Aula Prática 6

1. Seja a classe `Logger`, responsável por enviar para determinado repositório mensagens de *log*:

```
using System;

public sealed class Logger {
  int currLevel;
  //...
  public void Log(int level, string msg) {
     if (level >= currLevel)
               dispatch(msg);
  }
// faz log da mensagem msg
 protected void dispatch(string msg) {
     Console.WriteLine(msg);
  //…
}
  public int Level {
        set { currLevel = value; }
        get { return currLevel; }

        }
}
```

```
public class Program{

 public static void
 UseLogger( Logger log, long val1, string
val2, int errNumber){
     string msg;
     msg = String.Format("Ocorreu o erro {0}
         com os valores({1}, {2})",errNumber,
                     val1, val2);
     log.Log(5 , msg);
     }

 public static void Main(){
       Logger l=new Logger();
       l.Level=2;
       UseLogger(l,2,"xpto",2);
       l.Level=6;
       UseLogger(l,6,"xpto",6);
  }
}
```

A propriedade `level` permite obter/alterar o nível corrente do *logger*. Invocações do método `Log` que especificam um nível (`level`) inferior ao nível corrente são descartadas, não se fazendo nenhum registo das mesmas.
No método `UseLogger`, apresentado acima à direita, a construção da *string* `msg` é trabalho perdido se `log` tiver um nível corrente superior a 5, uma vez que a mensagem é descartada. De forma a evitar esta situação criou-se o tipo `delegate string Formatter()` com o objectivo de invocar o código de construção da *string* apenas quando for estritamente necessário. Modifique o método `Log` para passar a receber uma instância do tipo `Formatter` e altere o método `UseLogger` para invocar a nova versão do método `Log`.

2) Considere a classe `Sorter` e a sua utilização na classe `App`:

```
class Sorter{
 static void Sort<T>(IList<T> l,
                     IComparer<T> cmp){
  for(int i = 0; i < l.Count-1; i++){
   for(int j = i+1; j < l.Count; j++){
     if(cmp.Compare(l[i], l[j]) > 0){
       T aux = l[j];
       l[j] = l[i];
       l[i] = aux;
    }
   }
  }
 }
}
```

```
class App{
  void printElements(short[] a) {…}
   static void Main(){
     short[] dummy = {3,4,6,2,1,8,5,9,6,7,0};
     printElements(dummy);
     Sorter.Sort(dummy, new Int16Comparer());
     printElements(dummy);
   }
class Int16Comparer:IComparer<short>{
  public int Compare(short n1, short n2){
    return (int) n1 – n2;
  }
 }
}
```

Faça uma nova implementação das classes `Sorter` e `App` mantendo o comportamento apresentado, mas substituindo a utilização da interface `IComparer<T>` pelo *delegate* `int Comparison<T>(T x, T y)`.

3) Seja o *delegate* `public delegate Action<T> (T obj)` e o método `public static ForEach<T> (T[] a, <T> action)` da classe `Array` que executa `action` por cada elemento do array `a`. `voidvoidAction`Tirando partido do método `ForEach`, implemente o método genérico `Greatest` da classe ArrayUtils, parametrizado pelo tipo comparável `T`, que recebe como parâmetro um *array* de `T`, e retorna o maior elemento presente no *array*.