7

Inheritance

- Thou shalt not define an "has a" relationship
- Thou must not require too deep a class hierarchy
- Thou must require code re-use from a base class
- Thou shalt be defining a "is a" relationship
- Thou shalt require to extend a base class functionality to provide a wrapper class
- Thou must require one class change to affect all
- Thou must need to apply same methods and functionality to multiple data types

"Has a" and "Is a"

The relationships between classes can be defined in a number of ways. The simplest and most approachable is the "is a" and "has a" concept. An "is a" relationship will be the presentation of an object from a base class; inheritance.

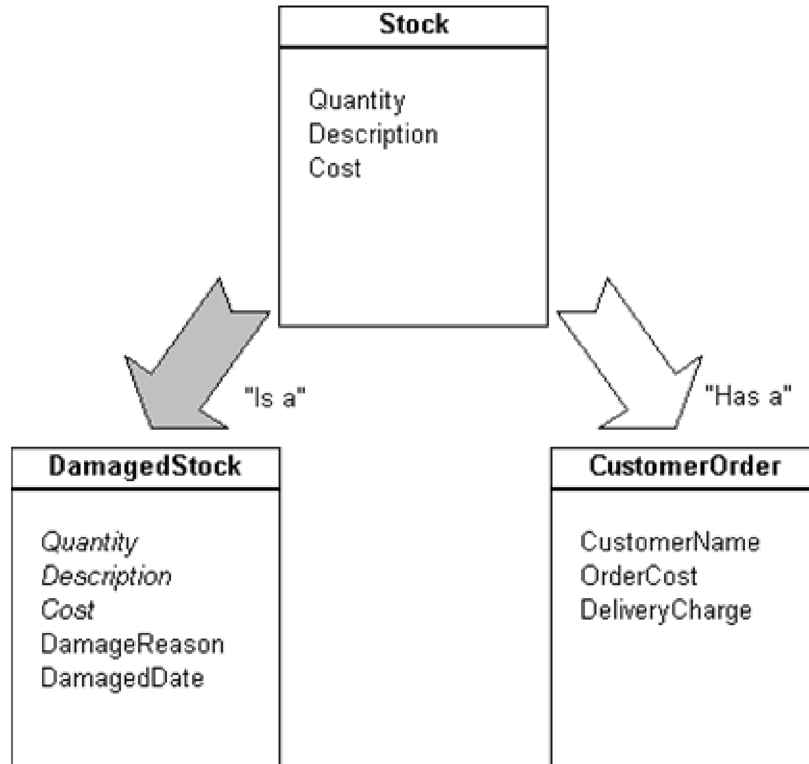
If we have a class named **Stock** and a class named **DamagedStock**, we are clearly demonstrating a "is a" relationship: **DamagedStock** "is a" **Stock**. On the other hand, a "has a" relationship would be presented more like the inclusion of a class named **CustomerOrders**. **CustomerOrders** "has a" **Stock** and in fact **CustomerOrders** "has a" **DamagedStock**.

The reason to create inherited classes using the concept "is a" makes life simple. If ClassA "is a" ClassB then we must assume that ClassA is everything that ClassB is, and then some (although it is technically possible to inherit a class from a base class without extending its functionality in any way, this exercise would be pointless).

Inherited classes inherit properties, events and methods that were defined in their base classes. Understanding this makes the "has a" concept clearer. Using the earlier example classes **Stock**, **DamagedStock** and **CustomerOrders**, we can include some properties to prove the point. If we state that the **Stock** class has a property of ***StockQuantity*** it is clear that **DamagedStock** "is" also going to acquire and require this property. If we add a property called ***DeliveryCharge*** to **CustomerOrders** it is immediately apparent that **Stock** will not require the use of this information.

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T" – Barbara Liskov, May 1988

The following illustration defines the "is a" and "has a" class relationships, the greyed arrow begin the inherited classes.



Code Reuse

If the requirement of inheritance is to develop a set of classes with core functionality to be reused in other applications or areas of your system then the advantages of inheritance are immense. The term "code reuse" applies simply to making use of existing code that has known functionality and has (or at least should have) been tested to ensure that it works correctly.

By inheriting from classes in order to make use of existing code we can develop libraries to assist common development procedures. In the unlikely event of these procedures not working correctly, we can alter a base class allowing all inherited classes to benefit from the change. The truth of the matter is that every time you declare an object from an existing built-in .NET class you are in fact utilising code reuse.

```
'The following line declares a variable of type String
Dim strCustomerName As String

If strCustomerName.Length > 0 Then
    'perform some code
End If
```

If Microsoft found that the Length() function in the String class has problems in certain circumstances they would simply make the change in their base class and then deploy the update to you; you would then benefit from this update without having to rewrite (or even redistribute) your application.

Suppose that the business application that you are developing requires that a customers zip code requires validation against an "address database" to confirm that the address given correctly matches the zip code. We could begin by creating a **Customer** class that looks something like the following.

```
Public Class Customer
    'The following varibles are simply created public
    'as no other processing requirements exist
    Public Address2 As String
    Public State As String
    Public Country As String
    Public CustomerName As String
    Public CustomerCode As Int32

    'local variable instances
    Private mstrAddress1 As String
    Private mstrZipCode As String

    Public Property Address1() As String
        Get
            Return mstrAddress1
        End Get
        Set(ByVal Value As String)
            'Validate a change has occured
            If mstrAddress1 <> Value Then
                If ValidateZip() Then
                    'Update if ok
                    mstrAddress1 = Value
                End If
            End If
        End Set
    End Property
```

```
    Public Property ZipCode() As String
        Get
            Return mstrZipCode
        End Get
        Set(ByVal Value As String)
            'Validate a change has occured
            If mstrZipCode <> Value Then
                If ValidateZip() Then
                    'Update if ok
                    mstrZipCode = Value
                End If
            End If
        End Set
    End Property

    Public Function ValidateZip() As Boolean
        'check that the zip is ok
        'ensure that we have valid data
        If mstrZipCode Is Nothing Then
            Debug.WriteLine("Zip Code is Empty")
            Return False
        End If
        If mstrZipCode.Length = 0 Then
            Debug.WriteLine("Zip Code has no charcters")
            Return False
        End If
        'data is valid so call an external application
        'to get an address back
        Dim strAddress1_Result As String
        strAddress1_Result = ExernalFunction(mstrZipCode)
        'verify the return
        Return strAddress1_Result.ToLower = _
               mstrAddress1.ToLower
    End Function
End Class
```

If we decided to add a **Suppliers** class to our application that also requires the same zip code validation we could inherit from a base class. In order to achieve a sensible base class we firstly need to extract the common functionality. In this case the address information is common so we extract the ***CustomerName*** and ***CustomerCode*** properties from the class and rename it from **Customer** to **Address**. This demonstrates that both **Suppliers** and **Customers** have addresses and each will be verified using the same logic. To achieve this we would declare both the **Suppliers** and **Customers** classes like so.

```
Public Class Customers
    Inherits Address

    Public CustomerName As String
    Public CustomerCode As Int32
End Class

Public Class Suppliers
    Inherits Address

    Public SupplierRef As String
    Public CreditLimit As Decimal
End Class
```

Whilst this example clearly demonstrates the ability of code reuse it also touches upon how easy it is for a developer to ignore the use of **interfaces**. The **Address** class could have been created as an interface if we simply required our classes to have the same properties and methods without the underlying code. The design of well-structured classes always requires the developer to consider the entire application before deciding exactly how to approach the engineering of the code. In our previous example we certainly had two base classes that would have provided the same core functionality, this meant that we had an ideal candidate for inheritance.

Global Changes

Suppose you developed a class that provided details for a **SalesOrder**. As the order can be taken over the phone, remotely and within the shop itself sub classing a **SalesOrder** class three separate times would allow us to treat these cases individually and equally. Each of the three scenarios require different properties and methods but common information; a phone sale "is a" **SalesOrder**.

A couple of years go by and your customer requests a few changes to the existing application. The requirements considered are simple and straightforward

1. The SalesOrderNumber needs prefixing with the SaleYear
2. Certain sales now contain alphanumeric characters within the SalesNumber instead of the numeric field definition that was originally proposed.

As you spent time developing you class hierarchy logically you now have a fighting chance of implementing your customer's changes without considerable downtime and bug introduction. You decide to firstly investigate an approach to resolve the year prefixing issue.

The solution to including the year within the SalesOrderNumber can be handled in two ways. You would either replace the currently `Property Get()` with an improved result or expose

another function/property to allow reading of this information. The approach taken depends entirely on the possible implications of your change. Modifying the current property would mean that ***no*** changes would be required to expose this information to the existing application.

```
'Old Class Definition
Public Class SalesOrder
    Public SaleYear As Int32
    'local variable instances
    Private mstrSalesOrderNumber As String
    Public Property SalesOrderNumber() As String
        Get
            Return mstrSalesOrderNumber
        End Get
        Set(ByVal Value As String)
            mstrSalesOrderNumber = Value
        End Set
    End Property
End Class

'New Class Definition
Public Class SalesOrder
    Public SaleYear As Int32
    'local variable instances
    Private mstrSalesOrderNumber As String
    Public Property SalesOrderNumber() As String
        Get
            Return SaleYear & "/" & mstrSalesOrderNumber
        End Get
        Set(ByVal Value As String)
            If Value.IndexOf("/") > 0 Then
                'A "/" was found so extract the number
                mstrSalesOrderNumber = _
                    Value.Substring(Value.IndexOf("/")+1)
            Else
                mstrSalesOrderNumber = Value
            End If
        End Set
    End Property
End Class
```

Alternatively the approach of adding a function would resolve this. The drawback of this is that the code that displays the information must be modified. In the example, a property lends itself best for this situation.

To resolve the second of the two requests we again have the options of extending the current functionality of the property or including an additional function. To prevent breaking the current interface and possible issues in any other classes inheriting from or using our class the solution to this problem is best resolved by adding a new property. A good rule of thumb is to add a property if the current property type is to change; this rule also applies to methods and events. This is demonstrated within the following code sample.

```
Public Class SalesOrder
    Public SalesNumber As Int32
    'local variable instances
    Private mstrSalesNumberAlpha As String

    Public Property SalesNumberAlpha() As String
        Get
            If mstrSalesNumberAlpha Is Nothing Then
                Return SalesNumber.ToString
            Else
                Return mstrSalesNumberAlpha
            End If
        End Get
        Set(ByVal Value As String)
            Try
                SalesNumber = Convert.ToInt32(Value)
                'clear local variable so we know we have
                'stored a number
                mstrSalesNumberAlpha = Nothing
            Catch e As FormatException
                'we have errored so store the property
                'into the string variable instead
                mstrSalesNumberAlpha = Value
                'clear the SalesNumber as this will
                'be the only way for existing classes
                'to know that things have changed
                SalesNumber = -1
            End Try
        End Set
    End Property
End Class
```

As you can see, we have prevented breaking the existing interface by simply extending it to include an additional property. Whilst the example demonstrates the assignment of a negative number to indicate that an alphanumeric SalesNumber has been assigned, users of the existing interface would of course be unaware of the new property and so those that were aware would also be aware of its behaviour.

Before we continue any further, a short time must be allowed to look at some of the inheritance keywords and what they mean

- **Inherits**, is a mechanism to specify a base class
- **NotInheritable**, prevents the developer from using this class as a base class
- **MustInherit**, ensures that the class is created ***only*** as a base class. An instance of a MustInheritable class cannot be created directly only classes inherited from it.
- **Overridable**, will allow a method or property to be overridden.
- **Overrides**, allows you to override an overridable method or property
- **NotOverridable**, ensures that inherited classes do not override this property or method
- **MustOverride**, insists that the derived class implements this property or method
- **Overloads**, allows the developer to declare a method or property with the same name as another overloaded one. The argument list for the new method or property must be different.
- **Shadows**, indicates that a method, property or event is shadowed by an identically named one.

Design Considerations

Even the most considered class hierarchies are likely to change with new requirements. The decisions made at the beginning of the development process will affect how easy your job will be later on.

To start with consider the data types of the information you are exposing whether this is in methods, properties or events. If you cannot be sure perhaps you consider exposing an object data type, use an Int64 instead of an Int32 or even offer a number of interfaces to cover all eventualities.

Expose items that you feel require exposing. As a rule keep things **Private**. By monitoring your scope declaration you will assist the reduction of possible implications in inherited classes. Exposing a **Private** as a **Public** down the road is much simpler than adding extra methods to compliment a change in a method that could have been **Private**.

If a member is included for use only in derived classes then these should be scoped as **Protected**. Again this allows us to control the deployment of our base class, we are in control of who sees what and when.

All base classes should be designed to be as generalised as possible. If your class design is too specific too early then the developer using you class to derive their own may find that they cannot

do quite what they want and revert to developing a class from scratch. This can be demonstrated by the creation of three classes: Automobile, SportsCar and HotRod.

```
Public Class Automobile
    Public WheelCount As Int32
    Public Doors As Int32
    Public Manufacturer As String
End Class

Public Class SportsCar
    Inherits Automobile

    Public TurboCharged As Boolean
    Public Spoiler As Boolean
End Class

Public Class HotRod
    Inherits SportsCar

    Public NitrosOxide As Boolean
    Public Parachute As Boolean
End Class
```

The class hierarchy exposes classes from the generalised, **Automobile** to the specific, **HotRod**. As a developer making use of these base classes you can choose to sub class from a specific class or sub class from a class higher up the hierarchy to create something new.

```
Public Class Truck
    Inherits Automobile

    Public LoadCapacity As Double
End Class
```

Default Behaviour

When we declare a new function or sub within a class that has been derived from another the default behaviour is to allow the routine to be overridden. It is good practice to ensure that you are aware of the default behaviour of inherited procedures to all you to code around the default behaviour to prevent confusion to other developers, yourself or even to cater for when the default behaviour changes. All of the following example functions are syntactically correct, but it is evident that the function without the default behaviour is much clearer.

```
Public Class Customer
    Private mdblOrderMargin As Double = 10.2

    Friend Function OutputMargin() As String
        'formats the double the locale percentage
        Return MarginPrefix() & _
               mdblOrderMargin.ToString("p")
    End Function

    Public Overridable Function MarginPrefix() As String
        Return "Margin: "
    End Function

    Public Overridable Property OrderMargin() As Double
        Get
            Return mdblOrderMargin
        End Get
        Set(ByVal Value As Double)
            mdblOrderMargin = Value
        End Set
    End Property
End Class

Public Class PreferredCustomer
    Inherits Customer

    Private Const mdblPreferredCustOffset As Double = 5

    Public Overrides Function MarginPrefix() As String
        'alter the return string
        Return "Pref Margin: "
    End Function

    Public NotOverridable Overrides Property _
     OrderMargin() As Double
        'reduce this preferred customer's margin to give
        'them a better deal
        Get
            Return MyBase.OrderMargin –
                   mdblPreferredCustOffset
        End Get
        Set(ByVal Value As Double)
            MyBase.OrderMargin -= mdblPreferredCustOffset
        End Set
```

```
    End Property
End Class
```

This code sample illustrates how confusing things can get. In the **PreferredCustomer** class both the `OrderMargin` and the `MarginPrefix` procedures override base class functionality. By default, the functions would allow themselves to also be **overridable**. This default behaviour has been prevented by the inclusion of the **NotOverridable** keyword. It is obvious to a developer making use of this class that we do not want the `OrderMargin` property overridden; we are in fact saying that whilst we are allowing inheritors to inherit from the **PreferredCustomer** class, the `OrderMargin` property cannot be further adjusted.

Another example of a default keyword is Overloads. A routine can be overloaded as long as the rules to overloading routines are obeyed. A useful tip is to include the **Overloads** keyword when you have in fact overloaded the procedure and exclude it otherwise. You will find your code easier to read and document when you can distinguish at a glance if the routine has one or multiple declarations.

```
'Overload sample
Public Class SalesOrder
    Public Class OrderLine
        Public ItemDescription As String
        Public ItemCost As String
    End Class

    Private Function GetNewOrderNumber() As Int32
        'this function accesses a database and
        'returns a valid order number, -1 is failure
        Return ExternalOrderNoFunction()
    End Function

    Public Overloads Sub ProcessOrder( _
                          ByVal Items() As OrderLine, _
                          ByVal Discount As Double)
        Dim objOrderLine As OrderLine

        'validate data
        If Items Is Nothing Then
            MessageBox.Show( _
                "Order contains no items", _
                "ProcessOrder Error")
            Exit Sub
        End If

        Dim intNewOrderNumber As Int32 = _
                                   GetNewOrderNumber()
```

```
        If intNewOrderNumber = -1 Then
            MessageBox.Show( _
                "Order number is invalid", _
                "ProcessOrder Error")
            Exit Sub
        End If

        For Each objOrderLine In Items
            'apply this discount
            objOrderLine.ItemCost -= _
                             objOrderLine.ItemCost * Discount
        Next objOrderLine
        'tidy up
        objOrderLine = Nothing

        'pass the new items and order number to an
        'external function that will commit to a
        'file/database
        ExternalOrderCommit(intNewOrderNumber, Items)
    End Sub


    Public Overloads Sub ProcessOrder( _
                          ByVal Items() As OrderLine)
        'this procedure allows a shortcut to process
        'an order without applying a discount
        ProcessOrder(Items, 0)
    End Sub
End Class
```

The second declaration of `ProcessOrder` appears further down the code output and could easily be forgotten that we have overloaded versions of this procedure. Using the **overloads** keyword has made the code easier to read. It should also be noted however that if you include the **overloads** keyword on one of your declarations of a procedure you ***must*** also include it on all other procedures with the same name. This is a very nice development aid. If more than one declaration of a procedure has been previously declared using the overloads keyword, when you try and overload it again within the same class, a design time error is generated if you do not include an **overload**.
There is one restriction against these rules of overloading, and that is you cannot use the **overloads** keyword against a classes constructor, or `New()`. This rule applies even if you have multiple constructors in your class.

Must Inherit Classes (Abstract Classes)

A class that has been defined as a MustInherit class cannot be instantiated from, only inherited. The choice to define an abstract class is probably one of the hardest class design decisions that

can be made by a developer. Abstract classes pose the closest similarities with interfaces and therefore can be considered to be the ones that developers confuse with interfaces; developers often create classes where interfaces would be more appropriate.
Without spending time discussing what interfaces are, especially as they are covered elsewhere in this book, I recommend that you spend a couple of moments looking at interfaces before continuing.
Abstract classes are usually partially implemented or not implemented at all. The most notable difference between an interface and an abstract class in VB.NET is that an abstract class can only be inherited by **one** class whereas a class may implement as many interfaces as is required. In addition, classes derived from abstract classes may also implement as many interfaces as is required.
Abstract classes are especially useful when it is required to offer a degree of functionality but also enforcing the sub-classer to implement functionality that is more pertinent to their requirements. Methods requiring implementation within abstract classes are denoted with the **MustOverride** keyword. The following example demonstrates an implementation of an abstract class:

```
Public MustInherit Class SecurityRights
    Private mstrUserName As String = ""

    Public Sub New()
        'Instantiate the class here
    End Sub

    Public Sub New(ByVal UserName As String)
        Me.New()
        If CanUserLogon() And LogOn(UserName) Then
            mstrUserName = UserName
        Else
            mstrUserName = ""
        End If
    End Sub

    Public Property UserName() As String
        Get
            Return mstrUserName
        End Get
        Set(ByVal Value As String)
            mstrUserName = Value
        End Set
    End Property

    Public MustOverride Function CanUserLogon() _
                         As Boolean
    Public MustOverride Overloads Function LogOn() _
                         As Boolean
```

```
    Public MustOverride Overloads Function LogOn _
                         (ByVal UserName As String) _
                         As Boolean
    Public MustOverride Sub LogOff()
End Class
```

In the above example, we created an abstract class that contained two implemented procedures, an implemented property and three unimplemented procedures. The class makes use of the unimplemented procedures within its implemented procedures. This offers the reasoning behind naming these types of classes abstract; the class does not care how a method comes to its conclusion, only that it does.

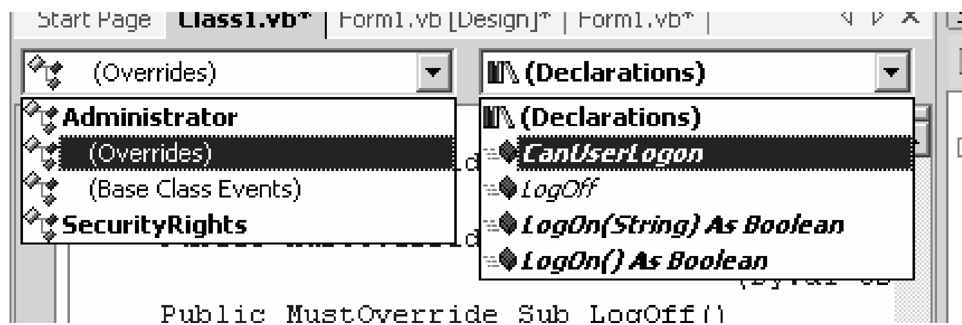
The example demonstrates that the instantiator requiring the argument of `UserName` needs to verify that both the user can logon and that the user does logon. How these processes evaluate these rules must be applied in the implemented class. Following is an example of what the implemented class may look like:

```
Public Class Administrator
    Inherits SecurityRights

    Public Overrides Function CanUserLogon() As Boolean
        'As we are an Administrator, we can log on
        Return True
    End Function

    Public Overrides Sub LogOff()
        'check that the user has no processes in
        'operation this check may simply see if the
        'user has open screens

        'clear the UserName property as this is how
        'we are determining that the user is logged on
        'or not

        UserName = ""
    End Sub

    Public Overloads Overrides Function LogOn() _
                            As Boolean
        If UserName.Length = 0 Then
            Return False
        Else
            Return True
        End If
    End Function
```

```
    Public Overloads Overrides Function LogOn _
                              (ByVal UserName As String) _
                              As Boolean
         Return True
     End Function
End Class
```

When an abstracted class is implemented, each of the abstract methods must also be implemented within the implemented class. The format of the implemented methods must be identical to that defined in the abstract class.

This process can be achieved automatically within the IDE by selecting Overrides from the first combobox in the IDE and then selecting the abstract procedure in the second combobox. The abstract methods can be identified by the fact that the text is italicised.



In order to decide whether to choose an interface or abstract class, following a number of simple rules makes the process easier.

- If you are designing large functional units as opposed to small outline definitions then an abstract class is more suitable than an interface.
- If a degree of common implemented functionality is required then you must create an abstract class as an interface only provides a template, it cannot contain any implementations.
- Abstract classes should be used for closely related objects, whereas an interface is used for when you require a range of disparate objects.
- If your class will require multiple versions then an abstract class will allow you to achieve this. Abstract classes facilitate versioning of your components where an interface should be left alone once deployed.

Scope

Deciding the scope for methods, events and properties is an important consideration during the design of your class hierarchy. How exposable something is and whom it is exposable to are in fact elements we have control over during the creation of base classes. Firstly, we should look at the scope operators available to us, and what their implications are. I use the terminology ***element*** to refer to a class, function, sub, event, property or variable in the following table:

Scope	**Description**
`Private`	The element can be referenced within the class itself.
`Public`	Elements defined as public are available outside of the class they are defined in. A public element benefits from the highest level of exposure.
`Friend`	A friend element is publicly available within the project containing the class (assuming that the scope of the class allows exposure)
`Protected`	A protected element can be accessed only be classes that inherit from the class containing the protected element. A protected element is in fact very similar to a friend with the exception that scope is restricted to inherited classes.
`Protected Friend`	A protected friend element is as the name implies, a combination of both the protected scope and friend scope.

The scope operations are restricted to certain types of declarations. The following table identifies the restrictions:

Type / **Scope**	**Class**	**Function / Sub**	**Event**	**Property**	**Variable**
`Private`	No[1]	Yes	Yes	Yes	Yes
`Public`	Yes[3]	Yes	Yes	Yes	Yes
`Friend`	Yes	Yes	Yes	Yes	Yes
`Protected`	No[1]	No[2]	No[2]	No[2]	No[2]
`Protected Friend`	No[1]	No[2]	No[2]	No[2]	No[2]

[1] ***Only within a type declared as public or friend***
[2] ***Only within a class.***
[3] ***Only if the class is not inheriting from a base class of lower scope precedence***

As is apparent from the tables, scope declaration follows a number of rules, most of which are common sense. As this chapter is primarily engaging the concept of inheritance, the only scope operators provided solely for classes are the **protected** and **protected friend** scope operators, these are the ones we should discuss further. The following example demonstrates the use of the **protected** scope operator in a base class:

```
Public Class Invoice
    Public ReadOnly Property InvoiceNumber() As String
        Get
            Return mstrInvoicePrefix & mintInvoiceId
        End Get
    End Property

    Protected Property InvoiceId() As Int32
        Get
            Return mintInvoiceId
        End Get
        Set(ByVal Value As Int32)
            mintInvoiceId = Value
        End Set
    End Property
    Private mintInvoiceId As Int32

    Public Property InvoicePrefix() As String
        Get
            Return mstrInvoicePrefix
        End Get
        Set(ByVal Value As String)
            'check values are different
            If mstrInvoicePrefix <> Value Then
                'assign to local variable
                mstrInvoicePrefix = Value
                'perform operation only if contains data
                If Not mstrInvoicePrefix Is Nothing Then
                    'upper case string
                    mstrInvoicePrefix = _
                              mstrInvoicePrefix.ToUpper
                End If
            End If
        End Set
    End Property
    'local variable, pre-initialised with data
    Private mstrInvoicePrefix As String = "INV"
End Class
```

Using the class shown previously we can create an inherited class that will make use of both the **public**

and **protected** properties:

```
Friend Class SalesInvoice
    Inherits Invoice

    Public Sub New()
        'Sales invoices have different prefix, this
        'is set within the constructor
        InvoicePrefix = "SAL"
    End Sub

    Public Sub RaiseInvoice()
        If InvoiceId = 0 Then
            'create a new invoice number
            InvoiceId = SomeExternalCall()
        Else
            'creating a duplicate invoice
        End If

        'call an external print function
        PrintInvoice(InvoiceId)

        MessageBox.Show("Invoice " & InvoiceNumber & _
                          " has been created", "Invoice")
    End Sub
End Class
```

The `RaiseInvoice` routine within the **SalesInvoice** class makes use of the **protected** property ***InvoiceId***. The ***InvoiceId*** property is considered to be **protected,** as the identifier is only necessary within the base class and any subclasses. A class created using either of these classes should have no need for this property and so exposure beyond this level could be dangerous. An example of code making use of the **SalesInvoice** class could be as follows:

```
Public Class CustomerOrder
    Public Sub CreateInvoice()
       'create a new instance of the SalesInvoice object
        Dim objInvoice As New SalesInvoice()

        'both the InvoiceNumber and InvoicePrefix
        'properties are exposed, the InvoiceId
        'property on the otherhand is not

        With objInvoice
            Debug.WriteLine(.InvoiceNumber)
            Debug.WriteLine(.InvoicePrefix)
```

```
            'run the invoice creation routing
            .RaiseInvoice()
        End With

        'tidy up
        objInvoice = Nothing
    End Sub
End Class
```

Simple Inheritance

Inheritance, unlike implements, allows you to create a class that is based on another class inheriting both its interfaces as well as its functionality. Simple inheritance has been covered throughout most of this chapter so far: the inheritance of one object from another.

Most base classes that we create are abstractions of required functionality. This is not the same as a truly abstract class defined earlier in this chapter, the previous example on abstraction illustrated a class that was abstracted deeply enough to **require** inheritance. The abstraction that we are about to discuss is rather the separation of entities into abstractions of their concepts.

For instance, an `Employee` object is an abstraction of an actual **Employee**. The object itself will probably not divulge information such as eating habits, favourite pets or whether they like reading. The object instead concentrates on what is similar and appropriate to **employees**. The concept of abstraction has been supported within VB since version 4 and the introduction of classes. Abstraction allows us to create an Employee object, this represents the abstracted employee, and we can then use this class to create instances of actual **employees**.

In simple terms: we abstract in order to concern ourselves with our requirements and not the actual information stored within the object.

```
Public Class Employee
    Public Property Name() As String
        Get
            Return mstrName
        End Get
        Set(ByVal Value As String)
            mstrName = Value
        End Set
    End Property
    Private mstrName As String

    Protected Property EmployeeId() As Int32
        Get
            Return mintEmployeeId
```

```
        End Get
        Set(ByVal Value As Int32)
            mintEmployeeId = Value
        End Set
    End Property
    Private mintEmployeeId As Int32

    Public Property ContactPhone() As String
        Get
            Return mstrContactPhone
        End Get
        Set(ByVal Value As String)
            mstrContactPhone = Value
        End Set
    End Property
    Private mstrContactPhone As String

    Public Function PerformPayroll() As Boolean
        'pay this employee
        Return True 'return the success of the operation
    End Function
End Class
```

Whilst this class represents an abstraction of an Employee, it is considered suitable within the requirements of our development. We understand that the employee is more than simply a number and name, as mentioned earlier they may have hobbies. You may, on occasion have a requirement to include other properties like hobbies, although the previous example of abstraction did not. In order to turn this abstracted representation of an Employee into a real employee we would:

```
    Private Sub Button1_Click( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
                          Handles Button1.Click
        'create an employee object
        Dim objEmployee As New Employee()

        With objEmployee
            .Name = "Joe Bloggs"
            .ContactPhone = "01234 56789"

            'the employee has now become an actual
            'person and so we can act on "them"
            'rather than "it"
            .PerformPayroll()
        End With
```

```
    End Sub
```

The ability to create a set of public interfaces whilst temporary ignoring the content of the interfaces, is called **encapsulation**. This allows us to create a set of public procedures stating the requirements of the object without implementing what those procedures will eventually do. In a large development team, this process is invaluable.

Imagine that you have a requirement to create invoices from your **Invoice Class**. Understanding the need for the requirement would be covered during abstraction of your base requirements. You have enough information to create an abstracted class that other developers can use before you have even written one line of code in your procedures. This concept is also important when you may not know the exact requirements for a function. Consider the example above, your customer may have expressed that they are unsure whether they would like the invoice exported into a word processing package or rather displayed as a report within the application. This decision does not require you to stop development whilst your customer contemplates their final requirements. Instead a team can steam ahead with the understanding that when they need to print an invoice, a procedure called `CreateInvoice` on the invoice class is available to do this.

A further benefit of encapsulation is apparent when your customer then changes their mind about how they want to print. Instead of word processor they choose report. This obviously causes no problem, you re-write the code within the `CreateInvoice` procedure to offer your customer the new requirement without breaking the existing code that implemented or used your class.

Consider the following example:

```
Public Class StockItem
    Public Property Quantity() As Int32
        Get
        End Get
        Set(ByVal Value As Int32)
        End Set
    End Property

    Public Property Description() As String
        Get
        End Get
        Set(ByVal Value As String)
        End Set
    End Property

    Public Property Cost() As Double
        Get
        End Get
        Set(ByVal Value As Double)
        End Set
```

```
    End Property

    Public Sub Reorder()
    End Sub

    Public Function IsStockLevelLow() As Boolean
    End Function
End Class

Public Class Stock
    Public Property StockItems() As StockItem()
        Get
        End Get
        Set(ByVal Value As StockItem())
        End Set
    End Property
End Class
```

The example encapsulates the requirements of your **StockItem** and **Stock** objects. Another developer takes your object and implements some code using the interface:

```
Public Class Form2
    Inherits System.Windows.Forms.Form

    Private Sub Button1_Click( _
                     ByVal sender As System.Object, _
                     ByVal e As System.EventArgs) _
                           Handles Button1.Click
        Dim objStock As New Stock()
        'reference the first stock item in the array
        Dim objStockItem As StockItem = _
                               objStock.StockItems(0)

        With objStockItem
            'output the values to debug
            Debug.WriteLine(.Quantity)
            Debug.WriteLine(.Description)
            Debug.WriteLine(.Cost)

            'check to see if we need to reorder
            If .IsStockLevelLow Then
                'reorder stock
                .Reorder()
            End If
        End With
```

```
        'tidy up
        objStockItem = Nothing
        objStock = Nothing
    End Sub
End Class
```

We have successfully implemented code based on a class that does nothing. We can now take the original **Stock** and **StockItem** classes and provide their functionality without breaking the code that uses our class or even informing the developer of what we have done. This is as follows:

```
Public Class StockItem
    Public Property Quantity() As Int32
        Get
            Return mintQuantity
        End Get
        Set(ByVal Value As Int32)
            mintQuantity = Value
        End Set
    End Property
    Private mintQuantity As Int32

    Public Property Description() As String
        Get
            Return mstrDescription
        End Get
        Set(ByVal Value As String)
            mstrDescription = Value
        End Set
    End Property
    Private mstrDescription As String

    Public Property Cost() As Double
        Get
            Return mdblCost
        End Get
        Set(ByVal Value As Double)
            mdblCost = Value
        End Set
    End Property
    Private mdblCost As Double

    Public Sub Reorder()
        'print out an order form
        'create an ordered stock item record
        'flag this item as on-order in the database
    End Sub
```

```
    Public Function IsStockLevelLow() As Boolean
        Return (mintQuantity < 5)
    End Function
End Class

Public Class Stock
    Public Property StockItems() As StockItem()
        Get
        End Get
        Set(ByVal Value As StockItem())
        End Set
    End Property
    Private mobjStockItems(1) As StockItem

    Public Sub New()
        Dim objStockItem As New StockItem()
        With objStockItem
            .Quantity = 100
            .Description = "Gizmo"
            .Cost = "10.54"
        End With
        'add to array
        mobjStockItems(0) = objStockItem
        With objStockItem
            .Quantity = 3
            .Description = "Thingy"
            .Cost = "872.76"
        End With
        'addd to array
        mobjStockItems(1) = objStockItem
        'tidy up
        objStockItem = Nothing
    End Sub
End Class
```

As you can see, the code has been added to make the classes more functional. As this is only an example I need not expand what the `Reorder` function does, a privilege bestowed upon me by **encapsulation**.

Inheritance comes with its own baggage of naming conventions. A number of years and languages have meant that there is a variety of ways of discussing the same thing. Visual Basic is in fact one of the most unconventional; most other languages adhere to similar conventions. Whilst this means we have to understand the way that other developers from different language backgrounds refer to aspects of inheritance, I feel that we have been blessed with the most user-

friendly syntax of any language. Consider the following sentences one from VB and the other from C#, I'm sure that you'll agree, VB is certainly easier to understand.

VB – A ***NotInheritable*** class is a class that cannot be inherited.
C# – A ***Sealed*** class is a class that cannot be inherited.

VB – An ***Overridable*** function can be overridden within a base class.
C# - An ***Abstract*** function can be overridden within a base class.

Whilst those that consider themselves to be hardcore object oriented programmers will surely disagree with me, I will stand my ground firmly. The idea of developing in an interpreted interface is to make writing code easier, and the closer the syntax is to that which we speak the easier it is to develop in. I'm sure you will make your own judgements, although at the end of the day, the keywords are a means to the same end.

As mentioned, there are several ways of describing the same thing. These ways are mixed and matched usually from the background of the person you are discussing inheritance with. To give you an understanding of same of the alternative ways of describing inheritance concepts, I will offer the VB equivalent.

To start with, inheritance in itself can often be referred to as **generalisation**. This is apparent when you undertake UML modelling procedures as the inheritance of objects is commonly referred to as generalising. The reason for this is that your **base class** is a more general version of your sub class.

If we create a class using inheritance, the class that we are inheriting from is called the **base class**. This type of class is also referred to as the **parent class** or **superclass**.

The class that inherits from the **base class** is called the **subclass**. This type of class also has a number of names, some of which are **child class** and **derived class**. If you decide that you prefer to refer to your **base class** as the **parent class**, you should also refer to your **subclass** as the **child class**; mixing conventions is bad practice and confusing to your peers.

Finally, as discussed at the beginning of this chapter, the inheritance relationship itself is also referred to as an **"is-a" relationship** or simply **deriving**; ClassB is **derived** from ClassA or ClassB **"is-a"** ClassA.

By now you should be comfortable with both why we use inheritance, the rules that help us decide the approach to take and an understanding of how we code such examples.

If any of this concepts are still unclear you should review the chapter thus for before moving into the following section. This section will presume that all these concepts are clear.

Advanced Inheritance

In actual fact I consider all forms of inheritance as advanced, the title of this section is simply to distinguish between the more basic methods of inheritance and those more complicated issues.

This section will approach the subjects of shadowing, overloading, overriding and polymorphism.

Polymorphism

Although polymorphism is not technically related to inheritance, polymorphic behaviour can be achieved within the boundaries of inheritance.

To implement polymorphism we need to simply start with a class that provides overridable methods:

```
Public MustInherit Class Order
    Public MustOverride Function CalculateCost( _
                          ByVal OrderCost As Double, _
                          ByVal DeliveryCost As Double) _
                           As Double
End Class
```

We then take this **base class** and inherit two **subclasses** from it:

```
Public Class SalesOrder
    Inherits Order

    Public Overrides Function CalculateCost( _
                        ByVal OrderCost As Double, _
                        ByVal DeliveryCost As Double) _
                         As Double
        Return OrderCost + DeliveryCost
    End Function
End Class

Public Class PurchaseOrder
    Inherits Order

    Public Property HandlingCharge() As Double
        Get
            Return mdblHandlingCharge
        End Get
        Set(ByVal Value As Double)
            mdblHandlingCharge = Value
```

```
        End Set
    End Property
    Private mdblHandlingCharge As Double = 13.2

    Public Overrides Function CalculateCost( _
                       ByVal OrderCost As Double, _
                       ByVal DeliveryCost As Double) _
                        As Double
        Return OrderCost + DeliveryCost - HandlingCharge
    End Function
End Class
```

Both of these classes are inherited from our original `Order` class. This means that they both have a common interface, both classes contain a `CalculateCost` function. The polymorphic behaviour comes simply from the fact that we can now implement code that does not care that `PurchaseOrder` and `SalesOrder` are different classes but only that they were derived from Order. The following code uses the previous classes to illustrate a working example of this process:

```
Public Class Form3
    Inherits System.Windows.Forms.Form

    Private Sub Button1_Click( _
                     ByVal sender As System.Object, _
                     ByVal e As System.EventArgs) _
                           Handles Button1.Click
        'create instances of our two order classes
        Dim objPO As New PurchaseOrder()
        Dim objSO As New SalesOrder()

        'show the results
        DisplayCalculatedCost(objPO, 12.34)
        DisplayCalculatedCost(objSO, 12.34)

        'tidy up
        objSO = Nothing
        objPO = Nothing
    End Sub

    Private Sub DisplayCalculatedCost( _
                              ByVal objOrder As Order, _
                              ByVal OrderCost As Double)
        Dim strMessage As String
        Dim dblDeliveryCharge As Double = 10

        'format the result into a locale currency
```

```
        strMessage = _
                  objOrder.CalculateCost(OrderCost, _
                  dblDeliveryCharge).ToString("c")

        'display a message with the value
        MessageBox.Show(strMessage, "Calculated Cost")
    End Sub
End Class
```

As is clearly demonstrated, the `DisplayCalculatedCost` routine does not care at all that we are passing both a `SalesOrder` class and a `PurchaseOrder` class into it. It requires that we pass an `Order` class, and as both `SalesOrder` and `PurchaseOrder` are **derived** from `Order` then we have successfully created a polymorphic function.

We can extend this functionality by checking the type of object passed in and then optionally performing an action on it.

```
    Private Sub DisplayCalculatedCost( _
                              ByVal objOrder As Order, _
                              ByVal OrderCost As Double)
        Dim strMessage As String
        Dim dblDeliveryCharge As Double = 10

        'format the result into a locale currency
        strMessage = objOrder.CalculateCost( _
                         OrderCost, _
                         dblDeliveryCharge).ToString("c")

        'display a message with the value
        MessageBox.Show(strMessage, "Calculated Cost")

        'check object type
        If objOrder.GetType.IsSubclassOf( _
                            GetType(PurchaseOrder)) Then
            Dim objPO As PurchaseOrder
            objPO = CType(objOrder, PurchaseOrder)
            MessageBox.Show( _
                objPO.HandlingCharge.ToString("c"), _
                "Handling Charge")
        End If
    End Sub
```

Notice the use of both the `GetType` method and the `GetType` operator. The `GetType` method returns the type of the class that we are accessing. The **type class** then in turn allows us to verify that the object is subclassed from a given type. The `GetType` operator returns the type

of object specified by a **typename**. Finally, we make use of the `CType` function to cast our `Order` object into a `PurchaseOrder` object, thus allowing use to make use of the `HandlingCharge` property that exists solely within the `PurchaseOrder` class.

Overloading

As we have discussed to principles of overloading to a degree in earlier sections of this chapter, you are probably familiar with some of the rules associate with procedure overloading. Before we begin to discuss some of the procedures of overloading, we should firstly familiarise ourselves with what is meant by a procedures **signature** (sometimes referred to as a prototype). The **signature** of a procedure is the combination of both its name and its argument list. It is the **signature** of the procedure that the compiler uses to identify your syntax whilst you are typing code in the IDE. This is demonstrated as follows using functions from previous code samples:

```
Private Sub DisplayCalculatedCost( _
                              ByVal objOrder As Order, _
                              ByVal OrderCost As Double)
End Sub
```

This example has a signature of:

```
method(Order, Double)
```

Whilst the function `CalculateCost`:

```
Public Overrides Function CalculateCost( _
                   ByVal OrderCost As Double, _
                   ByVal DeliveryCost As Double) _
                    As Double
    Return OrderCost + DeliveryCost – HandlingCharge
End Function
```

Has the signature of:

```
method(Double, Double)
```

As you can see, although the function returns a type it is not considered part of the procedures signature. This defines one of the rules of overloading; two or more functions cannot overload each other if they differ only by return types.

A method with optional parameters is considered to have multiple signatures, one for each variation of calling the procedure. Using the previous example and declaring the parameters as optional results in:

```
Public Overrides Function CalculateCost( _
             Optional ByVal OrderCost As Double = 0, _
             Optional ByVal DeliveryCost As Double = 0) _
                    As Double
    Return OrderCost + DeliveryCost - HandlingCharge
End Function
```

This results in the signatures:

```
method()
method(Double)
method(Double, Double)
```

Finally we should consider the signature resulting from a `ParamArray` argument type. As discussed earlier, the compiler creates a signature reference for each possible outcome of the function. In the case of a `ParamArray`, this list is infinite. This is demonstrated by:

```
Public Function CalculateCost( _
             ByRef OrderCost As Double, _
             ByVal ParamArray DeliveryCost() As Double) _
                    As Double
End Function
```

And resulting in the signatures:

```
method(ref Double)
method(ref Double, Double())
method(ref Double, Double)
method(ref Double, Double, Double)
method(ref Double, Double, Double, Double)
'...and so on
```

When you intend to have procedures you want to overload or when you wish to overload existing procedures there are a number of rules the apply:

- Each overload must differ by one or more of the following
  - Parameter order
  - Parameter Count
  - Parameter data types
- Each procedure should use the same procedure name; otherwise you are simply creating a new procedure.
- A **function** procedure can overload a **sub** procedure as long as they differ by

parameter listing (the return type rule applies here, a sub is a function is a sub).

- You cannot overload a method with a property of the same name and vice versa.

Overloading provides a neater alternative to using the optional argument type. As can be shown from the example above, the creation of two optional types results in three separate signatures for the function. Creating the function as three overloaded functions allows scope for developing each function further and clarifying the code somewhat:

```
Public Overloads Function CalculateCost() As Double
    Return MyClass.CalculateCost(0, 0)
End Function

Public Overloads Function CalculateCost( _
                 ByVal OrderCost As Double) _
                        As Double
    Return MyClass.CalculateCost(OrderCost, 0)
End Function

Public Overloads Function CalculateCost( _
                 ByVal OrderCost As Double, _
                 ByVal DeliveryCost As Double) _
                        As Double
    Return OrderCost + DeliveryCost - HandlingCharge
End Function
```

As you can see, we have implemented the original function is what is typically the “long hand” approach. Notice the use of the `MyClass` keyword to identify that we are calling the local version of the overloaded procedure. As VB.NET no longer supports the `IsMissing` function and that all optional parameters must have a value assigned, it makes sense to break the function up in this manner. Where we would have used the `IsMissing` function in the past, we now simply right the code in the relevant function.

Overriding

Consider the situation where we do not simply want to extent the base classes functionality but rather to replace it. Rather than leave an existing function alone we could choose to **override** it entirely. This is entirely possible as long as the base class allows it. Procedures cannot be overridden by default, the implications of having defaulted overridden procedures are obviously worrying. If the developer of the class that you intend to subclass from includes the `overridable` keyword then you can do just that; **override** the current functionality.

Returning to the SalesOrder class previously we could add a procedure that we intend to allow an inheritor to override, for reference we will also include a standard procedure:

```
Public Class SalesOrder
    Inherits Order

    Public Overrides Function CalculateCost( _
                        ByVal OrderCost As Double, _
                        ByVal DeliveryCost As Double) _
                         As Double
        Return OrderCost + DeliveryCost
    End Function

    Public Overridable Sub DisplayCost()
        MessageBox.Show(CalculateCost(0, 0), "Cost")
    End Sub

    Public Sub WriteCost
        Debug.WriteLine(CalculateCost(0, 0))
    End Sub
End Class
```

When we sub class from this base class we get an interesting result. Firstly, as expected we can override the `DisplayCost` routine to replace it with what we consider to be the improved output. Secondly, again as we expected, we cannot override the `WriteCost` routine. What is noticeable, and certainly much more interesting is that you can override the `CalculateCost` function. As explained previously, unless told otherwise, an overridden function will naturally allow itself to be overridden in an implemented subclass. This is certainly one to watch out for. To illustrate a class that is subclassed from `SalesOrder`, I have omitted an attempt to override `CalculateCost` as that was not the required outcome of this exercise and would not help to clarify the situation.

```
Public Class SupplierSalesOrder
    Inherits SalesOrder

    Public NotOverridable Overrides Sub DisplayCost()
        'prevent this method from being
        'overriden any further
        MessageBox.Show(CalculateCost(0, 10), _
                              "Supplier Cost")
    End Sub
End Class
```

The previous example not only illustrates the overriding of the `DisplayCost` routine, but also how the developer has chosen to prevent further overriding operations on this procedure.

As overriding is the approach that we take to implement polymorphism within inheritance, there is little more that can be said here that wasn't covered within the polymorphism section.

Shadowing

Shadowing is probably the most confusion of all the inheritance types. Whilst it bears similarities with **overriding**, the results are quite different.

When the base class designer has designed a class, as discussed earlier, considerations are made as to the scope of classes, procedures and so on. If the designer of the base class felt that a procedure should not be overridden, then they probably had good reason to do so. So that is it, we cannot override the procedure, wrong; we can **shadow** the existing implementation.

Before shadowing, careful consideration has to be made as to the implications of replacing the base class functionality. The class designer has probably made assumptions based on the fact that you should not be overriding his procedure. If you weigh up the consequences and are prepared to test thoroughly then the ability to shadow is a very useful concept.

As explained earlier, overloading allows us to replace a procedure with our own flavour, as long as this new procedure has a different signature. Whilst, on the other hand, shadowing replaces all current variations of the method with the one that we created using **shadows**.

If we extend the `PurchaseOrder` class that we created previously to contain an additional procedure we can demonstrate how shadowing affects subclasses.

```
Public Class PurchaseOrder
    Inherits Order

    Public Property HandlingCharge() As Double
        Get
            Return mdblHandlingCharge
        End Get
        Set(ByVal Value As Double)
            mdblHandlingCharge = Value
        End Set
    End Property
    Private mdblHandlingCharge As Double = 13.2

    Public Overloads Overrides Function CalculateCost( _
                   ByVal OrderCost As Double, _
                   ByVal DeliveryCost As Double) _
                    As Double
        Return OrderCost + DeliveryCost - HandlingCharge
    End Function

    Public Overloads Function CalculateCost( _
                   ByVal OrderCost As Double, _
                   ByVal DeliveryCost As Double, _
```

```
                   ByVal PackingCharge As Double) _
                    As Double
        Return (OrderCost + DeliveryCost) - _
               (HandlingCharge + PackingCharge)
    End Function
End Class
```

Now, we will subclass from the `PurchaseOrder` class into two separate classes. One of which will provide and demonstrate shadowing, the other the default behaviour.

```
Public Class PO_International
    Inherits PurchaseOrder

    Public Shadows Function CalculateCost( _
                          ByVal OrderCost As Double) _
                         As Double
        'international orders always have a
        'fixed 1000 price associated with them
        Return MyBase.CalculateCost(OrderCost, 1000, 500)
    End Function
End Class

Public Class PO_Local
    Inherits PurchaseOrder

    Public Overloads Overrides Function CalculateCost( _
                          ByVal OrderCost As Double, _
                          ByVal DeliveryCost As Double) _
                          As Double
        'A discounted delivery charge applies
        'to local orders
        Return MyBase.CalculateCost(OrderCost, _
                                   (DeliveryCost / 2))
    End Function
End Class

Public Class Form4
    Inherits System.Windows.Forms.Form

    Private Sub Button1_Click( _
                     ByVal sender As System.Object, _
                     ByVal e As System.EventArgs) _
                           Handles Button1.Click
        Dim objPO_Int As New PO_International()
        Debug.WriteLine("International")
        Debug.WriteLine(objPO_Int.CalculateCost(1234))
```

```
        Dim objPO_Local As New PO_Local()
        Debug.WriteLine("===============")
        Debug.WriteLine("Local")
        With objPO_Local
            Debug.WriteLine(.CalculateCost(10, 20))
            Debug.WriteLine(.CalculateCost(40, 30, 35))
        End With

        'tidy up
        objPO_Int = Nothing
        objPO_Local = Nothing
    End Sub
End Class
```

Notice the use of the `MyBase` keyword within the shadowing function:

```
Return MyBase.CalculateCost(OrderCost, 1000, 500)
```

By making use of MyBase, and as a result calling the underlying function, we have at least gone some way to prevent issues caused by the base class implementation not being called. The example produces the following output in the debug window:

International
1720.80
===============
Local
6.80
21.8

As you can see, by shadowing the procedure we have not only replaced it but have also prevented all other variations of it from becoming visible. Shadowing has one more string to its bow. By using shadowing you can replace a variable with a procedure:

```
Public Class StockItem
    Public Property Quantity() As Int32
        Get
            Return mintQuantity
        End Get
        Set(ByVal Value As Int32)
            mintQuantity = Value
        End Set
    End Property
    Private mintQuantity As Int32 = 0
End Class
```

```
Public Class StockItem_UI
    Inherits StockItem

    Public Shadows Function Quantity() As String
        Return "Quantity: " & MyBase.Quantity.ToString
    End Function
End Class
```

Again we have made use of the `MyBase` keyword to prevent further issues. Shadowing in this example has allowed us to create a class that exposes the `Quantity` property within a function.

I hope that by now you have become to appreciate some of the intricacies of applying and using inheritance. You should also understand how important it is to document your classes and their internal behaviour, without which an inheritor is working blind.

Summary

That is it, in a nutshell, inheritance. Without a doubt, a whole book could be dedicated to the intricacies of inheritance; although this chapter should have given you a practical thorough understanding of the why's and how's. We have covered a lot of ground including touching other subject areas such as implementation, such is how integrated inheritance is now to the VB.NET language.

We have discussed the syntax to create building block from which professional, well-organised classes can be derived from.

You should now be able to create abstract and base classes and also understand the subtle differences between the two. In addition we have covered the concepts of subclassing; that is creating a new class that both derives the interface as well as the functionality from a base class.

We have covered polymorphism, overloading, overriding and shadowing in some depth.

We have also discussed some of the rules that govern our inheritance development strategy. And understand that whilst rules can be broken, the essence of them is to assist both ourselves and other developers using our classes.

It has become apparent that there are many ways of approaching a solution. Each way offers it own merits and issues. Whilst we should consider that most approaches taken could be considered syntactically correct, they may not be technically correct. Yes, there is always more than one right way to do something, your job is to understand which right way is best for you, your team and of course your customer.