Laboration – Objekt-orientering in c#

Following exercises will build on top of each other

Exercise 1.1 – Creating a class

Create a class that represents a person. Let the class hold information about the person first name, last name and age. Let the constructor take in above data as parameters and create get-only functionality. Write a method in the Program class that takes the person-datatype as a parameter and print out the information about that person. Create a few person objects and pass into the function.

Exercise 1.2 – Adding methods to the class

Add a method Speak that writes out "Hello my name is {first name} {last name}" to the screen when calling it on a person object.

Exercise 1.3 – Printing the person class directly

Try using the Console.WriteLine to write out the person object to the screen by passing the object as parameter. That is, Console.WriteLine(personObj). What is the output?

Exercise 1.4 – Printing the person class directly (Continued)

Now, override the default ToString() method in the person class, and let it return the first name, last name and age concatenated. Try calling the Console.WriteLine(personObj) and see what happens. What was the output?

Exercise 1.5 – Counting the number of Person-objects created

In this example, we are going to write the functionality for keeping count of the number of person object created. Write a static integer variable in the person class, and write a static method that increases this variable by one each time it's called and one method that returns the current value. Where could we call this method, so that it increases each time we create a new person?

Exercise 1.6 – Inheritance

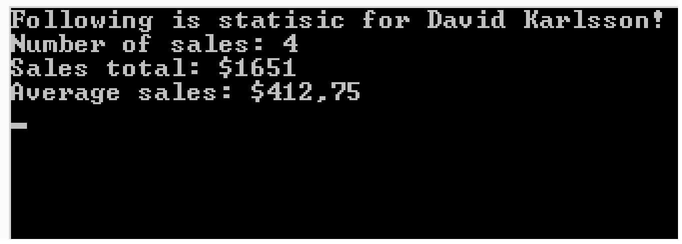
Write an employee class and a client class that are descendants of the person class. Implement a salary property on the employee class and create a constructor that calls the base person class constructor passing the information it needs. On both classes, override the ToString-method and let them return a distinct message depending whether they're clients and employees. Call the method you created in exercise 3 with both an employee and a client object.

Exercise 1.7 – Creating a sales class

In this exercise we want to create a class that hold a sale transaction that an employee makes with a client. The sales class should contain a name of the product sold, the money sold for in dollars (double), and a client class that references a client. Also, create a DateTime property that hold the information about when the sale was made. Let the employee class hold a collection of sales objects and implement a method that adds a new sales to the list. Also, create a constructor that instantiates the sales object. The date of the sale can be created in the constructor as the current date (DateTime.Now).

Exercise 1.8 – Calculating some sales statistics

Just storing each sales does not do us much good since the company is interested in how many sales an employee has made and how much money he have earned for the company. Now, they want you to create a few methods in the employee class that calculates these things. Let the first method be GetNumberOfSales(), which returns the number of sales. Let the second method be GetSalesTotal() that sums up the money sold for each element. Let the third method be to calculate the average sale for the employee. Then lastly, create a method that prints out the sales statistics for a specific employee.



```
Following is statisic for David Karlsson!
Number of sales: 4
Sales total: $1651
Average sales: $412,75
_
```

Exercise - Interfaces:

Create the interface *IDriveable* with the abstract methods *void StartEngine()*, *void StopEngine()*, *void Gas()*, *void Break()*, *void TurnLeft()* and *void TurnRight()*. Then create 3 classes Car, Motorbike and Bus. Implement the interface IDriveable to all three classes. Then create a method that takes an IDriveable object as input parameter that will call the methods in following sequence: *StartEngine()* -> *Gas()* -> *TurnLeft()* -> *TurnRight()* -> *TurnRight()* -> Break() and lastly *StopEngine()*. Try calling the method with all three interfaces. You do not have to create any real logic inside the methods, just print a distinct message like "Turned left (Motorbike)" and "Started the engine (Car)"s

How could interfaces be beneficial when creating an external library that someone else could use?

Side note: All collection classes actually implements the IEnumerable interface which makes it possible for them to be used in the foreach-loop. Even though a List and a Dictionary works internally different, they still *work since their implementation of the methods from is tailored to the specific the collection type. You can even implement IEnumerable in your own classes so they can be used in a forloop.*

Exercise – The school application

In assignment, you will learn to create a program that models a school. The school contains teachers and students and they can interact in different ways. The assignment is rather large so it will be divided into parts where we will create the different classes and then gradually combining them into a whole program. We will also modify the existing code with improvements as we go along, so remember to make a backup copy after each part. Do not however be intimidated to change your existing code, since this is a natural process in programming.

An overview of the classes we will be creating in this exercise:

- Student
- Teacher
- Course
- Grade
- School

To keep the project organized, create a new folder called Classes. Then as we go along, create all classes inside this folder as separate files. Keep in mind thought, when creating a new class by default, the accessor will be set to private. Make sure you add the public key word in the class declaration. Also, the namespace will instead become (Folder.ClassName), in this case Classes.NameOfClass since they reside inside the Classes folder. In the Program.cs-file, you can use "Using Classes;" to avoid typing the whole path.

<Picture>

Part 1. Creating the student class

The student class contains some basic information about a student.

Properties

In this class, you should implement the following properties:

- FirstName : String
- LastName : String
- DateOfBirth : DateTime
- StudentId : integer

Naturally for a student (or a human rather), the name and birthdate will not change so let all the above properties use Get-accessors only.

Methods

- Constructor
    - Implement a constructor with parameters for first name, last name and date of birth and let the student id be generated. Set the properties accordingly.

    Remember to create a private data member for each property, since the property itself will not allow us to set the value. You can use the readonly modifier on the data members. This makes sure that they cannot be changed during runtime except in the constructor.

- ToString()
    - Override the default ToString-method to return the students full name
- GetAge()
    - This method should calculate the students age based on his YearOfBirth and return it as a whole number. Tip: To get the current year, you can use DateTime.Now. Subtracting date gives back a timespan object which allows you to get the number of days between the two dates.

Part 2. Creating the teacher class

Properties

Like the student class, implement the properties FirstName, LastName, YearOfBirth and instead of StudentId, create the property TeacherId instead.

Methods

Implement the same methods as the student class, that is a Constructor, a ToString() method and a GetAge() method in the same way.

Part 3 – Creating the Course class

The Course class contains information about a specific course, like the Teacher responsible for the course, a list of students enrolled to it and a name.

Properties

- Name : String
- Teacher : Teacher
- Students : List<Student>

Methods

Part 4 - Creating the Grade class

The grade class will hold information about the grades the students have acquired during his time in school.

Properties

- Course : Course
- Student : Student
- Grading
    - This should be implemented as an enum-type. The members of the enum should be the grades A-F, where A is the highest and F is the lowest.
- DateAquired : DateTime

Methods

- ToString()
    - Override the default ToString()-method to return the name of the grade and a message depending on the Grading. What the message says, is up to you. Tip: when deciding which message to return, use a switch-statement for the Grading.

Part 5 – Refactor common behaviour into a new class

As you may have notices, the Teacher and Student classes contains a lot of redundant information. So you are now going to break out the common properties and methods into a Person class and then let the Teacher and Student class inherit from it. Tip: Check out the base-method when using the constructor.

Part 6 – Creating the School class

This class will serve as the common interface between the Students, Teachers, Course and grades. This class will handle enrolling a student to the school, enrolling a student to a specific course.

Properties

- Name : String
- Students : List<Student>
- Courses : Dictionary<String, Course>
- Teachers : List<Teacher>
- Grades : List<Grade>

Methods

- HasCourse(String) : bool
    - Checks whether the given course exists
- AddCourse(String, Course) : void
    - Add a new course to the school for the given key, if the course already exist, throw an exception.
- RemoveCourse(String, Course) : void
    - Removes a course for a given key. If the course does not exist, throw an exception.
- IsSchoolEnrolled(Student) : bool
    - Check if a student is enrolled to the school
- IsCourseEnrolled(Course, Student) : bool
    - Checks whether the student is enrolled to a given course.
- EnrollCourse(Course, student) : void
    - Enrolls a student to a given course. If the student is already enrolled, throw an exception.
- EnrollSchool(Student) : void

    - Enrolls the student to the school. If the student is already enrolled, throw an exception.
- WithrawFromCourse(Course, Student) : void
    - Removes a student from a given course, if the course or the student does not exist in the school, throw an exception.
- SetGrade(Grade, Course, Student) : void
    - Sets the students grade for a specific course. If the grade does exist, override the given value of that grade with the new one.
- GetGrades(Students) : List<Grade>
    - Returns the subset of the grades belonging to the given student

Since the Student, Teacher, Course and Grades are implemented as classes, they will be treated as reference types. This means you can compare them with the Equal-method. So when checking if a student exists, you can look for any student in the list of students where the object match with the given student.

Exceptions

If you want, feel free to create a new Exception class with a good name. For example CourseNotFoundException or StudentNotFoundException to use in your methods. In that case, you create a new class that inherits from "Exception". Else, you can use the ordinary Exception class for this example by using "throw new Exception("Student not found")".