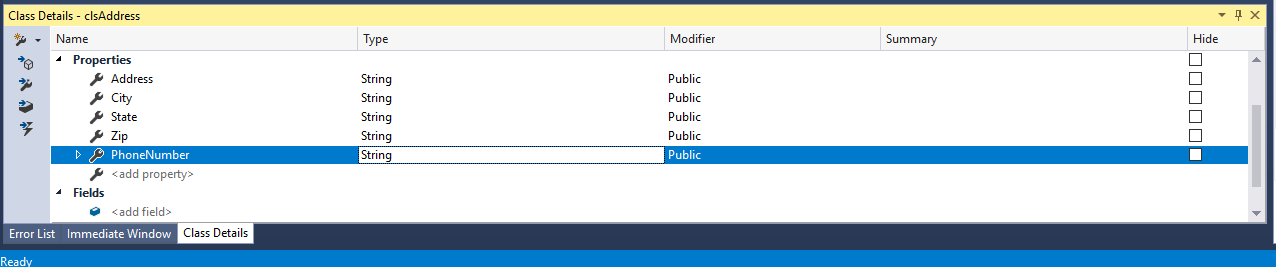
Let’s try this out. Left click on the Addresses object in the Class Diagram. Now, down in the Class Detail pane’s left corner (the lower pane), you’ll see a button for creating a New Field (it’s the top most button). Click the dropdown button next to the New Field icon:



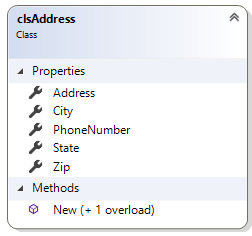
In the drop down that appears, pick New Property:



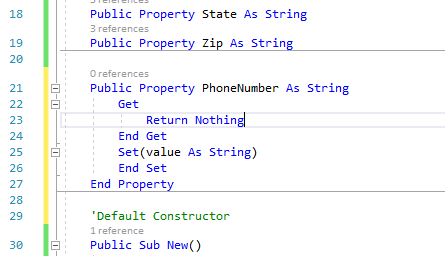
Finally, fill out the new line in the Properties entry form as follows:



Take a look at your class diagram and your new field is showing up:

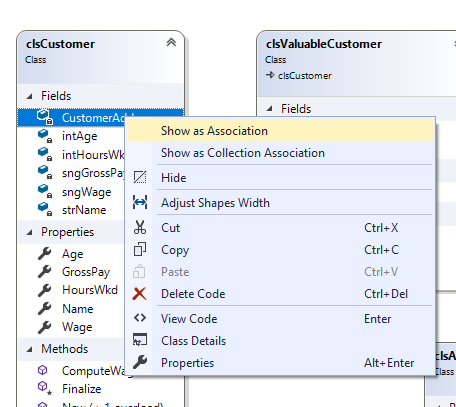


Better still, go back to your clsAddress code window:



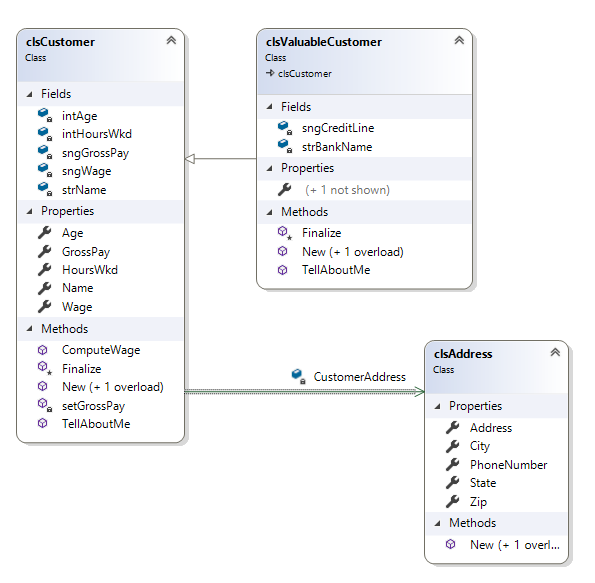
Notice that it wrote all the Property header and get/set stuff for me. Now it didn’t build the PhoneNumber Property quite right, but I warned you about that when we created the class using the simplified public Properties. Properties without the get/sets just isn’t “true” OOP. Bottom line, the diagram and your code are the same entity: you can drag and drop draw your classes and hook them together visually and code will be generated for you. Make changes to the code and the diagram reflects those changes. You’ll want to play around with this more, but at least you know it exists!

Also, if was bugging you that the Has-a relationship between Customer and Address didn’t show up on the diagram, we can remedy this. It’s not something that the Class Diagram will automatically do. Right click on the CustomerAddress field in the clsCustomer object in the Class Diagram:

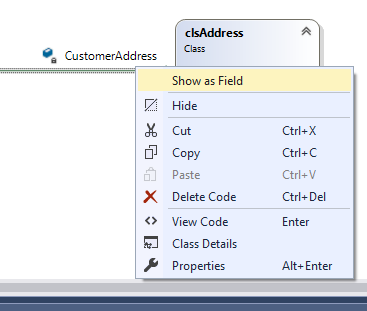


From the context menu, select Show as Association. I know that it’s not technically an Association by definition, but that’s the only kind of Has-a that the Class Diagram will render.

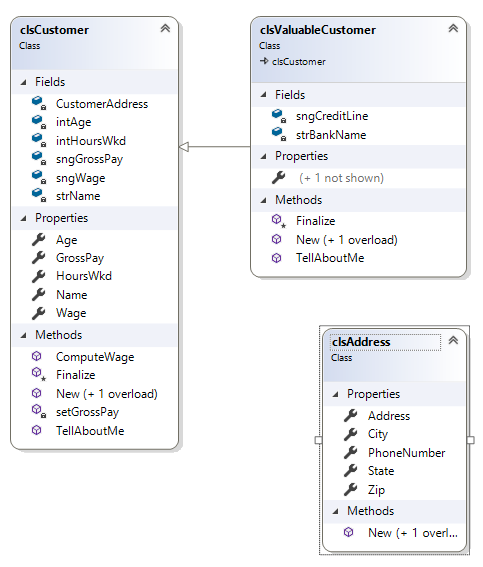
Here’s the modified diagram showing both inheritance types:



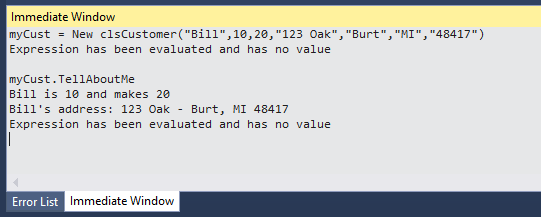
Notice that the CustomerAddress field disappears from the clsCustomer class. It now appears on the association link. We can toggle back and forth between viewing things as an association and as a field. It will not make any impact on the code in either the clsCustomer or clsAddress classes. If you want to change things back to the “field view” instead of seeing the association, just right-click on the association link and select Show as Field from the context menu that appears:



The association link will disappear and the field will reappear in the clsCustomers class:



One other bit of nougety goodness you should know about is some Immediate Window functionality. A lot of time, we might like to play with an instance of an object to make sure it works right without writing a whole bunch of code to drive it. So turn to the Immediate Window and try this out:



How sweet is that? I instantiated a new object on the fly and invoked its methods to make sure it works right! No more building a “driver” program just to test things out… About the only down side of doing this is that you have limited IntelliSense.

# A Polymorphic Example

Polymorphic is Greek meaning "many shapes." This is the ability for a method call to be placed in several classes. No matter what particular class gets called, it knows how to perform the method for itself.

Think about basic shapes. Can I ask you to draw a circle? Sure, you can do it. What about a square? No problem. How about a triangle? Again, there's no trouble drawing any of these shapes. No matter what the shape is you can draw it as long as you are told the shape type – that's a polymorphic method.

Let's look at an example of polymorphic behavior in VB code. Let's create a few classes for animals: a cat, a dog and a cow. Can each of those animals "speak?" Anthropomorphically we would say yes. A cat “speaks” when it makes its meow sound. A dog “speaks” when it barks. A cow “speaks” when it moos. You get the idea…

If we implement this idea of polymorphic behavior correctly, then anytime that I run across an instance of an animal, I can blindly ask it to speak without knowing what kind of an animal it is and it will carry out the method correctly for me. Pretty neat, eh?

Here’s the code for our polymorphic farm:

'Chapter 16 - Program 4

Module Module1

'Rather than carting around a bunch of separate files, I am

'going to declare all my classes in one file. This is not a

'good idea -- each class ought to have its own file as shown

'in the previous code examples. I am doing this simply

'for compactness reasons.

Public MustInherit Class clsAnimal

'You cannot create instances of clsAnimal -- it is an

'abstract class that is just used as a base to create

'child classes from

Public Sub New()

'Doesn't do anything special

End Sub

Public MustOverride Sub Speak()

'The generic clsAnimal can't speak

'but we will put an empty method in

'here and mark it as MustOverride --

'therefore, every child class that is

'derived from clsAnimal will have to

'implement a "local" Speak method

End Class

Public Class clsCat

'Our cat is a type of animal, so

'inherit everything that a clsAnimal has

Inherits clsAnimal

Public Sub New()

'New does nothing special here

'but if we needed the Cat constructor

'to do something specific for a cat, we

'could

End Sub

Public NotOverridable Overrides Sub Speak()

'The Speak method says any class that inherits from

'clsAnimal must implement it. That's the Overrides

'keyword. NotOverridable means that if any classes

'were to inherit from clsCat, they cannot implement

'a Speak method different than what Cat offers. So

'if you were to create a clsTiger and clsManx from

'clsCat, they would all only be able to say "Meow."

'That's the NotOverridable keyword.

Debug.WriteLine("Meow")

End Sub

Public Sub DrinkMilk()

'Cats drink milk which makes them unique from Dogs, Cows and Chickens

Debug.WriteLine("Sip Sip Sip")

End Sub

End Class

Public Class clsDog

'Our dog is a type of animal

Inherits clsAnimal

'Since we aren't doing anything with the constructor,

'we don't even need one. I just commented it out here so

'that you could see it, but I could just as easily delete it.

'Public Sub New()

'End Sub

Public NotOverridable Overrides Sub Speak()

'This is the Dog's "local" version of Speak

Debug.WriteLine("Bark")

End Sub

Public Sub EatsPaper()

'Dogs eat newspapers which makes them unique from Cats, Cows and

'Chickens

Debug.WriteLine("Shred Rip Tear")

End Sub

End Class

Public Class clsChicken

'Our chicken is a type of animal

Inherits clsAnimal

'Note no New method at all...

Public NotOverridable Overrides Sub Speak()

'Here's our "local" Speak method for Chickens

Debug.WriteLine("Cluck")

End Sub

Public Sub Picks()

'Chickens pick making them unique from Cats, Dogs and Cows

Debug.WriteLine("Scratch Pick Scratch")

End Sub

End Class

Public Class clsCow

'Our cow is a type of animal

Inherits clsAnimal

Public NotOverridable Overrides Sub Speak()

'The Cow's Speak method

Debug.WriteLine("Moo")

End Sub

Public Sub ChewsCud()

'Cows chew cud which makes them unique to Cats and Dogs

Debug.WriteLine("Chew Chew <Burp> Chew")

End Sub

End Class

Sub Main()

'Create some animals

Dim Bessie As New clsCow

Dim Rufus As New clsDog

Dim Kitty As New clsCat

Dim Sanders As New clsChicken

Debug.WriteLine("Time to play with the animals:")

'Let's play with our animals

Rufus.EatsPaper()

Rufus.Speak()

Bessie.Speak()

Kitty.Speak()

Kitty.DrinkMilk()

Sanders.Speak()

Sanders.Picks()

Debug.WriteLine("Play time is over")

Debug.WriteLine(" ")

'Create a farm for all the animals

Dim myFarm(3) As clsAnimal

'Put them in the farm

myFarm(0) = Kitty

myFarm(1) = Bessie

myFarm(2) = Sanders

myFarm(3) = Rufus

'Why can they all go in the Farm object? A clsCat Is-a clsAnimal

'So is a dog, chicken and cow. Since the array is of clsAnimal,

'all the derived classes can enter with no problem.

'Now iterate over all the critters for something really

'cool.

Debug.WriteLine("True Zoo Talk:") For Each critter In myFarm

critter.Speak()

Next

'Remember that the farm is just an array of clsAnimals, the

'base class, clsAnimal, had a Speak method in it (even though

'it didn't do anything and had to be overridden, it still

'had one), so we can generically call Speak and then

'whatever child type the critter is will call its

'appropriate Speak method -- polymorphic behavior!

End Sub

End Module

Here's the output from the program:

Time to play with the animals:

Shred Rip Tear

Bark

Moo

Meow

Sip Sip Sip

Cluck

Scratch Pick Scratch

Play time is over

True Zoo Talk:

Meow

Moo

Cluck

Bark

Polymorphic behavior is a very powerful and interesting facet of object-oriented programming that you should learn more about. When you start messing around with the concepts of inheritance and polymorphic activity, you really start expanding your problem-solving capabilities. The next stop on your education journey should be design patterns which require some in-depth knowledge of these topics.

# Summary on Basic Object-Orientation Keywords

We have covered all the basic pieces of object-orientation that I want to introduce you to in this chapter. Let's review the basic keywords that we used in creating the previous object-oriented application examples:

|  |  |
| --- | --- |
| *Keyword* | *Purpose* |
| Class | Used to declare a new class |
| Finalize | Used in the destruction of a class |
| Inherits | Used to indicate that a new class is being derived (inherited) from a parent class |
| MustInherit | Creates an abstract class that cannot be instantiated and from which child classes must be created |
| MustOverride | Forces a method to be overridden (can only be used in an abstract class) |
| MyBase | Used to reference a class' parent class |
| New | Used to create a new instantiation of a particular class |
| NotOverridable | Indicates that a child class cannot override a particular method in its parent class |
| Overridable | Used in a parent class to indicate that a method in the parent class can be overridden by an inherited child class |
| Overrides | Indicates that an inherited class is overriding a method from its parent |
| Private | Indicates a property/data member/method is only available to your class |
| Property | An alternative to accessors, which is the way that most VB controls work |
| Protected | Indicates a property/data member/method is only available within the class and classes that inherit from the class |
| Public | Indicates a property/data member/method is available to everything in your program |
| ReadOnly | Makes a property read only |
| WriteOnly | Makes a property write only |

So, while we did cover a lot of territory, we did not touch on all the OO keywords in VB's vocabulary. These are some additional topics that we want to explore:

|  |  |
| --- | --- |
| *Keyword* | *Purpose* |
| MyClass | Can be used to force a specific method invocation within a class |
| NotInheritable | Prevents a class from being inherited |
| Partial | Allows a class to be broken up and defined across several code files |
| Shadows | Redeclares an inherited method in a child class |
| Shared | Shared members can be used without the instantiation of a class; also, if used on an attribute, all instances share a single copy of the attribute |

Here’s an example program showcasing many of these other concepts:

'Chapter 16 - Program 5

NotInheritable Class MyNotes

'Nobody can try to make new classes off of this class

'This marks this class as the end of an inheritance chain

'

'They can use it as it is, but they cannot subclass from it.

End Class

Class clsOverloadingExample

'This example class shows how we can have overloaded Methods each with

'a different signature...all three methods are called MySquare and the

'correct version will be called at runtime based on the signature of

'the parameter type that is passed through

Public Overloads Function MySquare(ByVal value As Double) As Double

Return CDbl(value \* value)

End Function

Public Overloads Function MySquare(ByVal value As Integer) As Integer

Return CInt(value \* value)

End Function

Public Overloads Function MySquare(ByVal value As String) As String

If Val(value) Then

Return CStr(CSng(value) \* CSng(value))

Else

Return "Not Calculable"

End If

End Function

End Class

Class clsTime

'This example illustrates the Shared keyword. Nobody has to

'create an instance this class to use the shared method GetTheTime

Public Shared Function GetTheTime() As String

Return (Now().ToString("hh:mm"))

End Function

End Class

Class clsLanguage

'This is a simple class that can be inherited

Public Sub Greeting()

'Here's how we say hello by default

Debug.WriteLine("Hello")

End Sub

End Class

Class clsEnglish

'Now derive clsEnglish from clsLanguage

Inherits clsLanguage

'We don't need to do anything about the Greeting

'since the parent version will be called and it

'was already in English

End Class

Class clsSpanish

Inherits clsLanguage

'We don't want to call the parent Greeting version

'so we need to put in our own local version here, but

'since the parent didn't make us override it, just saying

'Greeting again would produce an error -- use the Shadows

'keyword and you are saying that the local class' method

'will be called if the Greeting method fires

Public Shadows Sub Greeting()

Debug.WriteLine("Hola")

End Sub

End Class

Partial Class clsWorker

'We can break up a class so that the definition for it takes

'place in mutliple locations. Why? What about if we want

'to lock down some of the source code, but still want people

'to be able to extend the base without inheriting from it...

'Isn't that really what forms do for you? VB writes the

'GUI for you from the drag and drop builder, yet you have a

'code behind file where you can add event handlers.

Property FirstName As String

Property LastName As String

Public Sub PrintName()

Debug.WriteLine("Name: " & FirstName & " " & LastName)

End Sub

End Class

Partial Class clsWorker

'Here's another chunk of clsWorker. Normally we would put these

'in separate files, but you get the idea.

'In this chunk, let's add some more properties...

Property Address As String

Property City As String

Property PhoneNumber As String

'and methods to what was already there

Public Sub PrintDetails()

MyClass.PrintName()

Debug.WriteLine("Lives at: " & Address & " " & City)

Debug.WriteLine("Phone: " & PhoneNumber)

End Sub

End Class

Module Module1

'Let's test stuff out

Sub Main()

Dim myEnglish As New clsEnglish

Dim mySpanish As New clsSpanish

Dim myOverloadedClass As New clsOverloadingExample

Dim myEmployee As New clsWorker

Debug.WriteLine("Examples of the Shadows keyword where we had " &

"different versions of the Greeting method:")

myEnglish.Greeting()

mySpanish.Greeting()

Debug.WriteLine(vbCrLf & "Using the Shared method where we don't " &

"even have to instantiate the class:")

Debug.WriteLine("The current time is " & clsTime.GetTheTime)

Debug.WriteLine(vbCrLf & "Calling an overloaded method with different " &

"signatures:")

Debug.WriteLine(myOverloadedClass.MySquare(5))

Debug.WriteLine(myOverloadedClass.MySquare(10.2))

Debug.WriteLine(myOverloadedClass.MySquare("3"))

Debug.WriteLine(myOverloadedClass.MySquare("Four"))

Debug.WriteLine(vbCrLf & "Working with properties and methods that " &

"were scattered across two partial class definitions:")

With myEmployee

.FirstName = "Bill"

.LastName = "Smith"

.Address = "123 Elm"

.City = "Burt"

.PhoneNumber = "555-1449"

End With

myEmployee.PrintName()

myEmployee.PrintDetails()

End Sub

End Module

We get the following output when we run the application:

Examples of the Shadows keyword where we had different versions of the Greeting method:

Hello

Hola

Using the Shared method where we don't even have to instantiate the class:

The current time is 11:48

Calling an overloaded method with different signatures:

25

104.04

9

Not Calculable

Working with properties and methods that were scattered across two partial class definitions:

Name: Bill Smith

Name: Bill Smith

Lives at: 123 Elm Burt

Phone: 555-1449

You should have a pretty good idea of how mixing class design with concepts like generics can start to build up powerful programs without a whole lot of coding. That’s the big payoff of object-orientation, but it takes practice and time to learn to “see” things this way. Once you start mastering these pieces, you are then ready to start moving up and thinking about things beyond the application layer and that is architectural design.

# Firing Events and Exceptions from your own Classes

So far, we have explored how to add methods, attributes and properties to a class. There is just one more critical piece remaining – how to add events to classes. After all, the forms and controls you’ve worked with are classes and they have events that we respond to with our event handlers. VB provides a statement called RaiseEvent that is used to trigger, or fire, an event.

Events are declared in a class using the keyword Event. You must remember to declare the event Public. Events can also send and receive parameters just like normal subroutines and functions.

There is one additional change that must be made to the user’s main form or program in order to allow the user’s class to raise events. When the class is declared, the instance of the class must include the keyword WithEvents to permit events to fire. Those of you with sharp eyes may remember that when we looked at the code generated by the GUI builder all of the controls were declared WithEvents there!

Here’s an example program that illustrates how to raise events. I created a class called clsMyClass. Inside the class it can raise two events: DoBeep (which is supposed to make the speaker beep) and DisplayMsg (which will display a text message). All that we declare inside this class is the various events that it can raise. The implementation of what to do when those events are raised take place elsewhere.

If you stop and ponder this idea for a minute, it shouldn't seem that strange. The Command button class has the ability to raise the event "Click" which occurs when the user clicks on the command button. We don't write the code to be executed when the Click event occurs in the Command button itself, instead we write it as an event handler in the Form where the Command button is placed (cmdButton\_Click).

So, in clsMyClass you will see the two events that the class can raise and two methods that can be called to get the class to raise the events. We’re not ready to consider how to write controls yet, so we’ve got to kind of fake the triggering of the events at this point. The form that follows the class has the code for the event handlers that are to be executed whenever the DoBeep or DisplayMsg events are raised. Here's the code for clsMyClass:

'Chapter 16 - Program 6

Public Class clsMyClass

'The class defines the events that the class can

'raise -- the implementation of how to respond to the events

'has to be handled in the user's program -- which is exactly

'what most control's do -- think about clicking a button --

'all you do is get an event raised -- you have to implement

'a handler, i.e Sub cmdxxx\_Click() in your form

'Name the events and their parameters that this class can raise

Public Event DisplayMsg(ByVal s As String)

Public Event DoBeep(ByVal n As Integer)

'I wrote this routine to force the DisplayMsg event to fire.

'This can be called anytime and all it does is raise the

'DisplayMsg event -- the handler for the event will be in the

'application's form

Public Sub RaiseEvent1()

'This next line actually throws the event

RaiseEvent DisplayMsg("This was the DisplayMsg event")

End Sub

'I wrote this routine to force the DoBeep event to fire.

'Again, this can be called simply to get the class to raise the

'DoBeep event, which is handled in the application's form

Public Sub RaiseEvent2()

RaiseEvent DoBeep(3)

End Sub

'Don't worry! Eventually we will look at how to build controls

'and then you will actually have to write code in the control class

'that will raise events when things arise. Enjoy the temporary

'simplicity of things!

End Class

Here's the code that sits behind the application's form. The form simply has one command button on it called Button1 which will force both events in the class to be raised:

'Chapter 16 - Program 6

Imports System.ComponentModel

Public Class Form1

'Declare an instance of our class and notify our code that

'the class will be raising events. If you don't use WithEvents,

'VB will not allow any events to be raised, e.g. no

'events will be sent to the form for processing

Dim WithEvents TestMyClass As New clsMyClass

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'This will force a DisplayMsg event to be raised

TestMyClass.RaiseEvent1()

'This will force a DoBeep event to be raised

TestMyClass.RaiseEvent2()

End Sub

Private Sub TestMyClass\_DisplayMsg(s As String) Handles TestMyClass.DisplayMsg

MessageBox.Show(s)

End Sub

Private Sub TestMyClass\_DoBeep(n As Integer) Handles TestMyClass.DoBeep

Dim intLoop As Integer

For intLoop = 1 To n

Beep() '--should make the speaker beep

Next

End Sub

Private Sub Form1\_Closing(sender As Object, e As CancelEventArgs)

Handles Me.Closing

'Let's be a good programming and clean up any object instances

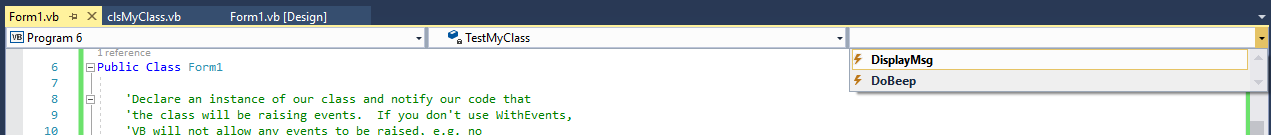
'we created before leaving the application

TestMyClass = Nothing

End Sub

End Class

Remember earlier we noted that the IntelliSense syntax autocompletion worked for classes that you develop. VB will also help you to write the event handlers for your classes' own events. The next screenshot shows the events that are available in the TestMyClass instance of clsMyClass from the dropdown chevrons above the code window:



Generating events really isn’t that much of a trick, and we can tap into them using the same old event model that we’ve been using since writing VB programs. Besides raising events, we can also get our classes to throw exceptions. We’ll write a simple class that handles the inventory level for some items that a retail store carries. If someone tries to take out more items than is in stock, we need to raise an exception. Likewise, if sometime tries to put in a negative inventory level value, or more than 999 units, when replenishing stock, we should flag that. We’ll throw ArgumentExceptions as a response. Here’s the source code for this program:

'Chapter 16 - Program 7

Module Module1

Class clsInventoryItem

Private intInventoryLevel As Integer

Private strName As String

Public Sub New(ByVal ItemName As String)

Name = ItemName

intInventoryLevel = 0

End Sub

Public Property Name As String

Get

Return strName

End Get

Set(ByVal value As String)

strName = value

End Set

End Property

Public ReadOnly Property InventoryLevel As Integer

Get

Return intInventoryLevel

End Get

End Property

Public Sub AddToInventory(ByVal Units As Integer)

If Units <= 0 Or Units >= 1000 Then

'If someone tries to put in a negative value or

'1000 or more units, our business rules says that's

'an invalid unit amount

Throw New ArgumentException("Exceeded Unit Range")

Else

intInventoryLevel += Units

End If

End Sub

Public Sub RemoveInventory(ByVal units As Integer)

If units > intInventoryLevel Then

'Another business rule says you can't take out more

'inventory than you have -- doh!

Throw New ArgumentException("Tried to remove units beyond " &

"current inventory available")

Else

intInventoryLevel -= units

End If

End Sub

Public Sub ShowInventory()

Debug.WriteLine("Item : {0} currently has {1} unit(s) in stock", Name,

InventoryLevel)

End Sub

End Class

Sub Main()

Dim myInventory1 As New clsInventoryItem("Computers")

Dim myInventory2 As New clsInventoryItem("Desks")

'Populate our inventory levels to their initial values:

myInventory1.AddToInventory(5)

myInventory1.ShowInventory()

myInventory2.AddToInventory(1)

myInventory2.ShowInventory()

'Let's do somethings that might be invalid:

Try

'Let's try to put in an insane inventory level amount

myInventory1.AddToInventory(10000)

Catch ex As ArgumentException

Debug.WriteLine(ex.Message)

End Try

Try

'Let's try to take out more inventory than we have

myInventory2.RemoveInventory(20)

Catch ex As ArgumentException

Debug.WriteLine(ex.Message)

End Try

End Sub

End Module

Here’s the output from the program:

Item : Computers currently has 5 unit(s) in stock

Item : Desks currently has 1 unit(s) in stock

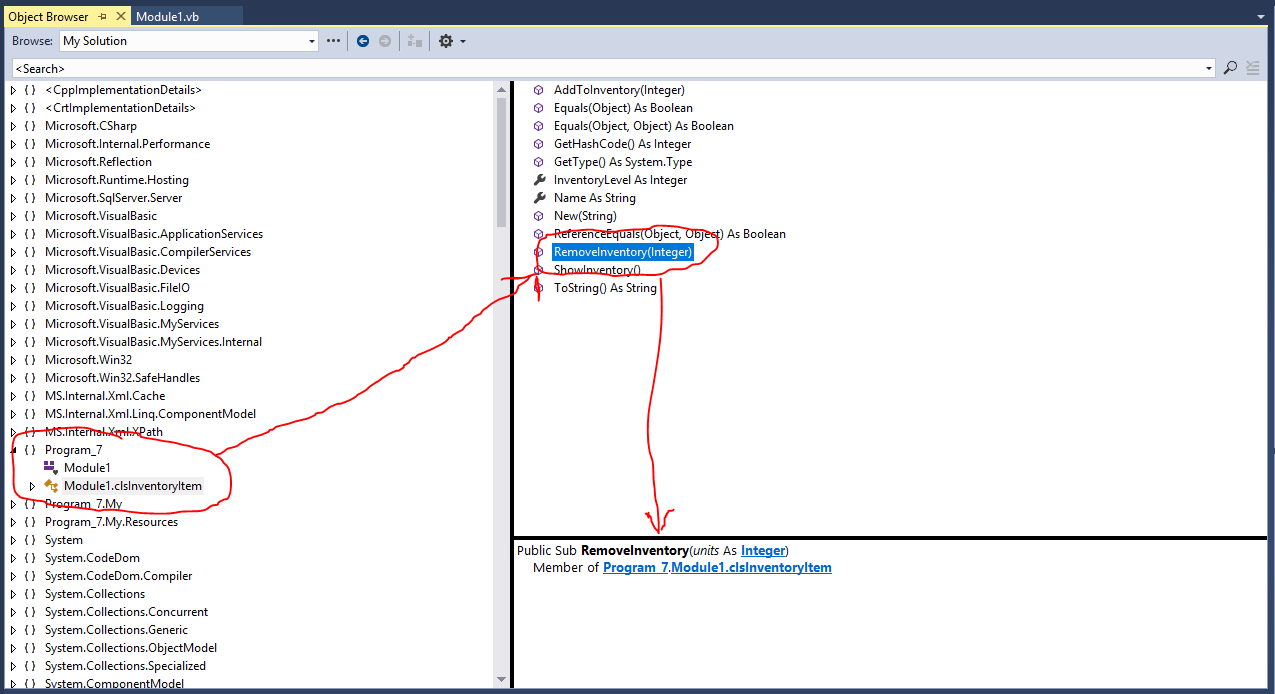
Exception thrown: 'System.ArgumentException' in Program 7.exe

Exceeded Unit Range

Exception thrown: 'System.ArgumentException' in Program 7.exe

Tried to remove units beyond current inventory available

A new tool that I would like to introduce in this chapter is the Object Browser, which you can find under the View🡪Object Browser menu. This utility will show every class that’s available in .NET, both those built in and your own. Take a look at the screenshot:



This gives you a quick way to be able to see what methods, properties and events are available in an entity without having to comb through the MSDN help. Sometimes a quick peek is all that you really need.

# Understanding Interfaces

We have already mentioned that VB doesn’t support multiple inheritance. What it does support is Interface inheritance, which is the ability for a class to implement the public interface of another class but not the other class' implementation. This is much safer than multiple inheritance, and in reality, it is actually kind of a sneaky form of it since you can inherit from one class and implement multiple interfaces from other classes!

Any class can serve as an interface to another class. The public interface of the class serving as the Interface usually won’t contain any code; it will just hold the various public names of methods that need to be re-implemented in other classes that inherit the interface. The keyword Implements is used to implement the Interface in another class.

You may be wondering what the big deal is with interfaces. In fact, you may wonder why bother with them at all? The truth of the matter is that interfaces are very powerful abstractions. Think about the interface on a cassette player, DVD player, VCR or CD player. What does the button with the square symbol do? It stops the device from playing. What about the single arrow button? It tells the device to begin playing. The double arrow buttons are the move forward and rewind buttons. That's the purpose of an interface. In fact, many software applications that play different forms of media use exactly the same interface buttons.

If we are going to create any new device and we want people to be able to easily play, stop, rewind and fast forward, we will probably choose the standard interface that's available on just about every electronic device that performs those functions today. Notice that the interface is abstracted from the actual implementation. The arrow button always means "play", however that function is physically implemented very differently on a videocassette tape versus a DVD disk.

Let’s take a look at an interface example in VB:

'Chapter 16 - Program 8

Public Interface IShape

'This is the interface that will be implemented through

'other classes -- all we do is list the name of the methods

'and their parameter should there any. Any class that chooses

'to implement this interface has to implement these methods!

Function Area() As Single

Function Name() As String

'Notice that this is not a class -- instead it is an interface, that's

'why there can't be any functionality in here -- this is an abstraction

'and any other class that decides that it wants to implement this

'interface has to provide an Area and a Name method...

End Interface

Now let’s create a class that will implement that interface: clsPoint

'Chapter 16 - Program 8

Public Class clsPoint 'A point at some (X,Y) location

'We are going to implement the interface described above

Implements IShape

'Data members for this class

Private mX As Integer

Private mY As Integer

'Here’s the implementation for the public interface

'We have to specify if child classes can override these --

'also notice that we are explicitly stating what method this

'implements

Public Overridable Function Area() As Single Implements IShape.Area

Return (0)

End Function

'You will not be able to compile until you implement every method

'in an interface -- that's an important note!

Public Overridable Function Name() As String Implements IShape.Name

Return ("Point")

End Function

'Default constructor

Public Sub New()

X = 0

Y = 0

End Sub

'Named constructor

Public Sub New(ByVal newX As Integer, ByVal newY As Integer)

X = newX

Y = newY

End Sub

'Here are the public properties for this class

Property X()

Get

Return (mX)

End Get

Set(ByVal xValue)

mX = xValue

End Set

End Property

Property Y()

Get

Return (mY)

End Get

Set(ByVal yValue)

mY = yValue

End Set

End Property

End Class

We’ll further derive our inheritance by creating clsCircle, which is a clsPoint with a Radius:

'Chapter 16 - Program 8

Public Class clsCircle 'Circle is based on Point (a circle is a point

Inherits clsPoint 'with a radius)

'We don't need to implement IShape here since our parent does,

'but we may need to override some of the interface methods that

'our parent implemented...

'Data members for this class

Private mRadius As Integer

Sub New()

MyBase.New()

Radius = 0

End Sub

Sub New(ByVal newX As Integer, ByVal newY As Integer,

ByVal newRadius As Integer)

MyBase.New(newX, newY)

Radius = newRadius

End Sub

Property Radius()

Get

Return (mRadius)

End Get

Set(ByVal radiusValue)

mRadius = radiusValue

End Set

End Property

'We are overriding the interface implementation from the parent

'class since we need to calculate Area differently for a Circle

'than we do for a Point

Public Overrides Function Area() As Single

Return (3.1415 \* (Radius \* Radius))

End Function

Public Overrides Function Name() As String

Return ("Circle")

End Function

End Class

Finally, here’s the main program:

'Chapter 16 - Program 8

Module Module1

Sub Main()

'Instances of our classes

Dim myPoint As New clsPoint(1, 3)

Dim myCircle As New clsCircle(0, 0, 2)

Dim myColl As New Collection()

Dim anObject As Object

Debug.WriteLine("Direct Reference to each object and its area:")

'Here's a direct reference to each object

Debug.WriteLine(myPoint.Name & " - " & myPoint.Area)

Debug.WriteLine(myCircle.Name & " - " & myCircle.Area)

'Throw the shapes in a collection

myColl.Add(myPoint)

myColl.Add(myCircle)

Debug.WriteLine(vbCrLf & "Iterating through a container of Objects:")

'Iterate through the collection

For Each anObject In myColl

'You must put the derived class first when trying to detect the

'original class type! If you the base class, .NET for some

'reason guesses wrong. So much for all the type safety stuff

'it is supposed to provide. My hunch is that you start with

'as specific a type as possible and then slowly generalize

'back up the inheritance tree. After all, we know at the

'very top (the root of the tree), Object is the granddaddy

'of every other type. With our discussion on boxing, we

'know we can always get Object to hold any more specific

'type if we need to...

If TypeOf anObject Is clsCircle Then

Debug.WriteLine("Circle centered at (" & anObject.X &

"," & anObject.Y & ")" & vbCrLf &

"Radius is " & anObject.Radius &

" Area is " & anObject.Area())

ElseIf TypeOf anObject Is clsPoint Then

Debug.WriteLine("Point located at (" & anObject.X &

"," & anObject.Y & ")")

End If

Next

End Sub

End Module

The output is pretty straightforward:

Direct Reference to each object and its area:

Point - 0

Circle - 12.566

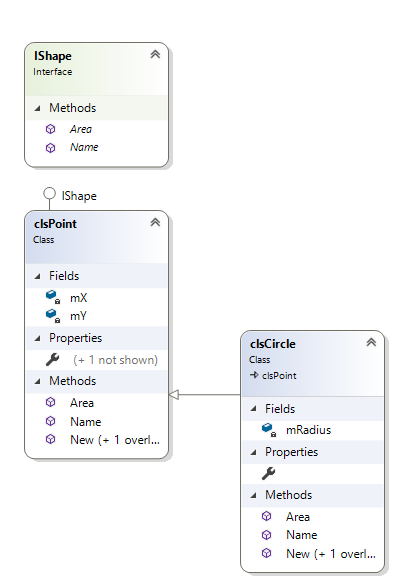
Iterating through a container of Objects:

Point located at (1,3)

Circle centered at (0,0)

Radius is 2 Area is 12.566

By the way, here’s where the Class Designer really starts to shine. If you were having a problem visualizing exactly how the pieces were all fitting together, the Designer can help:



Notice that Interfaces show up as green colored items, while classes appear in blue.

What does all of this Interface stuff really do for us? Well, a couple of things actually... First of all, we talked about how we can implement multiple interfaces to get around the no-multiple inheritance issue. That’s pretty significant if a class can’t have multiple parents. The second thing is that VB has a bunch of interfaces that we can implement in our classes to add in extra functionality. We’ve already learned about one of them: IDisposable – this was the interface that any class must implement if the Using keyword is to work upon it. Let’s look at implementing this interface in one of our classes:

'Chapter 16 – Program 9

Module Module1

Class clsEmployee

Implements IDisposable

'This is a .NET interface -- there are lots of them in the

'framework. Once we implement this one, our clsEmployee

'object can be used in a Using statement.

'Here's our class Properties, again without the accessors

Public Property Name As String

Public Property Hours As Integer

Public Property Wage As Single

'We need this for the implementation of the interface

Private disposed As Boolean

Public Sub ShowWage()

Debug.WriteLine("{0} earns {1}", Name, (Hours \* Wage))

End Sub

Public Overloads Sub Dispose() Implements IDisposable.Dispose

Dispose(True)

' This object will be cleaned up by the Dispose method.

' Therefore, you should call GC.SupressFinalize to

' take this object off the finalization queue

' and prevent finalization code for this object

' from executing a second time.

GC.SuppressFinalize(Me)

End Sub

Protected Overridable Overloads Sub Dispose(ByVal disposing As Boolean)

' Dispose(disposing As Boolean) executes in two distinct scenarios.

' If disposing equals True, the method has been called directly

' or indirectly by a user's code. Managed and unmanaged resources

' can be disposed of.

' If disposing equals False, the method has been called by the

' runtime from inside the finalizer and you should not reference

' other objects. Only unmanaged resources can be disposed of.

' Check to see if Dispose has already been called.

If Not Me.disposed Then

' If disposing equals True, dispose of all managed

' and unmanaged resources.

' Call the appropriate methods to clean up

' unmanaged resources here.

' If disposing is False,

' only the following code is executed.

' Note disposing has been done.

disposed = True

End If

End Sub

Protected Overrides Sub Finalize()

' This finalizer will run only if the Dispose method

' does not get called.

' It gives your base class the opportunity to finalize.

' Do not provide finalize methods in types derived from this class.

' Do not re-create Dispose clean-up code here.

' Calling Dispose(False) is optimal in terms of

' readability and maintainability.

Dispose(False)

MyBase.Finalize()

End Sub

End Class

Sub Main()

'This will allocate a new employee and

'I am setting the initial properties up in a Using

'block. Also notice the With keyword to instantiate

'the various class properties.

Using newEmp As New clsEmployee With

{.Name = "Bill", .Hours = 40, .Wage = 10.0}

'I can also use a With block inside of the Using block

With newEmp

.ShowWage()

End With

'As soon as we hit the End Using statement, the resource

'is released because it implemented IDisposable!

End Using

End Sub

End Module

In case you’re wondering about what other interfaces are available, here’s a list of some of the more commonly used:

|  |  |
| --- | --- |
| *Interface* | *Description* |
| [IAsyncResult](http://msdn.microsoft.com/en-us/library/system.iasyncresult.aspx) | Represents the status of an asynchronous operation. |
| [ICloneable](http://msdn.microsoft.com/en-us/library/system.icloneable.aspx) | Supports cloning, which creates a new instance of a class with the same value as an existing instance. |
| [IComparable](http://msdn.microsoft.com/en-us/library/system.icomparable.aspx) | Defines a generalized type-specific comparison method that a value type or class implements to order or sort its instances. |
| [IConvertible](http://msdn.microsoft.com/en-us/library/system.iconvertible.aspx) | Defines methods that convert the value of the implementing reference or value type to a common language runtime type that has an equivalent value. |
| [ICustomFormatter](http://msdn.microsoft.com/en-us/library/system.icustomformatter.aspx) | Defines a method that supports custom formatting of the value of an object. |
| [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx) | Defines a method to release allocated resources. |
| IEnumerable | Allows us to “walk” over a collection of objects and perform operations upon them. |
| [IEquatable(Of T)](http://msdn.microsoft.com/en-us/library/ms131187.aspx) | Defines a generalized method that a value type or class implements to create a type-specific method for determining equality of instances. |
| [IFormatProvider](http://msdn.microsoft.com/en-us/library/system.iformatprovider.aspx) | Provides a mechanism for retrieving an object to control formatting. |
| [IFormattable](http://msdn.microsoft.com/en-us/library/system.iformattable.aspx) | Provides functionality to format the value of an object into a string representation. |
| [IObservable(Of T)](http://msdn.microsoft.com/en-us/library/dd990377.aspx) | Defines a provider for push-based notifications. (If you are saying this looks like the Observer Design Pattern, you are right!) |
| [IObserver(Of T)](http://msdn.microsoft.com/en-us/library/dd783449.aspx) | Provides a mechanism for receiving push-based notifications. |
| [IProgress(Of T)](http://msdn.microsoft.com/en-us/library/hh138298.aspx) | Defines a provider for progress updates. |
| ISerializable | Used to convert an object into a binary format for serialized persistence. |

There’s no question that IEnumerable is an important interface. It’s what’s implemented behind many of the built-in data structures that are in .NET. Without it, we wouldn’t be able to use the For Each syntax to walk over the items in a data container or LINQ. Here’s a list of some the IEnumerable methods. Notice that many predicates are listed here:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [Aggregates](http://msdn.microsoft.com/en-us/library/bb548651.aspx) | Applies an accumulator function over a sequence. |
| [All](http://msdn.microsoft.com/en-us/library/bb548541.aspx) | Determines whether all elements of a sequence satisfy a condition. |
| [Any](http://msdn.microsoft.com/en-us/library/bb337697.aspx) | Determines whether a sequence contains any elements. |
| [Average](http://msdn.microsoft.com/en-us/library/bb354760.aspx) | Computes the average of a sequence of [Decimal](http://msdn.microsoft.com/en-us/library/system.decimal.aspx) values. |
| [Concat](http://msdn.microsoft.com/en-us/library/bb302894.aspx) | Concatenates two sequences. |
| [Contains](http://msdn.microsoft.com/en-us/library/bb352880.aspx) | Determines whether a sequence contains a specified element by using the default equality comparer. |
| [Count](http://msdn.microsoft.com/en-us/library/bb338038.aspx) | Returns the number of elements in a sequence. |
| [DefaultIfEmpty](http://msdn.microsoft.com/en-us/library/bb360179.aspx) | Returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty. |
| [Distinct](http://msdn.microsoft.com/en-us/library/bb348436.aspx) | Returns distinct elements from a sequence by using the default equality comparer to compare values. |
| [ElementAt](http://msdn.microsoft.com/en-us/library/bb299233.aspx) | Returns the element at a specified index in a sequence. |
| [ElementAtOrDefault](http://msdn.microsoft.com/en-us/library/bb494386.aspx) | Returns the element at a specified index in a sequence or a default value if the index is out of range. |
| [Empty](http://msdn.microsoft.com/en-us/library/bb341042.aspx) | Returns an empty [IEnumerable(OfT)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) that has the specified type argument. |
| [Except](http://msdn.microsoft.com/en-us/library/bb300779.aspx) | Produces the set difference of two sequences by using the default equality comparer to compare values. |
| [First](http://msdn.microsoft.com/en-us/library/bb291976.aspx) | Returns the first element of a sequence. |
| [FirstOrDefault](http://msdn.microsoft.com/en-us/library/bb340482.aspx) | Returns the first element of a sequence, or a default value if the sequence contains no elements. |
| [GroupBy](http://msdn.microsoft.com/en-us/library/bb534501.aspx) | Groups the elements of a sequence according to a specified key selector function. |
| [GroupJoin](http://msdn.microsoft.com/en-us/library/bb534297.aspx) | Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys. |
| [Intersect](http://msdn.microsoft.com/en-us/library/bb460136.aspx) | Produces the set intersection of two sequences by using the default equality comparer to compare values. |
| [Join](http://msdn.microsoft.com/en-us/library/bb534675.aspx) | Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys. |
| [Last](http://msdn.microsoft.com/en-us/library/bb358775.aspx) | Returns the last element of a sequence. |
| [LastOrDefault](http://msdn.microsoft.com/en-us/library/bb301849.aspx) | Returns the last element of a sequence, or a default value if the sequence contains no elements. |
| [LongCount](http://msdn.microsoft.com/en-us/library/bb353539.aspx) | Returns an [Int64](http://msdn.microsoft.com/en-us/library/system.int64.aspx) that represents the total number of elements in a sequence. |
| [Max](http://msdn.microsoft.com/en-us/library/bb335614.aspx) | Returns the maximum value in a sequence of [Decimal](http://msdn.microsoft.com/en-us/library/system.decimal.aspx) values. |
| [Min](http://msdn.microsoft.com/en-us/library/bb298087.aspx) | Returns the minimum value in a sequence of [Decimal](http://msdn.microsoft.com/en-us/library/system.decimal.aspx) values. |
| [OfType](http://msdn.microsoft.com/en-us/library/bb360913.aspx) | Filters the elements of an [IEnumerable](http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx) based on a specified type. |
| [OrderBy](http://msdn.microsoft.com/en-us/library/bb534966.aspx) | Sorts the elements of a sequence in ascending order according to a key. |
| [OrderByDescending](http://msdn.microsoft.com/en-us/library/bb534855.aspx) | Sorts the elements of a sequence in descending order according to a key. |
| [Range](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.range.aspx) | Generates a sequence of integral numbers within a specified range. |
| [Repeat](http://msdn.microsoft.com/en-us/library/bb348899.aspx) | Generates a sequence that contains one repeated value. |
| [Reverse](http://msdn.microsoft.com/en-us/library/bb358497.aspx) | Inverts the order of the elements in a sequence. |
| [Select](http://msdn.microsoft.com/en-us/library/bb548891.aspx) | Projects each element of a sequence into a new form. |
| [SelectMany](http://msdn.microsoft.com/en-us/library/bb534336.aspx) | Projects each element of a sequence to an [IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) and flattens the resulting sequences into one sequence. |
| [SequenceEqual](http://msdn.microsoft.com/en-us/library/bb348567.aspx) | Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type. |
| [Single](http://msdn.microsoft.com/en-us/library/bb155325.aspx) | Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence. |
| [SingleOrDefault](http://msdn.microsoft.com/en-us/library/bb342451.aspx) | Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence. |
| [Skip](http://msdn.microsoft.com/en-us/library/bb358985.aspx) | Bypasses a specified number of elements in a sequence and then returns the remaining elements. |
| [SkipWhile](http://msdn.microsoft.com/en-us/library/bb549075.aspx) | Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. |
| [Sum](http://msdn.microsoft.com/en-us/library/bb298138.aspx) | Computes the sum of a sequence of [Decimal](http://msdn.microsoft.com/en-us/library/system.decimal.aspx) values. |
| [TakeWhile](http://msdn.microsoft.com/en-us/library/bb503062.aspx) | Returns a specified number of contiguous elements from the start of a sequence. |
| [ThenBy](http://msdn.microsoft.com/en-us/library/bb534743.aspx) | Performs a subsequent ordering of the elements in a sequence in ascending order according to a key. |
| [ThenByDescending](http://msdn.microsoft.com/en-us/library/bb534736.aspx) | Performs a subsequent ordering of the elements in a sequence in descending order, according to a key. |
| [ToArray](http://msdn.microsoft.com/en-us/library/bb298736.aspx) | Creates an array from a[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). |
| [ToDictionary](http://msdn.microsoft.com/en-us/library/bb549277.aspx) | Creates a [Dictionary(Of TKey, TValue)](http://msdn.microsoft.com/en-us/library/xfhwa508.aspx) from an [IEnumerable(OfT)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) according to a specified key selector function. |
| [ToList](http://msdn.microsoft.com/en-us/library/bb342261.aspx) | Creates a [List(Of T)](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx) from an[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). |
| [ToLookup](http://msdn.microsoft.com/en-us/library/bb549073.aspx) | Creates a [Lookup(Of TKey, TElement)](http://msdn.microsoft.com/en-us/library/bb460184.aspx) from an[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) according to a specified key selector function. |
| [Union](http://msdn.microsoft.com/en-us/library/bb341731.aspx) | Produces the set union of two sequences by using the default equality comparer. |
| [Where](http://msdn.microsoft.com/en-us/library/bb534803.aspx) | Filters a sequence of values based on a predicate. |
| [Zip](http://msdn.microsoft.com/en-us/library/dd267698.aspx) | Merges two sequences by using the specified predicate function. |

As you can see there’s lots of things to consider when implementing an interface. This is again where the MSDN helps immensely! Here’s one more interface example program and it also shows how to overload operators in VB. We’ll create a point class and then learn how to overload operations like + and –, as well as being able to compare two points to each other, which requires implementing IComparable. Here’s the program code:

'Chapter 16 – Program 10

'Show how to overload operators in VB and also implement the IComparable interface

Module Module1

Public Class Point

'We want to be able to compare two points to each other and

'determine which is "less" than the other -- notice less in

'quotes: this is a subjective process where we have to

'determine what's meant by that. In any case implement the

'IComparable interface so that we can compare things.

Implements IComparable

Public Property X As Integer

Public Property Y As Integer

Public Sub New(ByVal newX As Integer, ByVal newY As Integer)

X = newX

Y = newY

End Sub

Public Overrides Function ToString() As String

Return String.Format("[{0}, {1}]", X, Y)

End Function

'Overload binary operators: we can overload +, -, \*, /, \, &, ^, And, Or,

'Xor

Public Shared Operator +(ByVal point1 As Point, ByVal point2 As Point)

As Point

Return New Point(point1.X + point2.X, point1.Y + point2.Y)

End Operator

Public Shared Operator -(ByVal point1 As Point, ByVal point2 As Point)

As Point

Return New Point(point1.X - point2.X, point1.Y - point2.Y)

End Operator

'Overload unary operators: we can overload +, -, Not, IsTrue, IsFalse

Public Shared Operator IsTrue(ByVal point1 As Point) As Boolean

'If the point is the origin, return True

Return (point1.X = 0) And (point1.Y = 0)

End Operator

'If we do IsTrue, we must do IsFalse

Public Shared Operator IsFalse(ByVal point1 As Point) As Boolean

'If the point is not at the origin

Return (point1.X <> 0) Or (point1.Y <> 0)

End Operator

'Overload comparison operators: we can overload =, <>, <, >, <=, >=

Public Shared Operator =(ByVal point1 As Point, ByVal point2 As Point)

As Boolean

Return point1.Equals(point2)

End Operator

'If we do =, we must do <>

Public Shared Operator <>(ByVal point1 As Point, ByVal point2 As Point)

As Boolean

Return Not (point1.Equals(point2))

End Operator

'Overload the Comparison operators through the IComparable Interface:

'we can overload <, >, >=, <=

Public Function CompareTo(ByVal obj As Object) As Integer \_

Implements IComparable.CompareTo

If TypeOf obj Is Point Then

Dim aPoint As Point = DirectCast(obj, Point)

'We have to make some meaning to how to compare points

'So, if this X and Y are greater than the other point's

'X and Y, return 1 (this point is greater than aPoint)

If Me.X > aPoint.X And Me.Y > aPoint.Y Then

Return 1

End If

'So, if this X and Y are less than the other point's

'X and Y, return -1 (this point is less than aPoint)

If Me.X < aPoint.X And Me.Y < aPoint.Y Then

Return -1

Else

'Any other case, so equality or only the X or only the Y

'is greater/less than the other Point's X or Y returns 0

Return 0

End If

End If

End Function

Public Shared Operator <(ByVal point1 As Point, ByVal point2 As Point)

As Boolean

Return (point1.CompareTo(point2) < 0)

End Operator

Public Shared Operator >(ByVal point1 As Point, ByVal point2 As Point)

As Boolean

Return (point1.CompareTo(point2) > 0)

End Operator

End Class

Sub Main()

'Declare some points

Dim MyPoint1 As New Point(5, 7)

Dim MyPoint2 As New Point(8, 12)

Dim MyPoint3 As New Point(0, 0)

Dim MyPoint4 As Point

'Test True/False

If (MyPoint3) Then

Debug.WriteLine("MyPoint3 is at the origin")

Else

Debug.WriteLine("MyPoint 3 is not at the origin")

End If

'Test Binary Operators

MyPoint4 = MyPoint1 + MyPoint2

Debug.WriteLine("MyPoint1 + MyPoint2 = " & MyPoint4.ToString())

MyPoint4 = MyPoint1 - MyPoint2

Debug.WriteLine("MyPoint1 - MyPoint2 = " & MyPoint4.ToString())

'Test Equality Operators

Debug.WriteLine("MyPoint1 = MyPoint2 : {0} ", MyPoint1 = MyPoint2)

Debug.WriteLine("MyPoint1 <> MyPoint2 : {0} ", MyPoint1 <> MyPoint2)

'Test Comparison Operators

Debug.WriteLine("MyPoint1 < MyPoint2 : {0}", MyPoint1 < MyPoint2)

Debug.WriteLine("MyPoint1 > MyPoint2 : {0}", MyPoint1 > MyPoint2)

End Sub

End Module

Here’s the output we get:

MyPoint3 is at the origin

MyPoint1 + MyPoint2 = [13, 19]

MyPoint1 - MyPoint2 = [-3, -5]

MyPoint1 = MyPoint2 : False

MyPoint1 <> MyPoint2 : True

MyPoint1 < MyPoint2 : True

MyPoint1 > MyPoint2 : False

Pretty easy and quite similar to overloading operators in other languages! Also, just in case you didn’t think about it: we could implement IComparable to explain how to “rank” two values for sorting purposes…

# DoEvents

The DoEvents command allows the operating system to process events and messages that are waiting in the message queue. You should embed this command whenever you write a chunk of code that is processor-intensive. If you don't put DoEvents in, you may find that your computer becomes a paperweight until your application is done running. DoEvents is contained in the Application namespace.

Here’s a simple program that shows how DoEvents helps us out. This code is behind a form which has 2 command buttons and 1 label placed on it. The default names for the controls were used.

'Chapter 16 - Program 11

Public Class Form1

Dim lngCount As Long

Dim blnFlag As Boolean

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'Start a "busy" loop -- add one forever

blnFlag = True

Do While blnFlag

lngCount = lngCount + 1

Label1.Text = lngCount & " iterations"

'Here's the DoEvents statement to pause and give other

'applications a chance to get some processing done

Application.DoEvents()

Loop

MessageBox.Show("Loop interrupted after " & lngCount & " iterations.")

End Sub

Private Sub Button2\_Click(sender As Object, e As EventArgs)

Handles Button2.Click

'Click this button to make the loop stop

blnFlag = False

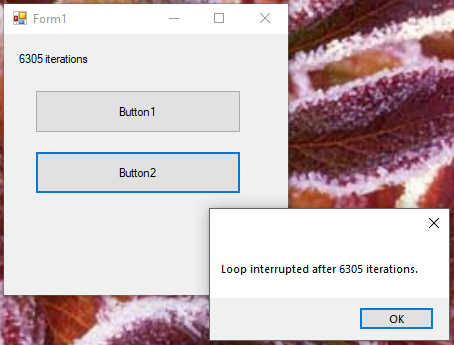
End Sub

End Class

When you run this application, the upper button will start the loop running, while the lower button is supposed to stop the loop. Run this program once to see that DoEvents allows your click to stop the loop (sort of, after a while, when it gets around to it).

Once you're convinced that you can stop the loop, remove the DoEvents statement. When you run the program again, you won’t be able to stop the loop! You won't even see the iteration label counter change and eventually you'll be told that your application isn't responding. Try it, but save your work first and good luck!

Here's a screenshot of the application while it's running, once the loop is stopped:



# Delegates

Delegates define arguments and return values and they are usually placed in an object of their own. We use them to support the processing of events, callback procedures and indirect calls to methods. You can think of a delegate as a pointer to a function. Delegates, events and objects all go hand-in-hand. You really need to understand a bit about them since they are heavily used in the .NET architecture.

Let’s look at a program that will create a delegate that can point at a math function:

'Chapter 16 - Program 12

Module Module1

'We will declare a delegate that can point at any other

'function that has the delegate's signature. In our case

'any function which takes two integers As arguments and sends

'back a single integer result: so the delegate could point

'at the math functions we are designing: +, -, \*, \

Public Delegate Function MathWithTwoArguments(ByVal Number1 As Integer,

ByVal Number2 As Integer) As

Integer

Public Class clsMathOperations

'Create a class with some methods we are interested in

'invoking with the delegate

Public Shared Function Add(ByVal Num1 As Integer,

ByVal Num2 As Integer) As Integer

Return Num1 + Num2

End Function

Public Shared Function Subtract(ByVal Num1 As Integer,

ByVal Num2 As Integer) As Integer

Return Num1 - Num2

End Function

Public Shared Function Multiply(ByVal Num1 As Integer,

ByVal Num2 As Integer) As Integer

Return Num1 \* Num2

End Function

End Class

'Note: Delegate target methods do not have to live inside of a class...

Public Function Divide(ByVal Num1 As Integer,

ByVal Num2 As Integer) As Integer

Return Num1 \ Num2

End Function

Sub PrintOut(ByVal Num1 As Integer, ByVal Num2 As Integer,

ByVal Op As Char, ByVal Answer As Integer)

Debug.WriteLine("{0} {1} {2} = {3}", Num1, Op, Num2, Answer)

End Sub

Sub Main()

'Create a delegate that points to the Add method in the class

Dim myDelegate As New MathWithTwoArguments(AddressOf

clsMathOperations.Add)

Dim intResult As Integer

'Since the delegate is pointing that Add function, we just

'need to send the numbers that we want added and call the

'delegate

intResult = myDelegate(5, 7)

PrintOut(5, 7, "+", intResult)

'Now make the delegate point at the Subtract function

myDelegate = New MathWithTwoArguments(AddressOf

clsMathOperations.Subtract)

intResult = myDelegate(7, 5)

PrintOut(7, 5, "-", intResult)

myDelegate = New MathWithTwoArguments(AddressOf

clsMathOperations.Multiply)

intResult = myDelegate(5, 7)

PrintOut(5, 7, "\*", intResult)

'Notice that the AddressOf is not pointing inside the class

'because we didn't define that function there. The delegate

'doesn't care as long as the signature is correct...

myDelegate = New MathWithTwoArguments(AddressOf Divide)

intResult = myDelegate(7, 5)

PrintOut(7, 5, "\", intResult)

End Sub

End Module

Here’s the output from the program:

5 + 7 = 12

7 - 5 = 2

5 \* 7 = 35

7 \ 5 = 1

While this was a brief introduction to delegates, you should at least have some understanding of what they are and how they work. You should also get some sense of how they, generics, anonymous types, lambdas and events all tie together. So, what’s cool about delegates? How about the ability to dynamically call a function at runtime when you don’t know which function will be needed? That’s pretty cool…

# Namespaces

Well we’ve certainly used a lot of them over the course of our discussions so far: System.IO, System.Data.SqlClient and so on. We’ll see even more down the road. Since we’ve really fleshed out object-orientation in this chapter and talked a bit about design, you might be chomping at the bit to bundle your own classes into a namespace. “How?” you ask. “No problemo!” I say. Just create a new class file and preface it with the Namespace keyword:

'Chapter 16 – Program 13

Namespace MyNamespace

Public Class Class1

'Let's just override Object's ToString...

Public Overrides Function ToString() As String

Return "I'm Class 1"

End Function

End Class

Public Class Class2

Public Overrides Function ToString() As String

Return "I'm Class 2"

End Function

End Class

Public Class Class3

Public Overrides Function ToString() As String

Return "I'm Class 3"

End Function

End Class

End Namespace

Now you can use this namespace in your own programs. Since we haven’t compiled it into a DLL or added it as a reference, we’ll have to just get by through using the scope resolver of the application – here’s a console application that exercises the new namespace:

'Chapter 16 - Program 13

Imports Program\_13.MyNamespace

Module Module1

Sub Main()

Dim a As New Class1

Dim b As New Class2

Dim c As New Class3

Debug.WriteLine(a.ToString())

Debug.WriteLine(b.ToString())

Debug.WriteLine(c.ToString())

End Sub

End Module

Since we identified where the namespace is located (in this case in the application) through an Imports statement, we can run the program and the compiler won’t throw any errors because it can find the namespace.

Here’s the output:

I'm Class 1

I'm Class 2

I'm Class 3