**Week 6: Lab Work**

# **Task 1: Inheritance**

Let’s add a few more classes to the *PolytechLibrary*  which you created in last lab session.

* Add a *class* and name it *Teacher.cs*
* Add attributes(fields) such as FirstName , LastName, Email\_Address and Subject as properties.
* Add another *class* name it *Student.cs*
* Add some fields(properties) , FirstName , LastName ….

Wait, didn’t you add same fields already for *teacher class*. Teacher and students have things in common such as firstname, lastname, email address and so on. Unlike Teacher, students do not have subject to teach. We can make use of a OO concept called inheritance here. What we can do is define a *class*, say *person,* and put all the common fields (attributes) there. Then make *Student* and *Teacher* classes inherit all the common attribute from *Person class.*  Of course, we will add the unique fields of their own in their class definition.

* Add a *class* *person*
* Add common attributes to *Person class and remove those from Teacher* and *student class*

public class Person {

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

}

Also,

Teacher and Student should look like this,

public class Teacher : Person {

public string Subject { get; set; }

}

public class Student : Person {

}

Note the class names *Teacher* and *Student* is followed by a colon and then *Person*. The Person class is called *base* or *parent* class. Similarly *Teacher* and *student* classes are called *derived* or *child* or *subclass*.

One other thing, if you don’t want others who are using the PolytechLibrary to be able to use the Person class directly(by directly I mean creating an object and using it from their code) you can make person class abstract. In this way you are making others to use *Teacher* and S*tudent* class directly from their code.

* Add a keyword *abstract* before the *class* keyword in the definition of *person* class

public abstract class Person {

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

}

Let’s add some attribute to the empty *Student* class. A attribute called *GradLevel* to show what level they are studying in. What type of data a GradLevel would hold ? In our polytech we have level 5, level 7 and level 8 graduates. How would you model this fixed but discrete set of data ?

Enumeration datatype are perfectly suited in this situation. So let’s define a *enum* and call it *Gradlevels.*Aslo add a property to get and set *Gradlevels* field*.*

public class Student: Person {

public enum GradeLevels {Level\_5, Level\_7, Level\_8, Level\_9 }

public GradeLevels GradeLevel {get; set; }

}

**Task 2:** **Working with derived class objects**

Let’s go back to our PolytechFormApp. Double click the button code.

Try making it a new Person, i.e. create an object of Person class. Did you get any error why?

Create *Teacher* and *Student* objects and try and access all the fields.

**Task 3: Abstract methods**

Abstract classes are classes that are designed such that you can't instantiate them or use them directly. The same is true for abstract methods.

An abstract method is a method with no code in it. Let's say, for example, we want to implement a method that will allow us to get a grade point average, but it's going to be a different computation depending on whether you're talking about the teacher or the student. The student is going to have the average of all the grades for the semester or the period or whatever time period we're talking about and the teacher's grade point average is going to be the average grade point average for all of the students in this class. So it makes sense to have two different, completely different implementations of a function by the same name. So create the abstract method and then w override it in each of the subclasses with specific implementations for those classes.

public abstract float ComputeGradeAverage ( ) ;

In *Student* class

public override float ComputeGradeAverage ( ) {

return 4.0 f;

}

In *Teacher* class

public override float ComputeGradeAverage ( ) {

//TODO: fix the implementation later

return 0.0f;

}

Note: If a method(s) is declared abstract in base class all derived classes must have its definition (implementation). In this case method *ComputeGradeAverage* must be implemented by Student and Teacher classes. Otherwise, the code won’t compile. Also, note that you can also declare any method abstract in non-abstract class.

**Task 4: Testing the abstract method**

Now that you have an abstract method and an overridden version of it in our two subclasses, why don't we take a look at what it looks like in our test program ?

* Add two buttons and write code to test the overridden version of *ComputeGradeAverage* from both classes.

Task 4:

Do all the examples exercise from Lecture slides.

Codes are given here, try to understand:

1. How base and derived classes work in C#
2. How polymorphism is achieved in C#
3. What does extension methods do in C# and why it is useful?

Task 5:

Work on your assignment.

Credits: C# essentials, Bruce Van Horn.