Introduction to Classes and Objects

A **class** is short for classification, and in object-oriented-speak, it corresponds to a user-defined specification for an object. The class is the definition; you can think of it like the blueprint for a complex device. It says how the device works, but if all you have is the blueprint then you can't actually use the device yet.

We will define classes with two major subcomponents:

Member Data; these are variables, properties, or adjectives of interest regarding the class. For example, if a class is "Auto" then the member data might specify variables to hold the make, VIN, owner, etc.

Methods; these are functions or subs that perform some action regarding the class. For the "Auto" class we might have methods to brake, accelerate, shift, etc.

A class only specifies how some device works, like a blueprint. To use the device, you need some **instantiation** of the blueprint; i.e. the device must be constructed. The instantiation is called an **instance** and is sometimes referred to as the object. To create an instance of a class we use the keyword **new**. We've seen some examples of this already, e.g. with Lists and the Random class.

To make a new class, from the "Project" menu select "Add Class". Here is the format to define a class:

```
Class ClassName
    PublicOrPrivate ClassVar1 As DataType
    PublicOrPrivate ClassVar2 As DataType
    …
    PublicOrPrivate Function FunctionName..
    PublicOrPrivate Sub SubName…
    …
    PublicOrPrivate Property …
End Class
```

The PublicOrPrivate is either the word **Public** or **Private**. If set to public then the variable, function, subroutine, or property is accessible from outside the class (using the dot notation, e.g. varname.ClassVar1). If set to private then the variable, function, subroutine, or property is accessible only inside the class. This principle supports the notion of data-hiding; data and variables that the user doesn't need to see should be hidden to prevent them from being mucked up accidentally (e.g. the innards of the auto are hidden from a driver).

All of these options, variables, subroutines, functions, and properties are optional. We'll talk about what the properties do a bit later.

Here is an example for a simple Money class. For now it only contains variables; later we will add some subroutines and functions and properties.

```
Public Class Money
    Public intDollars As Integer
    Private intCents As Integer
End Class
```

From another place in our program, such as in a button click event on a form:

```
Dim m1 As New Money
m1.intDollars = 10    ' Legal, Public
m1.intCents = 20            ' ILLEGAL, Private
```

This program will generate a compiler error since we are trying to access a private variable from outside the class.

For now let's set both member variables in the class to public:

```
Public Class Money
    Public intDollars As Integer
    Public intCents As Integer
End Class
```

Here is some code that uses this class:

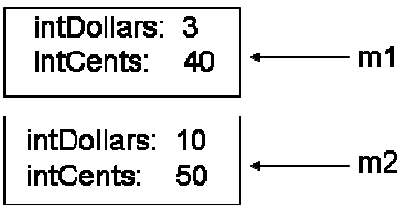
```
Dim m1 As New Money
Dim m2 As New Money

m1.intDollars = 3
m1.intCents = 40
m2.intDollars = 10
m2.intCents = 50
Console.WriteLine(m1.intDollars & " " & m1.intCents)
Console.WriteLine(m2.intDollars & " " & m2.intCents)
```

The output of this program is:

3 40
10 50

When the program reaches the WriteLine statement, we have created two separate instances of the Money object, each with different values stored in their member variables:



This can be quite convenient, because we can now associate multiple variables together in a single object. While both of these variables were of type integer in this example, the types could be anything. For example, a class to represent an Employee might contain variables like the following:

```
Class Employee
        Public strName As String
        Public intAge As Integer
        Public dblHourlyWage As Double
        Public strID As String
End Class
```

In this example, we are associated different variable types with the Employee object. This is a powerful construct to help organize our data efficiently and logically.

Class Constructors

Because we use classes to encapsulate data types, it is essential that class objects be initialized properly. When we defined the Money class, we were relying upon the user to set the value for dollars and cents outside the class. What if the client forgets to initialize the values? This can be such a serious problem that VB.NET provides a mechanism to guarantee that all class instances are properly initialized, called the class constructor.

A class constructor is a Sub with the special name of **New**. We can even make multiple constructors with multiple parameters, to differentiate different ways a class may be initialized. Below are two constructors for the Money class along with a subroutine to print the currency value:

```
Public Class Money
    Private intDollars As Integer
    Private intCents As Integer

    ' This is the default constructor, invoked if we create
    ' the object with no parameters
    Public Sub New()
        intDollars = 1
        intCents = 0
    End Sub

    ' This constructor invoked if we create the object with
    ' a dollar and cents value
    Public Sub New(ByVal intNewDollars As Integer, _
                   ByVal intNewCents As Integer)
        intDollars = intNewDollars
        intCents = intNewCents
    End Sub
```

```
    ' This subroutine prints the value out
    Public Sub PrintValue()
        Console.WriteLine(intDollars & "." & intCents)
    End Sub
End Class
```

Code in a buttonclick event to use the money class:

```
Dim m1 As New Money
Dim m2 As New Money(5, 50)

m1.PrintValue()
m2.PrintValue()
```

When this program runs, the output is:

1.0 ← From m1
5.50 ← From m2

When we create m1, we give no parameters in Money(). This invokes the default constructor, which is given as:

```
Public Sub New()
```

This code initializes dollars to 1 and cents to 0.

When we create m2, we give two parameters in Money(5,50). VB.NET will then search for a constructor that has two parameters that match the ones provided. The constructor that is found is then invoked:

```
Public Sub New(ByVal intNewDollars As Integer, _
               ByVal intNewCents As Integer)
```

This code then sets the member variables to the input parameters, resulting in the output previously specified.

Let's add some more code to the Money class to make it a bit more useful. In particular, since the class variables are private, let's make a way for the user of the class to get and set the dollars and cents. One way to provide access is through a **property** block. The **Get** procedure of the property block is used to retrieve the value of a variable. The **Set** procedure of the property block is used to set the value of a variable.

Add the following to the Money Class:

```
' Provide access to get and set the dollars variable
Public Property Dollars() As Integer
    Get
        Return intDollars
    End Get
    Set(ByVal value As Integer)
        intDollars = value
    End Set
End Property

' Provide access to get and set the cents variable
Public Property Cents() As Integer
    Get
        Return intCents
    End Get
    Set(ByVal value As Integer)
        intCents = value
    End Set
End Property
```

We can now access the property "Dollars" and "Cents" from the caller:

```
Dim m1 As New Money
Dim m2 As New Money(5, 50)

m1.Dollars = 25     ' Uses the SET part of the Dollars Property
m1.Cents = 75       ' Uses the SET part of the Cents Property
m1.PrintValue()
' Use the GET part of Dollars and Cents the properties
Console.WriteLine(m2.Dollars & "." & m2.Cents)
```

The output is:

25.75 ← From m1.PrintValue()
5.50 ← From Console.WriteLine(m2.Dollars & "." & m2.Cents)

One of the nice things about the Property block is that we can add more code to perform validation and control regarding what values we want to get and set. For example, let's say that we did the following:

```
Dim m1 As New Money

m1.Dollars = 25     ' Use the SET part of the Properties
m1.Cents = 375
m1.PrintValue()
```

The output is: 25.375

This is not really desirable, because it really means we have 375 cents, but this would probably be interpreted as 37.5 cents. A better solution would be to turn groups of 100 cents into dollars. We can add the proper logic to do this in the Property block:

```
' Provide access to get and set the cents variable
Public Property Cents() As Integer
    Get
        Return intCents
    End Get
    Set(ByVal value As Integer)
        ' Increment dollars if we have more than 99 cents
        If value > 99 Then
            intDollars += value \ 100
            intCents = value Mod 100
        End If
    End Set
End Property
```

The output now becomes: 28.75

Notice the abstraction we have implemented in the Cents property. If we ever add together something more than 100 cents, then we automatically update the cents into dollars. This logic is hidden for us in the Money class simply by executing the line of code:

```
moneyVar.Cents = 375
```

If we had just made the dollars and cents variables be public variables, then it would allow the outside user to set the cents to values over 100, possibly causing errors in how the program is interpreted.

By also making the dollars and cents private, we have the option to change the internal details and have these changes hidden from a user of the class. For example, we might decide to use a single variable of type Double to store the dollars and the cents. Then we could provide the proper logic in the Dollars and Cents properties to set and extract the dollar and cent amounts out of the Double and return them as an Integer. The user of the class won't see any difference.

Let's look at an example using objects and then continue with inheritance and polymorphism.

As an example, let's re-write the Trivia game. We'll make a single List that holds the trivia data instead of creating separate arrays for the question, answer, value, and whether or not a question was used.

First, here is the Trivia class. In addition to the question, answer, value, and used information, it also has a constructor for setting everything when the object is made:

```
Public Class Trivia
    Private strQuestion As String
    Private strAnswer As String
    Private intValue As Integer
    Private blnUsed As Boolean

    ' Constructor to initialize a new Trivia question
    Public Sub New(ByVal strNewQuestion As String, _
                   ByVal strNewAnswer As String, _
                   ByVal intNewValue As Integer)
        Me.blnUsed = False
        Me.strQuestion = strNewQuestion
        Me.strAnswer = strNewAnswer
        Me.intValue = intNewValue
    End Sub

    ' Properties for each member variable
    Public Property Question() As String
        Get
            Return Me.strQuestion
        End Get
        Set(ByVal value As String)
            Me.strQuestion = value
        End Set
    End Property

    Public Property Answer() As String
        Get
            Return Me.strAnswer
        End Get
        Set(ByVal value As String)
            Me.strAnswer = value
        End Set
    End Property

    Public Property Value() As Integer
        Get
            Return Me.intValue
        End Get
        Set(ByVal value As Integer)
            Me.intValue = value
        End Set
    End Property

    Public Property Used() As Boolean
        Get
            Return Me.blnUsed
        End Get
        Set(ByVal value As Boolean)
            Me.blnUsed = False
        End Set
    End Property
End Class
```

First let's declare our class level variables. We only need to store one List of Trivia objects. The list will keep track of how many trivia questions we have added, so we don't need a separate variable to count the number of entries:

```
    Dim listTrivia As New List(Of Trivia)
```

We can delete these variables:

```
    Dim listQuestions As New List(Of String)
    Dim listAnswers As New List(Of String)
    Dim listValues As New List(Of Integer)
    Dim listQuestionsUsed As New List(Of Boolean)
```

Let's start fixing our program by modifying the Load event where we load in all the questions. We had the following:

```
 For i = 0 To intNumQuestions - 1
    strQuestion = triviaFile.ReadLine
    strAnswer = triviaFile.ReadLine
    strTemp = triviaFile.ReadLine
    listQuestions.Add(strQuestion)
    listAnswers.Add(strAnswer)
    listValues.Add(CInt(strTemp))
    listQuestionsUsed.Add(False)
Next
```

This gets changed to create a new Trivia object each iteration of the loop and adding it to the list:

```
For i = 0 To intNumQuestions - 1
   strQuestion = triviaFile.ReadLine
   strAnswer = triviaFile.ReadLine
   strTemp = triviaFile.ReadLine
   Dim intValue As Integer = CInt(strTemp)
   Dim triviaQ As Trivia = New Trivia(strQuestion, strAnswer, intValue)
   listTrivia.Add(triviaQ)
Next
```

Getting most of our helper functions to work with the new structure requires accessing listTrivia followed by the index and the property we want for that trivia question:

```
    Public Sub PickRandomQuestion()
        ' Loop until we find a question we haven't asked before,
        ' i.e. blnQuestionsUsed is false for that question
        Do
            intQuestionIndex = rand.Next(0, listTrivia.Count)
        Loop Until listTrivia(intQuestionIndex).Used = False
        listTrivia(intQuestionIndex).Used = True
    End Sub
```

```
    Public Sub ShowQuestion()
        Me.lblQuestion.Text = listTrivia(intQuestionIndex).Question
        Me.lblQuestionNum.Text = "Question #" & CStr(intWhichQuestion)
        Me.txtAnswer.Text = ""
    End Sub

    Function IsCorrect(ByVal strGuess As String) As Boolean
        If (strGuess.ToLower =
listTrivia(intQuestionIndex).Answer.ToLower) Then
            Return True
        Else
            Return False
        End If
    End Function
```

For the button click event, this time I saved a reference to the current trivia object in a variable (triviaQ) and used that instead of referencing the arrays all the time:

```
    Private Sub btnAnswer_Click(. . .) Handles btnAnswer.Click
        Dim sGuess As String
        sGuess = Me.txtAnswer.Text
        ' Get current trivia object
        Dim triviaQ As Trivia = listTrivia(intQuestionIndex)

        If IsCorrect(sGuess) Then
            Dim points As Integer
            points = triviaQ.Value  ' Use current trivia object's value
            MessageBox.Show("That's Right! You got " & CStr(points) & "
points.")
            Me.intScore += points
        Else
            Dim points As Integer
            Dim strAnswer As String
            strAnswer = triviaQ.Answer
            points = triviaQ.Value
            MessageBox.Show("That's wrong! You lost " & CStr(points) &
" points.  The answer is " & strAnswer)
            Me.intScore -= points
        End If
        intWhichQuestion += 1
        If intWhichQuestion = 5 Then
            MessageBox.Show("Game over.  Your score is " & Me.intScore)
            Me.Close()
        Else
            PickRandomQuestion()
            ShowQuestion()
        End If
        Me.txtAnswer.Focus()
    End Sub
```

The program runs the same as before, but programmers most would argue that it is organized in a way that is easier to read and modify if adding more features to the project.

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 (i.e. subroutines and functions) with the same name, but have those methods do different things with different objects.

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:

```
Public Class Twix
    Private intCalories As Integer
    Private ingredients As New List(Of String)

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

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

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

Public Class Reeses
    Private intCalories As Integer
    Private ingredients As New List(Of String)

    Public Sub New()
        intCalories = 460
        ingredients.Add("Sugar")
        ingredients.Add("Chocolate")
        ingredients.Add("Peanut Butter")
    End Sub
```

```
    Public Function GetInfo() As String
        Dim strIngred As String
        Dim i As Integer

        strIngred = CStr(ingredients(0))
        For i = 1 To ingredients.Count - 1
            strIngred = strIngred & " " & CStr(ingredients(i))
        Next
        Return "Calories: " & CStr(intCalories) & _
               " Ingredients: " & strIngred
    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:

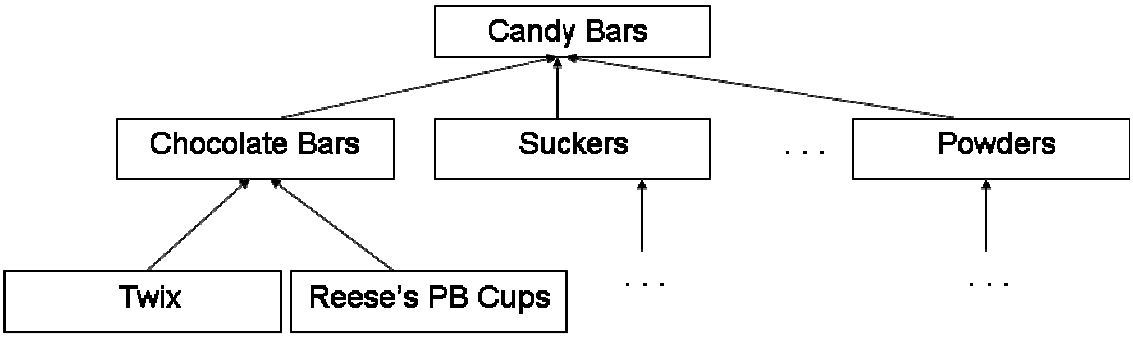
```
        Dim twixbar As New Twix
        Dim reesescups As New Reeses
        Console.WriteLine(twixbar.GetInfo)
        Console.WriteLine(reesescups.GetInfo)
```

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.

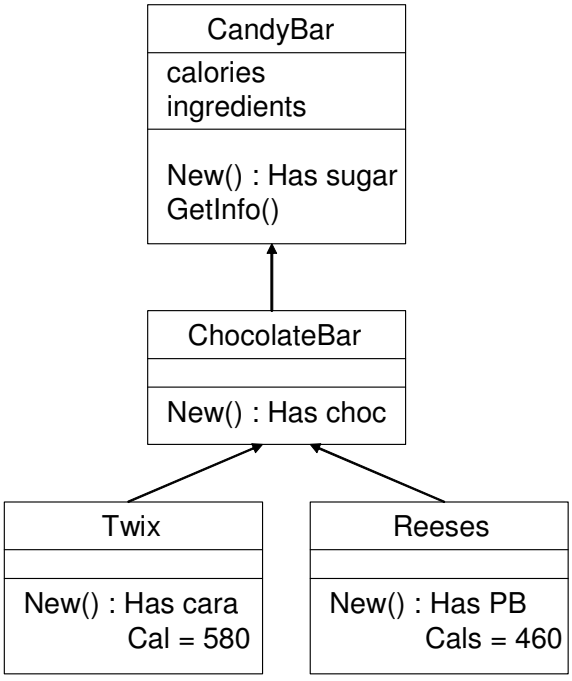
We can create a class for each type of candy and link them together as indicated in this hierarchy. The property of inheritance will give us the "isa" relationship so that anything we define for the Candy Bars class will automatically be inherited by all classes below it, saving us the trouble of re-defining 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 intCalories As Integer
    Protected ingredients As New List(Of String)

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

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

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

Public Class ChocolateBar
    Inherits CandyBar
    Public Sub New()
        ingredients.Add("Chocolate")        ' All candy bars have sugar
    End Sub
End Class

Public Class Twix
    Inherits ChocolateBar
    Public Sub New()
        intCalories = 580
        ingredients.Add("Caramel")
    End Sub
End Class

Public Class Reeses
    Inherits ChocolateBar

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

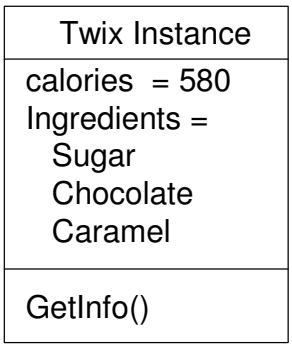
Here is code that we can run, whose output is identical to before:

```
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:



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 **overridable** 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:

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:

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

Now if we run our code:

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

We get:

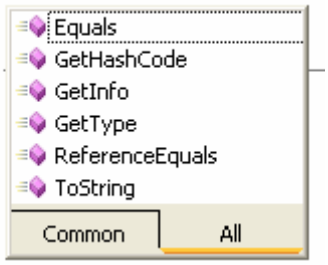
```
Calories: 580 Ingredients: Sugar Chocolate Caramel Two for me,
none for you
Calories: 460 Ingredients: Sugar Chocolate Peanut Butter
```

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.

It turns out that VB.NET defines several methods for the Object class. If you type any object and a dot, e.g.:

```
twixbar.
```



Then we get a pop-up window with all methods available. We wrote the GetInfo method, but where did Equals, GetType, ToString, etc. come from? The answer is that these methods were all inherited from the Object 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.