

Seminar 11

week 11 (7-11 December 2020)

A. Discussion of some possible issues regarding concurrency in your current Java project:

1. Concurrent HashMap or synchronized methods
2. Unique key generation (sync, AtomicInteger)
3. sync heap {read ... write}

B. Discuss LINQ from C# using the following examples.

Please note that the following notes and examples are taken from some LINQ tutorials available free on the web.

Before LINQ:

```
1.int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };
2.int i = 0;
3.
4.// Display numbers larger than 5
5.while (i < numbers.GetLength(0))
6.{
7.if (numbers[i] > 5)
8.Console.WriteLine(numbers[i]);

9.++i;
10.}
```

With LINQ, data sources could now be queried in an **SQL-like syntax**, as shown in the rewritten code snippet below:

```
int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };

var res = from n in numbers
          where n > 5
          select n;

foreach (int n in res)
    Console.WriteLine(n);
```

You may be interested in querying data from various sources, such as arrays, dictionaries, xml and entities created from entity framework. But instead of having to use a different API for each data source, LINQ provides a consistent and uniform programming model to work with all supported data sources.

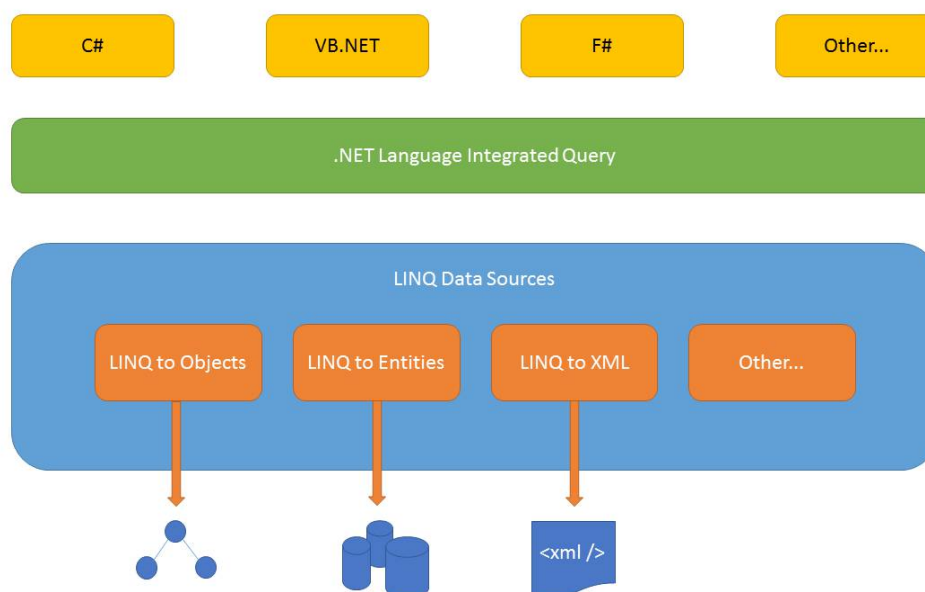
Some of the most used LINQ data sources, which are all part the .NET framework, are:

- LINQ to Objects:** for in-memory collections based on *IEnumerable*, such as *Dictionary* and *List*.
- LINQ to Entities:** for Entity Framework on object context.

•**LINQ to XML**: for in-memory XML documents.

As long as you include the namespace `System.Linq` in your code, you are good to go with all of the above data sources.

You can use LINQ from all the popular .NET languages (C#, VB.NET and F#) as illustrated in the architecture overview below.



Example: In a collection of random numbers we want to retrieve only values greater than 50, sort them in ascending order, and lastly cast the result into strings:

LINQ offers two ways to write queries:

1) with SQL-like syntax called *Query expressions*

```
int[] numbers = { 7, 53, 45, 99 };
```

```
var res = from n in numbers
where n > 50
orderby n
select n.ToString();
```

2) a method like approach called *Lambda expressions*

```
int[] numbers = { 7, 53, 45, 99 };
```

```
var res = numbers.Where(n => n > 50)
                .OrderBy(n => n)
                .Select(n => n.ToString());
```

LINQ queries can execute in two different ways: deferred and immediate.

With deferred execution, the resulting sequence of a LINQ query is not generated until it is required. The following query does not actually execute until `Max()` is called, and a final result is required:

```
int[] numbers = { 1, 2, 3, 4, 5 };
var result = numbers.Where(n => n >= 2 && n <= 4);
Console.WriteLine(result.Max()); // <- query executes at this point

// Output:
//4
```

Example:

Instead of LINQ having to first iterate over all three chains (first `Select()`, then `Where()` and lastly `ToList()`) the result is not generated until it meets `ToList()`:

```
string[] words = { "one", "two", "three" };
var result = words.Select((w, i) => new { Index = i, Value = w }).Where(w =>
w.Value.Length == 3).ToList();

Debug.WriteLine("Prints index for words that have a string length of 3:");
foreach(var word in result)
    Debug.WriteLine (word.Index.ToString());

// Output:
// Prints index for words that have a string length of 3:
// 0
// 1
```

Adding items to an existing query is another benefit of deferred execution. The following example shows the concept:

```
List vegetables = new List { "Carrot", "Selleri" };
var result = from v in vegetables select v;
Debug.WriteLine("Elements in vegetables array (before add): " + result.Count());
vegetables.Add("Broccoli");
Debug.WriteLine("Elements in vegetables array (after add): " + result.Count());
// Output:
// Elements in vegetables array (before add): 2
// Elements in vegetables array (after add): 3
```

Deferred execution makes it useful to combine or extend queries. Have a look at this example, which creates a base query and then extends it into two new separate queries:

```
int[] numbers = { 1, 5, 10, 18, 23 };
var baseQuery = from n in numbers select n;
var oddQuery = from b in baseQuery where b % 2 == 1 select b;

Debug.WriteLine("Sum of odd numbers: " + oddQuery.Sum()); // <- query executes at this point

var evenQuery = from b in baseQuery where b % 2 == 0 select b;
Debug.WriteLine("Sum of even numbers: " + evenQuery.Sum()); // <- query executes at this point

// Output:
// Sum of odd numbers: 29
// Sum of even numbers: 28
```

LINQ Operators(some of them)

Aggregate: Performs a specified operation to each element in a collection, while carrying the result forward.

```
var numbers = new int[] { 1, 2, 3, 4, 5 };

var result = numbers.Aggregate((a, b) => a * b);

Debug.WriteLine("Aggregated numbers by multiplication:");
Debug.WriteLine(result);

//Output:
//Aggregated numbers by multiplication:
//120
```

Any: Checks if any elements in a collection satisfies a specified condition.

```
string[] names = { "Bob", "Ned", "Amy", "Bill" };

var result = names.Any(n => n.StartsWith("B"));

Debug.WriteLine("Does any of the names start with the letter 'B':");
Debug.WriteLine(result);
```

ElementAtOrDefault: Retrieves element from a collection at specified (zero-based) index, but gets default value if out-of-range.

```
string[] colors = { "Red", "Green", "Blue" };

var resultIndex1 = colors.ElementAtOrDefault(1);

var resultIndex10 = colors.ElementAtOrDefault(10);

Debug.WriteLine("Element at index 1 in the array contains:");
Debug.WriteLine(resultIndex1);

Debug.WriteLine("Element at index 10 in the array does not exist:");
Debug.WriteLine(resultIndex10 == null);
```

Output:

```
//Element at index 1 in the array contains:
//Green
//Element at index 10 in the array does not exist:
//True
```

SelectMany: Flattens collections into a single collection (similar to cross join in SQL).

```
string[] fruits = { "Grape", "Orange", "Apple" };
int[] amounts = { 1, 2, 3 };

var result = fruits.SelectMany(f => amounts, (f, a) => new
{
    Fruit = f,
    Amount = a
});

Debug.WriteLine("Selecting all values from each array, and mixing them:");
foreach (var o in result)
    Debug.WriteLine(o.Fruit + ", " + o.Amount);
}
```

Output:

Selecting all values from each array, and mixing them:

Grape, 1
Grape, 2
Grape, 3
Orange, 1
Orange, 2
Orange, 3
Apple, 1
Apple, 2
Apple, 3

ToDictionary: Converts collection into a Dictionary with Key and Value.

```
class English2German
```

```
{  
    public string EnglishSalute { get; set; }  
    public string GermanSalute { get; set; }  
}
```

```
static void Sample_ToDictionary_Lambda_Simple()
```

```
{  
    English2German[] english2German =  
    {  
        new English2German { EnglishSalute = "Good morning", GermanSalute = "Guten Morgen" },  
        new English2German { EnglishSalute = "Good day", GermanSalute = "Guten Tag" },  
        new English2German { EnglishSalute = "Good evening", GermanSalute = "Guten Abend" },  
    };  
  
    var result = english2German.ToDictionary(k => k.EnglishSalute, v => v.GermanSalute);  
  
    Debug.WriteLine("Values inserted into dictionary:");  
    foreach (KeyValuePair<string, string> dic in result)  
        Debug.WriteLine(String.Format("English salute {0} is {1} in German", dic.Key, dic.Value));  
}
```

Output:

Values put into dictionary:

English salute Good morning is Guten Morgen in German
English salute Good day is Guten Tag in German
English salute Good evening is Guten Abend in German

AsEnumerable: casts a collection to IEnumerable of same type.

```
string[] names = { "John", "Suzy", "Dave" };

var result = names.AsEnumerable();

Debug.WriteLine("Iterating over IEnumerable collection:");
foreach (var name in result)
    Debug.WriteLine(name);
```

Concat: Concatenates (combines) two collections.

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 4, 5, 6 };

var result = numbers1.Concat(numbers2);

Debug.WriteLine("Concatenating numbers1 and numbers2 gives:");
foreach (int number in result)
    Debug.WriteLine(number);
}
```

Output:

Concatenating numbers1 and numbers2 gives:

1
2
3
4
5
6

Distinct: Removes duplicate elements from a collection

```
int[] numbers = { 1, 2, 2, 3, 5, 6, 6, 6, 8, 9 };

var result = (from n in numbers.Distinct()
              select n);

Debug.WriteLine("Distinct removes duplicate elements:");
foreach (int number in result)
    Debug.WriteLine(number);
}
```

Output:

Distinct removes duplicate elements:

1
2

3
5
6
8
9

Except: Removes all elements from one collection which exist in another collection.

```
int[] numbers1 = { 1, 2, 3 };  
int[] numbers2 = { 3, 4, 5 };
```

```
var result = (from n in numbers1.Except(numbers2)  
              select n);
```

```
Debug.WriteLine("Except creates a single sequence from numbers1 and removes the duplicates  
found in numbers2:");
```

```
foreach (int number in result)  
    Debug.WriteLine(number);
```

```
}
```

Output:

Except creates a single sequence from numbers1 and removes the duplicates found in numbers2:

1

2

GroupBy: Projects elements of a collection into groups by key.

```
int[] numbers = { 10, 15, 20, 25, 30, 35 };
```

```
var result = from n in numbers  
              group n by (n % 10 == 0) into groups  
              select groups;
```

```
Debug.WriteLine("GroupBy has created two groups:");
```

```
foreach (IGrouping<bool, int> group in result)
```

```
{
```

```
    if (group.Key == true)
```

```
        Debug.WriteLine("Divisible by 10");
```

```
    else
```

```
        Debug.WriteLine("Not Divisible by 10");
```

```
        foreach (int number in group)
```

```
            Debug.WriteLine(number);
```

```
    }
```

```
}
```

Output:

GroupBy has created two groups:

Divisible by 10

10

20

30
Not Divisible by 10
15
25
35

Intersect: Takes only the elements that are shared between two collections.

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 3, 4, 5 };

var result = (from n in numbers1.Intersect(numbers2)
              select n);

Debug.WriteLine("Intersect creates a single sequence with only the duplicates:");
foreach (int number in result)
    Debug.WriteLine(number);
}
```

Output:

Intersect creates a single sequence with only the duplicates:

3

Join: Joins two collections by a common key value, and is similar to inner join in SQL

```
string[] warmCountries = { "Turkey", "Italy", "Spain", "Saudi Arabia", "Etioipia" };
string[] europeanCountries = { "Denmark", "Germany", "Italy", "Portugal", "Spain" };

var result = (from w in warmCountries
              join e in europeanCountries on w equals e
              select w);
```

```
Debug.WriteLine("Joined countries which are both warm and European using Query Syntax:");
foreach (var country in result)
    Debug.WriteLine(country);
}
```

Output:

Joined countries which are both warm and European using Query Syntax:

Italy
Spain

OrderBy: Sorts a collection in ascending order.

```
class Car
{
    public string Name { get; set; }
    public int HorsePower { get; set; }
}

static void Sample_OrderBy_Linq_Objects()
{
    Car[] cars =
    {
        new Car { Name = "Super Car", HorsePower = 215 },
        new Car { Name = "Economy Car", HorsePower = 75 },
        new Car { Name = "Family Car", HorsePower = 145 },
    };

    var result = from c in cars
                 orderby c.HorsePower
                 select c;

    Debug.WriteLine("Ordered list of cars by horsepower using Query Syntax:");
    foreach (Car car in result)
        Debug.WriteLine(String.Format("{0}: {1} horses", car.Name, car.HorsePower));
}
Output:
Ordered list of cars by horsepower using Query Syntax:
Economy Car: 75 horses
Family Car: 145 horses
Super Car: 215 horses
```

Range: Generates sequence of numeric values.

```
var result = from n in Enumerable.Range(0, 10)
             select n;

Debug.WriteLine("Counting from 0 to 9:");
foreach (int number in result)
    Debug.WriteLine(number);
}
Output:
Counting from 0 to 9:
0
1
2
3
4
```

5
6
7
8
9

Repeat: Creates a collection of repeated elements, where first argument is value to repeat, and second argument is number of times to repeat.

```
string word = "Banana";
```

```
var result = from w in Enumerable.Repeat(word, 5)
              select w;
```

```
Debug.WriteLine("String is repeated 5 times:");
foreach (string str in result)
    Debug.WriteLine(str);
```

```
}
```

Output:

String is repeated 5 times:

Banana

Banana

Banana

Banana

Banana

Reverse: Reverses elements in a collection.

```
char[] characters = { 's', 'a', 'm', 'p', 'l', 'e' };
```

```
var result = (from c in characters.Reverse()
               select c);
```

```
Debug.WriteLine("Characters in reverse order:");
foreach (char character in result)
    Debug.WriteLine(character);
```

```
}
```

Output:

Characters in reverse order:

e

l

p

m

a

s

SkipWhile: Skips elements in a collection while specified condition is met.

```
string[] words = { "one", "two", "three", "four", "five", "six" };

var result = words.SkipWhile(w => w.Length == 3);

Debug.WriteLine("Skips words while the condition is met:");
foreach (string word in result)
    Debug.WriteLine(word);
}
```

Output:

Skips words while the condition is met:

three

four

five

Take: Takes specified number of elements in a collection, starting from first element.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var result = numbers.Take(5);

Debug.WriteLine("Takes the first 5 numbers only:");
foreach (int number in result)
    Debug.WriteLine(number);
}
```

Output:

Takes the first 5 numbers only:

1

2

3

4

5

ThenBy: Use after earlier sorting, to further sort a collection in ascending order.

```
string[] capitals = { "Berlin", "Paris", "Madrid", "Tokyo", "London", "Athens", "Beijing",
"Seoul" };
```

```
var result = (from c in capitals
              orderby c.Length
              select c)
              .ThenBy(c => c);
```

```
Debug.WriteLine("Ordered list of capitals, first by length and then alphabetical:");
foreach (string capital in result)
    Debug.WriteLine(capital);
```

```
}
```

Output:

Ordered list of capitals, first by length and then alphabetical:

Paris

Seoul

Tokyo

Athens

Berlin

London

Madrid

Beijing

Union: Combines two collections and removes duplicate elements.

```
int[] numbers1 = { 1, 2, 3 };
```

```
int[] numbers2 = { 3, 4, 5 };
```

```
var result = (from n in numbers1.Union(numbers2)
              select n);
```

```
Debug.WriteLine("Union creates a single sequence and eliminates the duplicates:");
```

```
foreach (int number in result)
```

```
    Debug.WriteLine(number);
```

```
}
```

Output:

Union creates a single sequence and eliminates the duplicates:

1

2

3

4

5

Zip: Processes two collections in parallel with func instance, and combines result into a new collection.

```
int[] numbers1 = { 1, 2, 3 };
```

```
int[] numbers2 = { 10, 11, 12 };
```

```
var result = numbers1.Zip(numbers2, (a, b) => (a * b));
```

```
Debug.WriteLine("Using Zip to combine two arrays into one (1*10, 2*11, 3*12):");
```

```
foreach (int number in result)
```

```
    Debug.WriteLine(number);
```

```
}
```

Output:

Using Zip to combine two arrays into one (1*10, 2*11, 3*12):

10

22

36