# Homework: Common Type System

This document defines the homework assignments from the ["OOP" Course @ Software University](https://softuni.bg/courses/oop/). Please submit as homework a single zip / rar / 7z archive holding the solutions (source code) of all below described problems. The solutions should be written in C#.

## Value and Reference Types

Describe the differences between value and reference types. Answer the following questions:

* Which .NET data types are **value types**? Which types are **reference types**?
  + Value types are: int, bool, long, decimal, char, float, double, byte, sbyte, short, uint, ulong, ushort, struct, enum.
  + Reference types are: class, delegate, dynamic, object, interface, string, interpolate strings.
* How do **value** and **reference** types differ when **assigned to a variable**?
  + Value types: when assigned a small amount of memory is allocated in the stuck and is copied to the variable;
  + Reference types: when assigned a pointer is created on the stuck and the object is stored in the heap.
* How do **value** and **reference** types behave when **passed as argument to a method**?
  + Value types: when passed a smaller, compared to reference types, amount of memory is allocated to preserve the values.
  + Reference types: when passed the value is stored in the heap – it allocates more memory than value types.
* Where **in memory** are **value types allocated**? Where are **reference types allocated**?
  + Value types: in the stuck.
  + Reference types: in the heap;
* What is data **mutation**? Can **value** types be **mutated**? Can **reference** types be **mutated**?
  + A data mutation is when the memory dynamically changes. While value types have fix sized, reference types can dynamically grow.
* What is the difference between **mutating** and **setting** (assigning) a variable? Give examples.
  + When assigning a variable a fixed size of the memory (say 800 kb) is allocated to preserve values. If values size reaches the end size, the memory does not expand automatically and if another value is passed an out of memory exception will be triggered. Contrary, a mutating variable will dynamically expand when the maximal memory size is reached. Thus out of memory exception will depend solely on the size of the hard disk.
* What are the **out** and **ref** keywords used for? In what do they differ?
  + Ref and Out are used to pass arguments within a method or function. Both indicate that an argument is passed by reference (by default arguments are passed to a method by value).  The main difference is that while Out arguments do not need to be initialized before being passed (ex. int a;), the Ref do (ex. int a = 5). Also, passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter. And finally declaring a parameter to an Out method is useful when multiple values need to be returned from a function or method.

Submit your answers as a **.txt** file as part of your homework.

## Customer

Define a class Customer, which contains data about a customer – **first name**, **middle** **name** and **last name**, **ID** (EGN), **permanent address**, **mobile phone**, **e-mail**, list of **payments** and **customer type**.

* Define a class **Payment** which holds a **product name** and **price**.
* Define an enumeration for the customer type, holding the following types of customers: **One-time** , **Regular**, **Golden**, **Diamond**.
* Override the standard methods, inherited by **System.Object**: **Equals()**, **ToString()**, **GetHashCode()** and operators **==** and **!=**.
* Implement the **ICloneable** interface. The **Clone()** method should make a deep copy of all object fields into a **new** object of type **Customer**.
* Implement the **IComparable<Customer>** interface to compare customers by **full name** (as first criteria, in lexicographic order) and by **ID** (as second criteria, in ascending order).

## String Disperser

Define a class **StringDisperser**.

* The constructor should take several strings as arguments.
* Override the standard methods, inherited by **System.Object**: **Equals()**, **ToString()**, **GetHashCode()** and operators **==** and **!=**.
* Implement the **ICloneable** interface. The **Clone()** method should make a deep copy of all object fields into a **new** object of type **StringDisperser**.
* Implement the **IComparable<StringDisperser>** interface to compare string dispersers by their **total string value** lexicographically.
* Implement the **IEnumerable** interface to allow **foreach** on objects of type **StringDisperser**. The items returned should be the **characters** of each string.

|  |  |
| --- | --- |
| **Input** | **Output** |
| StringDisperser stringDisperser = new StringDisperser("gosho", "pesho", "tanio");  foreach (var ch in stringDisperser)  {  Console.Write(ch + " ");  } | g o s h o p e s h o t a n i o |

## \*\* Custom Tree

Define the data structure **binary search tree**. It should support the following operations:

* **Adding** a new element
* **Searching** elements
* **Deleting** elements

It is not necessary to keep the tree balanced. Implement the standard methods from **System.Object** – **ToString()**, **Equals()**, **GetHashCode()** and the operators for comparison **==** and **!=**. Implement **IEnumerable<T>** to traverse the tree.

**Add()** and implement the **ICloneable** interface for deep copy of the tree.

**Tip**: Define two separate types – structure **BinarySearchTree** (for the tree) and class **TreeNode** (for the tree elements).