**Week 7 lab work**

Understandingvirtual methods and interface in C#.

**Task 1: Virtual methods**

In the last lab you saw how to create an abstract method, which is required to be overridden in all sub-classes. Also, the *abstract* method doesn’t have any logic in the class. In many cases you will want to write something in the base class, but have the option of overriding it later. In C#, these are called *virtual* methods. Let’s add a virtual method in *Person* class,

public virtual string SendMessage(string message)

{

var sb = new StringBuilder();

var timeStamp = string.Format("Sent on {0:D} at {0:t}", DateTime.Now);

sb.AppendLine(timeStamp);

sb.AppendLine("");

sb.AppendLine("Dear " + FirstName + ",");

sb.AppendLine(message);

return sb.ToString();

}

**Task 2: Overriding virtual methods in subclasses**

Remember that with an *abstract* method, you have to implement that in all subclasses , but *virtual* method is optional. Virtual Methods can be used as it is or  can be modified to do something else. Let’s try both here,

Use as it is in the *Teacher* class, but in *Student* write a different version it as follows

public override string SendMessage(string message)

{

var original = *base*.SendMessage(message);

var sb = new StringBuilder(original);

sb.AppendLine("This message is private and confidential.");

return sb.ToString();

}

*An object of Teacher* class( being a child class of *Person)* will inherit ( override in this case) the same version of *SendMessage*() method as in Person class. However, an object of *student* class is overriding the *Person* class *SendMessage()* method and add something of its own. This is just an example, you can write a completely different method in terms of logic in the overridden version in child class.

**Task 3 : Interfaces**

You can derive a child(derived) class from child class where the child classes have things in common that are expressed in the base class, and specialized elements are kept in the derivative classes. In another words, if two objects and thereby classes can form “is a” relationship then Inheritance relation exists.

But as I discussed in class there are many situations where there are two things that apparently seem to some have common things, but they don't have very much in common with regards to properties and cannot establish “is a” relationship. In these situations we use a construct called interface. An example that fits into our current project might be tracking student assignments. There are lots of different kinds of assignments. Let’s model a couple of them.

In PolytechLibrary, add a *class* and name it *Communication* paper and add following properties,

public class CommunicationPaper

{

public string Title { get; set; }

public int MinimumWordCount { get; set; }

public string PaperText { get; set; }

}

Add another *class*

public class ResearchPaper

{

public string Hypothesis { get; set; }

public string Materials { get; set; }

public string Method { get; set; }

public string Conclusion { get; set; }

}

**Adding an interface**

“An interface is a feature of Object Oriented Programming that allows you to require a class to implement certain properties and methods. This practice allows you to do a couple of different things. First of all, it kind of acts like a contract between all of your programmers. What you're effectively saying with an interface is that if you use an interface, you must implement a certain method signature or a certain property and it must be named a certain way. By doing that, you can actually sort of mimic the effects of an abstract class without requiring the rigid structure. The other upside to interfaces is that objects can actually implement multiple interfaces, whereas they can only inherit from one class. This allows you to define behaviors in an interface and implement those in different combinations on your objects without having to resort to a complicated hierarchical structure. So far, we have two assignment types. It stands to reason that you would want to be able to score both of the assignment types that we have right now” [1].

 So, let's add an interface.

Right-click PolytechLibrary -> Add-> New Item-> Interface.

While naming your Interface best practice is to start them with a letter “I”.

public interface IScored

{

float Score { get; set; }

float MaximumScore { get; set; }

}

You don't actually write logic or code in *interface*, you only define the structure. Having a class implement an interface looks exactly like inheritance. You use a colon just like you were inheriting from a class, but this time you use your interface; and this is one of the reasons why you want to be sure to name your interfaces beginning with a letter I, so that other programmers will understand that they're using an interface here rather than using a base class.

public class ScienceExperiment : IScored

{

public string Hypothesis { get; set; }

public string Materials { get; set; }

public string Method { get; set; }

public string Conclusion { get; set; }

public float Score { get; set; }

public float MaximumScore { get; set; }

}

Properties actually behave like an method in C# and thus can be put together in *interface.* An *interface* usually contains method which are to be implemented by the other class. We are going to a lot of built-in interfaces after you guys come back from the Easter break.

**Task 4: Reading and Writing from a file**

Read and try out some examples from the link below

<https://support.microsoft.com/en-nz/help/816149/how-to-read-from-and-write-to-a-text-file-by-using-visual-c>

Read about Streamreader and Streamwriter in C#.

**Task 5: Measure time taken by any of your program**

var watch = System.Diagnostics.Stopwatch.StartNew();

// the code that you want to measure

watch.Stop();

var elapsedMs = watch.ElapsedMilliseconds;

Console.WriteLine(elapsedMs);

**Task 6:**

Work on assignment.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[1] Credits: C# essentials, Bruce Van Horn.