Appello TAP del 12/09/2016

Scrivere nome, cognome e matricola sul foglio protocollo, indicando anche se avete nel piano di studi TAP da 6 CFU (quello attuale) o da 8 CFU (quello "vecchio"). Avete a disposizione due ore.

Esercizio 1 (7+7 = 14 punti)

1. Implementare il seguente metodo statico

```
public static bool MoveNext_OutCurr<T>(IEnumerator<T>[] enumerators, out T[] currents)
```

che incapsula l'invocazione di `MoveNext` e `Current` su tutti gli enumeratori in `enumerators` in un unico passaggio, assumendo che tutti gli elementi di `enumerators` siano fra loro distinti.

Pertanto, `MoveNext_OutCurr`

- restituisce `true` se e solo se al momento della chiamata tutti gli enumeratori in `enumerators` sono posizionati *non* a fine sequenza;
- quando restituisce `true`
  - sposta in avanti di una posizione tutti gli enumeratori per cui questo è possibile.
  - inizializza `currents` con gli elementi correnti (dopo l'aggiornamento della posizione).
- non è specificato il comportamento quando restituisce `false`.

Ad esempio, se `enumerators` contiene *nuovi* enumeratori alle tre sequenze di interi $[10, 100, 1000]$, $[20, 200]$ e $[30, 300, 3000]$, quindi posizionati prima dell'inizio delle rispettive sequenze

- la prima chiamata di `MoveNext_OutCurr` restituirà `true` e inizializzerà `currents` con $[10, 20, 30]$;
- la seconda chiamata restituirà `true` e inizializzerà `currents` con $[100, 200, 300]$;
- la terza chiamata restituirà `false` e non è specificato come inizializzerà `currents`, né se sposterà o meno la posizione degli enumeratori (ad esempio ne può spostare solo alcuni).

2. Si assuma data la seguente interfaccia

```
public interface IDoable<TSource, TRes>
{
    TRes DoIt(params TSource[] args);
}
```

di cui possibili implementazioni (impiegate anche negli esempi seguenti) sono, ad esempio:

```
internal class MyDoableInt
                  : IDoable<int, int>{
   private int seed;
   public int DoIt(params int[] args){
      var result = seed;
      foreach (var i in args)
         result += i;
      return result;
    }
   internal MyDoableInt(int value){
       seed = value;
   }
}
```

```
internal class MyDoableString
                  : IDoable<string, string>{
   private string seed;
   public string DoIt(params string[] args){
       var result = seed;
       foreach (var s in args)
          result += s;
       return result;
   }
  internal MyDoableString(string value){
    seed = value;
  }
}
```

Scrivere l'extension-method `AggregateDo` che, presi una sequenza `source` di elementi di tipo `IDoable<TSource, TRes>` e un numero arbitrario di argomenti di tipo `IEnumerable<TSource>[]`, restituisce la sequenza dei valori di tipo `TRes` ottenuti invocando il metodo `DoIt` su ciascun elemento di `source`, passando gli elementi correnti degli altri argomenti come parametri a `DoIt`, fintanto che tutte le sequenze ne contengono.
Si noti che tutte le sequenze, sia in input che in output, possono essere infinite.

Il metodo dovrà sollevare l'eccezione `ArgumentNullException` se `source` o uno degli altri argomenti sono nulli.

Per esempio, il seguente frammento di codice

```
var x = new[] {1, 2, 3, 4}.Select(i => new MyDoableInt(i))
                          .AggregateDo(new[] {10, 20, 30}, new[] {100, 200});
foreach (var i in x) Console.Write("{0}, ", i);
```

stampa `111, 222, .`

Esercizio 2 (3+3+4 = 10 punti)

Implementare, usando NUnit ed eventualmente Moq, i seguenti test relativi al metodo `AggregateDo`, dell'esercizio precedente.

- Input della chiamata sotto test: `source` deve essere la sequenza

```
MyDoableString("pippo"), MyDoableString("topolino"), MyDoableString("basettoni")
```

  senza parametri aggiuntivi.

  Output atteso: la sequenza `"pippo"`, `"topolino"`, `"basettoni"`.

- Input della chiamata sotto test: `source` deve essere una sequenza infinita e ci devono essere 3 parametri aggiuntivi di cui il secondo, e solo esso, nullo.

  Output atteso: deve essere sollevata un'eccezione di tipo `ArgumentNullException`.

- Input della chiamata sotto test: `source` deve essere la sequenza infinita

```
MyDoableInt(10), MyDoableInt(20), MyDoableInt(30), MyDoableInt(40), MyDoableInt(50),...
```

  gli argomenti aggiuntivi devono essere 3, rispettivamente la sequenza infinita che restituisce sempre 1, quella che restituisce sempre 2 e quella che restituisce sempre 3.

  Il test deve essere parametrico con un parametro intero `howMany` e verificare che i primi `howMany` del risultato della chiamata siano i primi `howMany` elementi della sequenza $16, 26, 36, 46, \ldots$

Esercizio 3 ([-6, 6] punti)

Rispondere alle seguenti domande a risposta multipla. Fate attenzione che

- alcune [combinazioni di] risposte sono così sbagliate da portare *punteggio negativo*;
- alcune domande hanno più risposte corrette e per totalizzare il punteggio pieno dovete selezionarle *tutte*.

1. Data l'espressione LINQ

```
Enumerable.Range(100, 50).Where(i => i >= 110)
```

   quali delle seguenti affermazioni sono vere?

   - ☐ Il tipo dell'espressione è `IEnumerable<int>`
   - ☐ Il tipo dell'espressione è `int[]`
   - ☐ Il tipo dell'espressione è `IQueryable<int>`
   - ☐ Il tipo dell'espressione è `int`
   - ☐ L'espressione non è sintatticamente corretta perché il metodo `Where` si può invocare solo su un'espressione di tipo `DbSet<T>`
   - ☐ L'espressione può essere assegnata ad una variabile di tipo `IQueryable<int>`
   - ☐ Per poter assegnare l'espressione ad una variabile di tipo `IQueryable<int>` bisogna prima cambiarle il tipo, ad esempio usando il metodo `AsQueryable()`
   - ☐ L'espressione può essere assegnata ad una variabile di tipo `int[]`
   - ☐ All'espressione può essere assegnata un valore di tipo `int[]`
   - ☐ Nessuna delle precedenti affermazioni è vera

2. Subito dopo le dichiarazioni

```
var count = 0;
var x = Enumerable.Range(100, 50).Where(i => ++count>=0 && i >= 110);
```

quali delle seguenti affermazioni sono vere?

- ☐ `count` è strettamente positivo
- ☐ `x` è `null`
- ☐ `x.GetEnumerator()` è corretta
- ☐ `x.GetEnumerator().Current` è corretta e restituisce 110
- ☐ `x.GetEnumerator().Current` non è corretta, ma lo sarebbe se fosse preceduta dalla chiamata `x.GetEnumerator().MoveNext()` e in tal caso restituirebbe 110
- ☐ `x.Current` è corretta
- ☐ `x.Current` non è corretta perché siamo posizionati prima dell'inizio della sequenza, ma lo sarebbe se fosse preceduta dalla chiamata `x.MoveNext()` e in tal caso restituirebbe 110
- ☐ usare il costrutto `foreach` è l'unico modo possibile per poter accedere agli interi che fanno parte della sequenza rappresentata da `x`
- ☐ `x[0] == 110` si valuta in `true`

3. Quali delle seguenti affermazioni sono vere?

- ☐ la dichiarazione `Func<int,int> aaa = zzz => 3+zzz;` è corretta;
- ☐ il frammento di codice

```
Func<int,int> aaa = zzz => 3+zzz;
aaa = zzz => 7+zzz;
```

è corretto

- ☐ la dichiarazione `var bbb = zzz => 3+zzz;` è corretta;
- ☐ in uno scope in cui sia visibile la dichiarazione `delegate int MyFun(int a);`, anche la dichiarazione `MyFun ccc = zzz => 3 + zzz;` è corretta;
- ☐ la dichiarazione `Predicate<int> ddd = zzz => zzz == 0;` è corretta;
- ☐ la dichiarazione `Func<int,bool> eee = zzz => zzz == 0;` è corretta;
- ☐ il frammento di codice

```
Predicate<int> ddd = x => x != 666;
Func<int,bool> eee = y => y <= 0;
eee = ddd;
```

è corretto

- ☐ la dichiarazione

```
Func<Func<int, int>, Func<int, int>, Func<int, int>>
     comp = (first, second) => w => second(first(w));
```

è corretta