Inheritance and Polymorphism

The concept of inheritance is a common feature of an object-oriented programming language. Inheritance allows a programmer to define a general class, and then later define more specific classes that share or inherit all of the properties of the more general class. This allows the programmer to save time and energy that might otherwise be spent writing duplicate code.

Related to inheritance is the concept of polymorphism. Polymorphism allows us to define different methods with the same name, but have those methods do different things with different objects. This is related to method overloading, but as we will see is different in that the position of the object in the inheritance hierarchy determines what method gets invoked.

For example, perhaps we would like to build an application about candy. For starters, let’s say we want to do something with Twix bars and something with Reese’s Peanut Butter Cups. We might make classes like the following:

```csharp
Public Class Twix
    Private calories As Integer
    Private ingredients As ArrayList

    Public Sub New()
        calories = 580
        ingredients = New ArrayList
        ingredients.Add("Sugar")
        ingredients.Add("Chocolate")
        ingredients.Add("Caramel")
    End Sub

    Public Function GetInfo() As String
        Dim sIngred As String
        Dim i As Integer

        sIngred = CStr(ingredients(0))
        For i = 1 To ingredients.Count - 1
            sIngred = sIngred & " " & CStr(ingredients(i))
        Next
        Return "Calories: " & CStr(calories) & " Ingredients: " & sIngred
    End Function
End Class

Public Class Reeses
    Private calories As Integer
    Private ingredients As ArrayList

    Public Sub New()
        calories = 460
        ingredients = New ArrayList
        ingredients.Add("Sugar")
        ingredients.Add("Chocolate")
        ingredients.Add("Peanut Butter")
    End Sub
```
Public Function GetInfo() As String
    Dim sIngred As String
    Dim i As Integer

    sIngred = CStr(ingredients(0))
    For i = 1 To ingredients.Count - 1
        sIngred = sIngred & " " & CStr(ingredients(i))
    Next
    Return "Calories: " & CStr(calories) & 
        " Ingredients: " & sIngred
End Function
End Class

You should already be familiar with how one might use these classes. For example, the following code creates two candy bars and prints their info:

    Dim twixbar As New Twix
    Dim reesescups As New Reeses
    Console.WriteLine(twixbar.GetInfo)
    Console.WriteLine(reesescups.GetInfo)

This program outputs:

    Calories: 580 Ingredients: Sugar Chocolate Caramel
    Calories: 460 Ingredients: Sugar Chocolate Peanut Butter

This might be fine for some applications, but right off the bat we can see that we are duplicating a lot of the same code. For example, the GetInfo() subroutine is going to be the same for any candy bar. As the program is now, if we had 100 different candy bars, we would have 100 different GetInfo() subroutines.

Instead, we can take advantage of a natural ordering of candy bars. We can visualize the types of candy bars in a hierarchy as follows:
A Twix Bar is a Chocolate Bar which in turn is a Candy Bar. This means that a Twix bar has all the properties that Chocolate Bars have, which in turn have all the properties that Candy Bars have.

We can create a class for each type of candy and link them together as indicated in this hierarchy. The property of inheritance will give us the “isa” relationship so that anything we define for the Candy Bars class will automatically be inherited by all classes below it, saving us the trouble of re-defining them. A child or derived class is a class defined by adding instance variables and methods to an existing class. That existing class that we are building upon is called the base or parent class. For example, the Twix class is derived from the base class of Chocolate Bars. To create a derived class, we add the keywords Inherits <parent-class> to the class definition, followed by the name of the base class:

    class Twix
    Inherits ChocolateBars

The child classes inherit all of the public variables, properties, and methods from the parent class as well! This is nice because we get to reuse the same code that is already written for parent classes.

There is also a new categorization of class variables called protected. This modifier indicates that a variable is not accessible from outside the class, but is inherited by all children.

Here is an example for the Twix, Reeses, ChocolateBars, and CandyBars classes:
In this example, we define the calories and ingredients variables in the CandyBar class since these are variables that apply to any Candy Bar. These variables are inherited by all classes below it, so they automatically get access to the variables without having to redefine them.

At the CandyBar level we also have a function, GetInfo(). It returns a string of the calories and ingredients. It is also accessible by any class defined below it, so the function only exists in one place.

When we create an object, the constructors for all the parent classes will also be invoked. Consequently, when we make a Twix object, VB.NET will first invoke the constructor for CandyBar, then the constructor for ChocolateBar, and finally Twix would be last.

Here is our sample code:
Public Class CandyBar
    Protected calories As Integer
    Protected ingredients As ArrayList

    Public Sub New()
        ingredients = New ArrayList
        ingredients.Add("Sugar")   ' All candy bars have sugar
    End Sub

    Public Function GetInfo() As String
        Dim sIngred As String
        Dim i As Integer

        sIngred = CStr(ingredients(0))
        For i = 1 To ingredients.Count - 1
            sIngred = sIngred & " " & CStr(ingredients(i))
        Next
        Return "Calories: " & CStr(calories) & 
            " Ingredients: " & sIngred
    End Function
End Class

Public Class ChocolateBar
    Inherits CandyBar

    Public Sub New()
        ingredients.Add("Chocolate")   ' All chocolate bars have choc
    End Sub
End Class

Public Class Twix
    Inherits ChocolateBar

    Public Sub New()
        calories = 580
        ingredients.Add("Caramel")
    End Sub
End Class

Public Class Reeses
    Inherits ChocolateBar

    Public Sub New()
        calories = 460
        ingredients.Add("Peanut Butter")
    End Sub
End Class

Here is code that we can run, whose output is identical to before:

    Dim twixbar As New Twix
    Dim reesescups As New Reeses
    Console.WriteLine(twixbar.GetInfo)
    Console.WriteLine(reesescups.GetInfo)
When we create an instance of a Twix object (via Dim twixbar as new Twix) here is what happens:

1. The twix object inherits its own variables of "calories" and "ingredients"
2. The parent constructors are called first:
   a. CandyBar’s constructor adds “sugar” to ingredients
   b. ChocolateBar’s constructor adds “chocolate” to ingredients
   c. Twix’s constructor adds “caramel” to ingredients and sets calories to 580

Graphically, the Twix object looks something like this:

<table>
<thead>
<tr>
<th>Twix Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>calories = 580</td>
</tr>
<tr>
<td>Ingredients =</td>
</tr>
<tr>
<td>Sugar</td>
</tr>
<tr>
<td>Chocolate</td>
</tr>
<tr>
<td>Caramel</td>
</tr>
<tr>
<td>GetInfo()</td>
</tr>
</tbody>
</table>

Invoking GetInfo() outputs all of the ingredients and calories for each object. This eliminates repeat code since all child objects share the same code base.

We could also add specific subroutines, functions, or variables at lower levels of the hierarchy and those variables would only be accessible at that level or lower.

**Polymorphism and Overriding**

At times a programmer may want to define the same method for different classes in the inheritance hierarchy, but have the method do different things. This is possible though a construct called **polymorphism** and **overriding**. When we define a Sub or Function with the same name in the inheritance hierarchy, VB.NET will automatically use the method or property that is most specific to the object used.

To use this feature, we must add the keyword **overrides** to the methods we would like to allow to be overridden in the base class. For example, the following allows the getInfo() function to be overridden in the Twix class:
In CandyBar Class:

```vbnet
Overridable Function GetInfo() As String
    Dim sIngred As String
    Dim i As Integer
    sIngred = CStr(ingredients(0))
    For i = 1 To ingredients.Count - 1
        sIngred = sIngred & " " & CStr(ingredients(i))
    Next
    Return "Calories: " & CStr(calories) & 
    " Ingredients: " & sIngred
End Function
```

Now we can define a function of the same name in the Twix class except we use the keyword `overrides`. The MyBase.GetInfo() call invokes the parent definition of GetInfo:

In Twix Class:

```vbnet
Overrides Function GetInfo() As String
    Return MyBase.GetInfo() & ": Two for me, none for you"
End Function
```

Now if we run our code:

```vbnet
Dim twixbar As New Twix
Dim reesescups As New Reeses
Console.WriteLine(twixbar.GetInfo)
Console.WriteLine(reesescups.GetInfo)
```

We get:

```
Calories: 580 Ingredients: Sugar Chocolate Caramel Two for me, none for you
Calories: 460 Ingredients: Sugar Chocolate Peanut Butter
```

This feature can be very useful for customizing subroutines at more specific levels of the inheritance hierarchy while retaining the generality offered at higher levels of abstraction.

It turns out that every class we make is actually a descendant of the predefined class named `Object`. This means that every class we create will inherit methods from Object. This is how the ArrayList is able to handle an array of any data type – it is defined to hold the type Object which is a superclass of all other classes.

It turns out that VB.NET defines a function for the Object class. This function is then inherited automatically by every class we make. The function is `GetType()` which will return back the type of the class.

There are many other subtleties regarding the hierarchy of objects (e.g. we can assign a variable to be of type Twix to a variable defined of type CandyBar, but not vice versa) that we will skip here but you would cover in more detail in a class on Object Oriented Programming.