Deadline for evaluation: 28-11-2016

(as usual) These exercises must be solved with a **test driven approach**: **one single** solution must contain one subproject for each of the exercises plus one unit testing project.

Exercise 01

Define a class **GSM**, which contains information about a **mobile phone**: model, manufacturer, price, owner, features of the battery (model, idle time and hours talk) and features of the screen (size and colors).
Declare several **constructors** for each of the classes created by the previous task, which have different lists of parameters (for complete information about a student or part of it). Data fields that are unknown have to be initialized respectively with **`null`** or **`0`**.
Add a **static field `samsungGalaxyS7`**, which stores information about mobile phone model Samsung Galaxy S7. Add a method to the same class, which returns information about this static field.
Add an **enumeration `BatteryType`**, which contains the values for type of the battery (Li-Ion, NiMH, NiCd, …) and use it as a new field for the class **`Battery`**. Add a method to the class **GSM**, which returns information about the object as a **`string`**. Define properties to encapsulate the data in classes **GSM**, **`Battery`** and **`Display`**.
Write a class **`GSMTest`**, which has to **test the functionality** of class **GSM** with a unit testing approach. Create few objects of the class and store them into an array. Check information about the created objects. Check information about the static field **`samsungGalaxyS7`**.
Create a class **`Call`**, which contains information about a call made via mobile phone. It should contain information about date, time of start and duration of the call. Add a property for keeping a **call history** – **`CallHistory`**, which holds a list of call records. In **GSM** class add methods for adding and deleting calls (**`Call`**) in the archive of mobile phone calls. Add a method, which deletes all calls from the archive. In **GSM** class, add a method that calculates the total amount of calls (**`Call`**) from the archive of phone calls (**`CallHistory`**), as the price of a phone call is passed as a parameter to the method.
Create a class **`GSMCallHistoryTest`**, with which to test the functionality of the class **GSM** as an object of type **GSM**. Then add to it a few phone calls (**`Call`**). Check information about each phone call. Assuming that the price per minute is **`0.37`**, calculate and Check the total cost of all calls. Remove the longest conversation from archive with phone calls and calculate the total price for all calls again. Finally, clear the archive.

Exercise 02

There is a **book library**. Define classes respectively for a **book** and a **library**. The library must contain a name and a list of books. The books must contain the title, author, publisher, release date and ISBN-number. In the class, which describes the library, create methods to add a book to the library, to search for a book by a predefined author, to display information about a book and to delete a book from the library.
Write a **test class**, which creates an object of type library, adds several books to it and checks information about each of them. Implement a test functionality, which finds all books authored by Stephen King and deletes them. Finally, check information for each of the remaining books.

Exercise 03

Write a class that **extracts** from an XML file the text only (without the tags). Write also the appropriate **unit tests**. Sample input file:

```
<?xml version="1.0"?>
<student>
  <name>Peter</name>
  <age>21</age>
```

Deadline for evaluation: 28-11-2016

```
  <interests count="3">
    <interest>Games</interest>
    <interest>C#</interest>
    <interest>Java</interest>
  </interests>
</student>
```

Sample output:

```
Peter
21
Games
C#
Java
```

Exercise 04

Implement the data structure dynamic **doubly linked list** (**`DoublyLinkedList<T>`**) – list, the elements of which have pointers both to the **next** and the **previous** elements. Implement the operations for adding, removing and searching for an element, as well as inserting an element at a given index, retrieving an element by a given index and a method, which returns an array with the elements of the list.

Implement **`DoubleLinkedListNode<T>`** class, which has fields **`Previous`**, **`Next`** and **`Value`**. It will hold to hold a single list node. Implement also **`DoubleLinkedList<T>`** class to hold the whole list.

Test all the implemented classes and functionalities with the appropriate **unit tests**.

Exercise 05

Let's have as given a graph **G (V, E)**. Write a program that **finds all connected components** of the graph, i.e. finds all maximal connected sub-graphs. A maximal connected sub-graph of **G** is a connected graph such that no other connected sub-graphs of **G**, contains it.

Test all the implemented classes and functionalities with the appropriate **unit tests**.

Exercise 06

Declare a `Time` structure that stores a time of day such as 10:05 or 00:45 as the number of minutes since midnight (that is, 605 and 45 in these examples). A struct type `Time` can be declared as follows:

```
public struct Time
{
    private readonly int minutes;
    public Time(int hh, int mm)
    {
        this.minutes = 60 * hh + mm;
    }
    public override String ToString()
    {
        return minutes.ToString();
    }
}
```

Deadline for evaluation: 28-11-2016

In the Time struct type, declare a read-only property Hour returning the number of hours and a read-only property Minute returning the number of minutes. For instance, `new Time(23, 45).Minute` should be 45.
Modify the `ToString()` method so that it shows a `Time` in the format hh:mm, for instance 10:05, instead of 605. You may use `String.Format` to do the formatting.
In the `Time` struct type, define two overloaded operators:

- Overload (+) so that it can add two `Time` values, giving a `Time` value.
- Overload (-) so that it can subtract two `Time` values, giving a `Time` value.

It is convenient to also declare an additional constructor `Time(int)`.
For instance, you should be able to do this:

```
Time t1 = new Time(9, 30);
Console.WriteLine(t1 + new Time(1, 15));
Console.WriteLine(t1 - new Time(1, 15));
```

In struct type Time, declare the following conversions:

- an implicit conversion from `int` (minutes since midnight) to `Time`
- an explicit conversion from `Time` to `int` (minutes since midnight)

For instance, you should be able to do this:

```
Time t1 = new Time(9, 30);
Time t2 = 120; // Two hours
int m1 = (int)t1;
Console.WriteLine("t1={0} and t2={1} and m1={2}", t1, t2, m1);
Time t3 = t1 + 45;
```

In the following you are asked to do some tests for better understanding difference between classes and structures: at the end, undo all these modifications and return to the current state of the implemented class and methods. Try to answer to the proposed questions in the comments.

Try to declare a non-static field of type Time in the struct type Time. Why is this illegal? Why is it legal for a class to have a non-static field of the same type as the class? Can you declare a static field noon of type Time in the struct type? Why?
Make the minutes field of struct type Time public (and not readonly) instead of private readonly. Then execute this code:

```
Time t1 = new Time(9, 30);
Time t2 = t1;
t1.minutes = 100;
Console.WriteLine("t1={0} and t2={1}", t1, t2);
```

What result do you get? Why? What result do you get if you change Time to be a class instead of a struct type? Why?

Exercise 07

Create a class type called Person. Populate the Person class with the following properties to store the following information: First name, Last name, Email address, Date of birth.

Deadline for evaluation: 28-11-2016

Add constructors that accept the following parameter lists:

- All four parameters
- First, Last, Email
- First, Last, Date of birth

Add read-only properties that return the following computed information:

- `Adult` - whether or not the person is over 18
- `SunSign` - the traditional western sun sign of this person
- `ChineseSign` - the chinese astrological sign (animal) of this person
- `Birthday` - whether or not today is the person's birthday
- `ScreenName` - a default screen name that you might see being offered to a first time user of AOL or Yahoo (e.g. John Doe born on Feburary 25th, 1980 might be jdoe225 or johndoe022580)

For date related feature you should use the System.DateTime structure.

Take some time and think carefully about your Person class and think carefully about things that might go wrong when using it. For example, you should not be able to create people that have not been born. In addition, a valid email address is of the form joeschmoe@mydomain.com. Find exception classes in the .NET class hierarchy that would match the problems you might find. In many cases, you will not find a matching class. Implement exception classes that convey the appropriate error. At very least you should implement the following exceptions:

- Date of birth is in the future
- Date of birth is too far in the past (we're only dealing with people who are alive).
- Invalid email address

Exercise 08

Create an interface called Payable. This is the interface that will be used by the accounting department's software (which you are not responsible for authoring) for all things that they need to write checks for. The Payable interface should contain three functions:

- Retrieve amount due
- Add to amount due
- Payment address

Derive an Employee class from the Person class (see previous exercise). The Employee class should add the following properties:

- Salary
- Mailing address

In addition, the Employee class should implement the Payable interface. The implementation of the functions specified in the Payable interface should make sense. In other words, the payment address should be the mailing address of the employee. In order to make this work right, you will need to allocate an internally protected state variable that keeps track of the amount of money due. This state variable will obviously be modified by the functions defined in the interface. You can of course, try to do this with a property and add this property to the Payable interface.

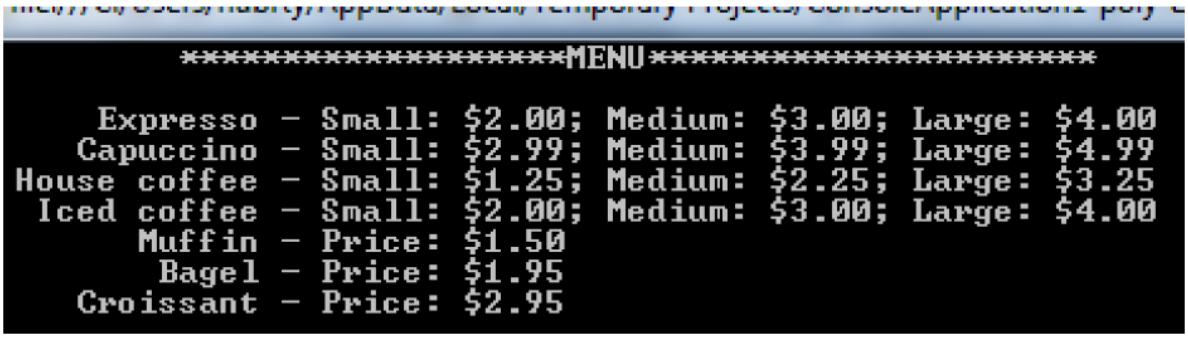
Deadline for evaluation: 28-11-2016

Exercise 09

Write a base class named: **MenuItem**. It should have a name. It should have a method called **printToString**, to print the corresponding item into a string; this method should be virtual so that we can implement polymorphism. It should have a set method or a constructor, or both, related member variable name.

Write 2 derived classes: **Beverage** and **Snack**. **Beverage** should have 3 prices: **small**, **medium** and **large** sizes. It should have a method named **printToString** overriding his base class method. Write a constructor or a set method to handle the prices of the three sizes. Snack has 1 member variable: price. It also has **printToString** method overriding his base class method. It should have a constructor, or a set method to handle the price of a snack object.

In the test methods, declare an array of 7 objects of **MenuItem**. Then use 4 elements to store information of the drinks as shown in the picture below. The last 3 elements will have information of 3 snack items. The test should check that the correct methods are being called properly.



Exercise 10

Create a base class and name this class as **Employee**. It should have 1 member variable: **empName**. Add a constructor so that the employee name can be saved for each object. Implement a virtual method called **calcPaidCheck**. Create a derived class named **HourlyWorker** with an override method **calcPaidCheck**. Paid amount is computed as the “hourly rate” of that employee multiplied by the “hours worked”. Create a derived class named **SalaryWorker** with an override method **CalcPaidCheck** and the paid amount with be annual-salary/12. Test these classes by creating an array of **Employee**, and test with 2 hourly worker employees and 2 salary employees.