Aula Prática 7

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{
  static 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`. 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*.

4) Considere a classe ConvertingPoints

```
//...
public struct Point{ //…
}

public class Program{

public static Point[] ConvertingPoints( int[] array ){
    //...
}

public static void Main(){
       int[] values={1,2,3,4,5,6,7,8,9};
       Point[] newValues=ConvertingPoints(values );
       foreach(Point j in newValues)
                  Console.WriteLine(j);
}
}
```

Output:

```
(1,1)
(1,2)
(1,3)
(1,4)
(1,5)
(1,6)
(1,7)
(1,8)
(1,9)
```

Seja o delegate public delegate `TOutput Converter<TInput,TOutput> (TInput input)` e o método `public static U[] ConvertAll<T,U>(T[] array, Converter<T,U> converter)`. Tirando partido deste método, implemente o método ConvertingPoints, acrescentando a Point o necessário, para que produza o output pretendido.

5) Assuma que se pretende obter uma implementação da abordagem map/reduce para processamento de dados. Neste exercício queremos apenas implementar a operação mapper, que recebe uma colecção de itens enumerável, aos quais queremos aplicar a operação map. O resultado da operação map a cada item retorna uma chave e um valor. Implemente o método Mapper, utilizando o que aprendeu a respeito de delegates e extension methods. Implemente o método CountWords, em que cada item é uma linha de texto e em que o output são pares palavra e número de ocorrências da palavra na linha em questão.

```
static class Program {

public static IEnumerable<KeyValuePair<K, U>> Mapper<T, K, U>(
                     this IEnumerable<T> input,
                     Func<T, IEnumerable<KeyValuePair<K, U>>> map){
       //. . .
}

public static IEnumerable<KeyValuePair<string, int>> CountWords(IEnumerable<string> lines) {
       //…
}



public static void Main() {
  string[] lines = {"ola mundo ola ave", "ave ola ave"};
  foreach (KeyValuePair<string, int> p in CountWords(lines))
       Console.WriteLine(p);
  }
}
```

Output:

```
[ola, 2]
[mundo, 1]
[ave, 1]
[ave, 2]
[ola, 1]
```

6) Implemente agora os métodos Joiner e Reducer. Dado uma colecção de pares chave valor, o método Joiner deverá retornar uma colecção de grupos em que cada grupo agrupa os itens com a mesma chave. O método Reducer recebe uma colecção de grupos e, dada a função reduce, retorna para cada grupo um par chave e valor, em que o valor é determinado pela função reduce a partir dos itens no grupo com a essa chave.
Utilize os métodos Joiner e Reducer para extender o método CountWords por forma a que retorne agora o número de ocorrências de cada palavra na colecção de linhas inicial.

```
static class Program {

public static IEnumerable<KeyValuePair<K, U>> Mapper<T, K, U>(
                     this IEnumerable<T> input,
                     Func<T, IEnumerable<KeyValuePair<K, U>>> map){
       //. . .
}

public static IEnumerable<IGrouping<K, KeyValuePair<K,U>>> Joiner<K, U>(
                     this IEnumerable<KeyValuePair<K, U>> input) where K:IComparable<K> {

              //…
       }

public static IEnumerable<KeyValuePair<K, V>> Reducer<K, U, V>(
                           this IEnumerable<IGrouping<K, KeyValuePair<K,U>>> groups,
                           Func<IGrouping<K, KeyValuePair<K,U>>, KeyValuePair<K, V>> reduce) {
//…
}

public static IEnumerable<KeyValuePair<string, int>> CountWords(IEnumerable<string> lines) {
       //…
}



public static void Main() {
  string[] lines = {"ola mundo ola ave", "ave ola ave"};
  foreach (KeyValuePair<string, int> p in CountWords(lines))
       Console.WriteLine(p);
  }
}
```