Exercises Collections

1. A sequence generator

a. Create a class Sequence , and let it implement IEnumerable<int>
   i. Create an appropriate implementation of IEnumerator<int> and return an instance where appropriate

b. Add a way of parameterizing the Sequience, either by properties, methods or constructors, to allow setting
   i. Sequence start
   ii. Sequence end or count (if any)
   iii. Sequence skip
      - 1 3 5 7 has start 1, skip 2 and count 4
      - Perhaps one could imagine that 1 1 2 3 5 8 could be a sequence?

2. A Random numbers Enumerable

a. Create a class RandomNumbers, and let it implement IEnumerable<int>
   i. Create an appropriate implementation of IEnumerator<int> and return an instance where appropriate
   ii. Create properties and/or constructors to set – discuss your approach in the group
      1. Seed
      2. Max value
      3. Min value

b. Create a class RandomNNumbers parameterized with the number of random numbers it generates

3. A Sorted List: SortedList<T>

A sorted list maintains the proper ordering of elements whenever elements are added or removed.

a. In order to uphold the ordering, you must specify a constraint such that only elements that implement IComparable can be inserted.

b. Your data structure should implement ICollection<T> functionality.
   i. (*optional challenge*) You should supply an indexer with read-only capabilities. That is, users must not be able to insert elements into a particular index position (as this may break the ordering), but they should be allowed to ask what element is in a particular index position.

c. You should supply three enumerators:
   i. A forward enumerator (this should be the default)
   ii. A backward enumerator (clients have to ask for this by calling myList.GetElementsReversed()

iii. An enumerator that accepts a predicate that can be used to filter the elements. Only the elements that fulfil the predicate should be enumerated – in forward order. myList.GetElements(Predicate<>)

d. Test using a class of your own design.

4. Standard Query Operators: Numbers (LINQ)

Given a list of random numbers:

```
List<int> numbers = new List<int>();
Random r = new Random();
int randomNum = 0;
for (int i = 1; i < 20; i++)
{
  randomNum = r.Next(0, 100); //random number between 0 and 100
  numbers.Add(randomNum);
}
```

Use the appropriate query operators (inspect the API), to accomplish the following:

a. Find all elements that are multiples of the value of an outer variable.

b. Find all elements between MAX and MIN as specified by two outer variables (e.g., all numbers between 20 and 40).

c. Return the greatest number between MAX and MIN (e.g, the number 38 if MIN=20, MAX=40)

d. Multiply all elements with a given value as specified by an outer variable.

e. Order the elements in descending order.

f. Combine 2, 4, and 5 into one expression.

g. (*optional challenge*) Use the method Enumerable.Range to create a list of random numbers in as few lines(statements) as possible (two is possible, one is *doable*). Remember **Random** must only be initialized once!

5. More complex queries

Her er en person-klasse:

```
public class Person
{
    public string Name { get; set; }
    public double Weight { get; set; }
    public int Age { get; set; }
}
```

Og her er nogle personer:

```
List<Person> people = new List<Person>()
{
    new Person() { Name = "Ib", Weight = 89.6, Age = 27 },
    new Person() { Name = "Kaj", Weight = 65.7, Age = 17 },
    new Person() { Name = "Ole", Weight = 77, Age = 7 },
    new Person() { Name = "Anders", Weight = 72, Age = 40 },
    new Person() { Name = "Børge", Weight = 88.8, Age = 13 }
};
```

Using LINQ

a. Order the people-list by weight
b. Order the people-list by name in reverse
c. Get a list of the names (ONLY names) of all people in the list with a name containing an ‘a’ or ‘A’, and are older than 10 years.
d. Find the name of the teenager with the longest name
e. (*optional challenge*) Find the weight of the teenager with the longest name

6. Query motorvehicles

Given this Vehicle hierarchy:

```
abstract class MotorVehicle
{
    protected Fuel _fuel;

    public string Make { get; set; } //VW, Audi, Skoda...
    public string Model { get; set; } //Golf, Polo, A3, Fabia, etc.
    public int Year { get; set; }
    public decimal Price { get; set; }

    public virtual Fuel Fuel
    {
        get { return _fuel; }
        set { _fuel = value; }
    }
}

class Bus : MotorVehicle
{
    public Bus()
    {
        _fuel = Fuel.Diesel;
    }

    public int NumSeats { get; set; }

    public override Fuel Fuel
    {
        set { } //do nothing - only diesel is allowed
    }
}

class Car : MotorVehicle
{
    public bool HasSunRoof { get; set; }
}
```

And some pre-baked vehicles:

```
public static void TestVehicles()
{
```

```
List<MotorVehicle> vehicles = new List<MotorVehicle>()
{
    new Car() { Make = "Opel",   Model = "Zafira", Year = 2002,
        Fuel = Fuel.Octane95, Price = 112000 },
    new Car() { Make = "Ford",   Model = "Fiesta", Year = 1994,
        Fuel = Fuel.Octane92, HasSunRoof = true, Price = 72000 },
    new Car() { Make = "Mazda",  Model = "6",      Year = 2007,
        Fuel = Fuel.Octane95, Price = 200000 },
    new Car() { Make = "Opel",   Model = "Astra",  Year = 1995,
        Fuel = Fuel.Octane92, HasSunRoof = true, Price = 45000 },
    new Car() { Make = "Opel",   Model = "Astra",  Year = 1997,
        Fuel = Fuel.Diesel, Price = 52000 },
    new Car() { Make = "Opel",   Model = "Zafira", Year = 2001,
        Fuel = Fuel.Diesel, Price = 137000 },
    new Car() { Make = "Ford",   Model = "Focus",  Year = 2007,
        Fuel = Fuel.Octane92, HasSunRoof = true, Price = 199999 },
    new Car() { Make = "Opel",   Model = "Astra",  Year = 1996,
        Fuel = Fuel.Diesel, Price = 29000 },
    new Bus() { Make = "Scania", Model = "Buzz",   Year = 1999,
        Price = 275000, NumSeats = 52},
    new Bus() { Make = "Scania", Model = "Fuzz",   Year = 2000,
        Price = 225000, NumSeats = 12}
};
//...
```

a. Find the average price of all vehicles.

b. Find the average number of seats for busses.

c. Find the number of cars that have a sun roof.

d. Group vehicles by make

e. Find all octane vehicles (Octane 92 or 95) that cost between a specified and maximum price. Order the result by make, model, and price.

f. Find all veteran vehicles, i.e., vehicles that are more than 25 years old. Project the resulting elements into an anonymous type with field “Model_Make” that is a concatenation of the vehicle’s make and model, and a “YearsOld” field that tells how old the car is.