Advanced Development Techniques

XML format
JSON format
Datastructure-independent operations: LINQ
XLINQ

XML (w3schools.com)

```
<?xml version="1.0"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

- **Hierarchic data descriptor format**
- **XML declaration + elements/nodes + attributes**
  - Nodes: <bookstore></bookstore>... = <tag></tag>
  - Attributes: in <book> there is *category="..."*

XML

```
<book category="COOKING">
 <title lang="en">Everyday Italian</title>
 <author>Giada De Laurentiis</author>
 <year>2005</year>
 <price>30.00</price>
</book>
```

- The description of an object is done by embedded sub-nodes and attributes
- Using sub-nodes (children nodes) we can also represent hierarchy and composition – it all depends on the interpretation of the XML data

```
<?xml version="1.0"?>
<bookstore>
  <book category="COOKING"> ... </book>
  <book category="WEB"> ... </book>
</bookstore>
```

XML

- **Strict rules (Well-formed XML vs Valid XML)**
  - First row: xml declaration (optional), with charset definition:
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <?xml version="1.0" encoding="UTF-8"?>
- **All nodes can have:**
  - Text content
  - Children nodes
  - Attributes
- **A single root node is obligatory that surrounds the whole document (<bookstore> node)**
  - Must properly close all nodes (<tag></tag> or <tag />)
    - Must respect the nesting order of the nodes
    - BAD: <a>
    - GOOD: <a>
  - Upper and lowercase characters are considered different

XML + .NET

- **Three different approaches**
- **XmlReader, XmlWriter**
  - Fast, requires less memory
  - Only one-way operations
  - It’s hard to handle complex XML files
  - XML transformations are terrible (editing/exchanging nodes)
- **XmlDocument, XmlNode, XmlElement, Xml*...**
  - Slow, requires A LOT of memory (builds up the whole XML tree)
  - Modification is possible, but not necessarily easy
  - Exchanging nodes is very easy
- **XDocument, XElement, XAttribute, XDeclaration, X*...**
  - Efficient and LINQ-compatible!

XElement

- Flexible constructor (with the “params” keyword) almost any XML can be created with a single constructor call

- `var xe = new XElement("person", "Joe");`

```
<person>Joe</person>
```

- `var xe2 = new XElement("person",`
  `new XElement("name", "Joe"),`
  `new XElement("age", 25));`

```
<person>
  <name>Joe</name>
  <age>25</age>
</person>
```

XAttribute

- With constructor:

```
var xe = new XElement("person",
        new XAttribute("id", 43984),
        new XElement("name", "Joe"),
        new XElement("age", 25));
```

```
<person id="43984">
  <name>Joe</name>
  <age>25</age>
</person>
```

- Later:

```
var xe2 = new XElement("person",
        new XElement("name", "Joe"),
        new XElement("age", 25));
 xe2.SetAttributeValue("id", 43984);
```

Saving an XDocument

```
XDocument outDoc = new XDocument(
   new XElement("people",
      new XElement("person",
         new XAttribute("id", 1),
         new XElement("name", "Joe"),
         new XElement("age", 22)),
      new XElement("person",
         new XAttribute("id", 2),
         new XElement("name", "Quagmire"),
         new XElement("age", 34))));
outDoc.Save("people.xml");
```

- Load from file/url: `XDocument doc = XDocument.Load("http://users.nik.uni-obuda.hu/prog3/data/people.xml");`
- Load from string: `XDocument doc = XDocument.Parse(str);`

Simple XML processing

```
<people>
  <person id="43984">
    <name>Joe</name>
    <age>25</age>
    <phone>0618515133</phone>
  </person>
...
</people>
```

.Element(name)
.Attribute(name)

.Elements(name)
.Attributes(name)

```
string personName = xDoc.Element("people").Element("person")
      .Element("name").Value;
int id = int.Parse(xDoc.Element("people").Element("person")
      .Attribute("id").Value);
```

... OR typecasting ... (not only with attributes; also works with elements)

```
int id = (int)(xDoc.Element("people").Element("person")
      .Attribute("id"));
```

JSON

XML:

```xml
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON:

```json
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON

- **JavaScript Object Notation**
  - Hierarchic data descriptor format Just like the XML
- **In JSON**
  - No closing tag shorter
  - Has native syntax for arrays
  - There is no easy way for cyclic / recursive data structures
  - It is possible to handle errors
- **Natively supported by browsers**
  - But all modern languages have JSON support
- **Usage in .NET**
  - DataContractJsonSerializer – almost never
  - Newtonsoft.JSON / JSON.NET – even Microsoft uses this
  - Can be easily used with LINQ methods
  - Very simple usage for full serialization / deserialization
  - Can be indexed if using JObject/JArray/JToken

Simple JSON

```
Random rnd = new Random();
List<Color> colorList = new List<Color>();
for (int i = 0; i < 10000; i++)
{
    colorList.Add(new Color() {
        Red = rnd.Next(0, 256),
        Green = rnd.Next(0, 256),
        Blue = rnd.Next(0, 256) });
}

string json = JsonConvert.
    SerializeObject(colorList, Formatting.Indented);
File.WriteAllText("colors.json", json);
```

```
var colorList2 = JsonConvert.
    DeserializeObject<List<Color>>(json);
var colorList3 = JsonConvert.
    DeserializeObject<List<Color>>(json, new JsonSerializerSettings()
{
    Error = (sender, errorArgs) =>
    {
        errorMsg += errorArgs.ErrorContext.Error.Message + "\n\n";
        errorArgs.ErrorContext.Handled = true;
    }
```

LINQ (Language Integrated Queries)

- **Provides a simple syntax to handle collections independent from their source and structure**
  - Simple syntax: “operators” that perform common operations that are typically solved with loops/conditions
  - Structure-independent: array, list, XML, database …
- **LINQ To Objects, LINQ To XML, LINQ to Entities, …**
- **Parts and language elements**
  - \+ Lambda expressions
  - \+ var keyword, anonymous classes
  - \+ Extension methods
  - LINQ operators (that are actually extension methods)
  - Procedural/OOP vs DECLARATIVE
  - LINQ query syntax (from, where, select, join, orderby...)

Var keyword and anonymous classes

- **var: we want the compiler to decide the type of the variable**
  - Value assignment is obligatory in the same statement

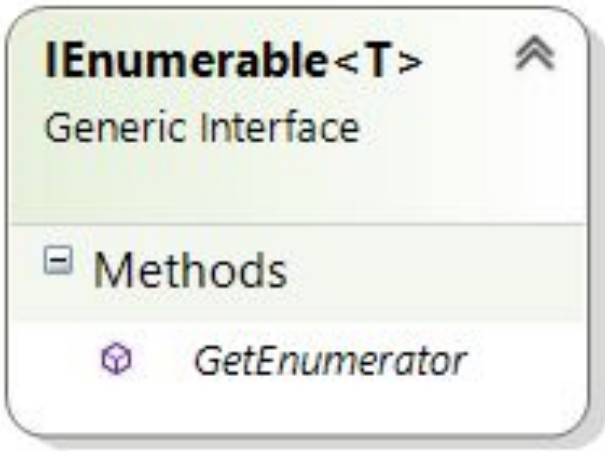
```
var x = 6;
var z = new Student();
```

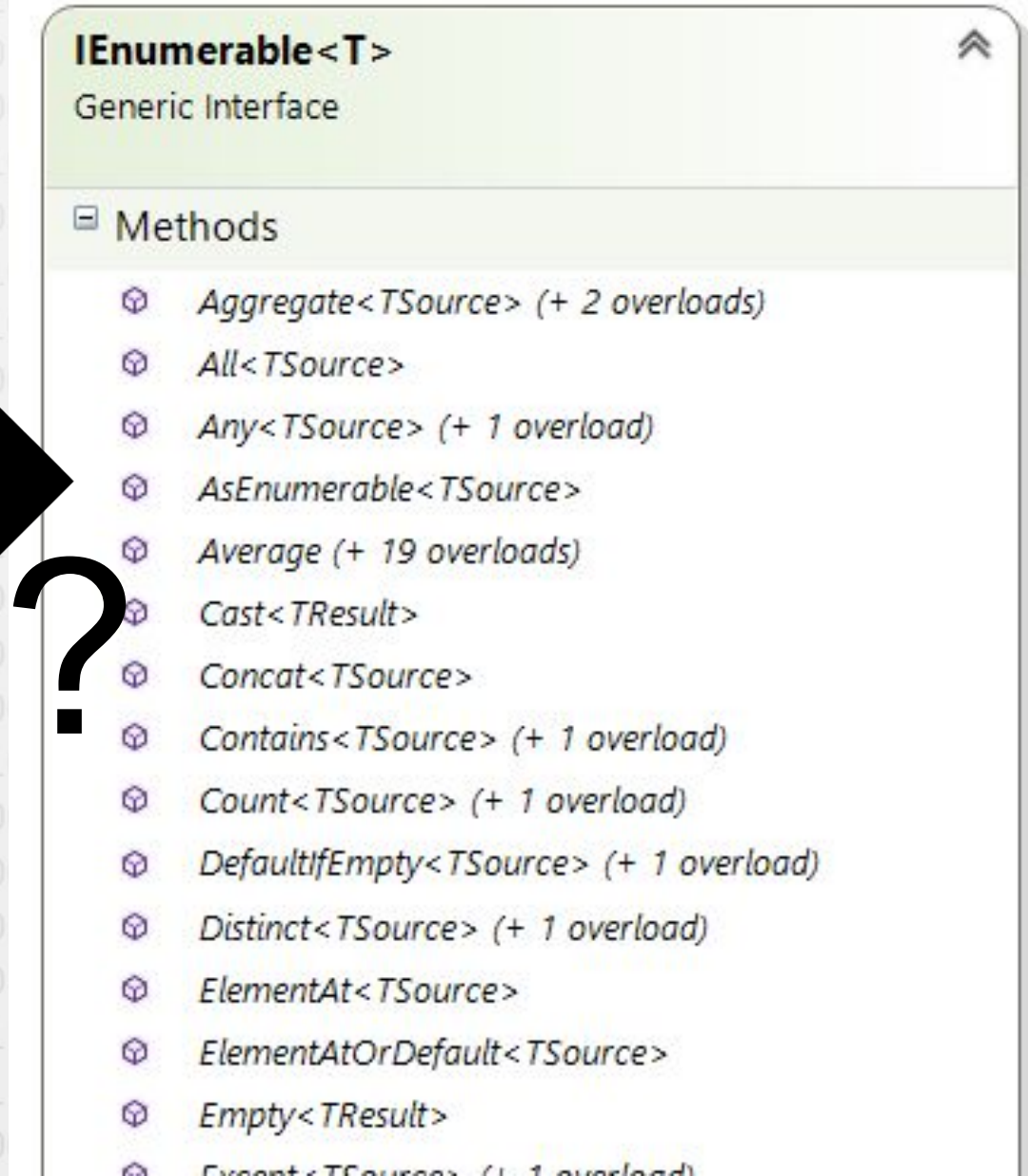
- **anonymous classes: instead of temporary, small classes that are only used to store data**

```
var PersonalData = new
{
    Name = "Béla",
    Age = 23,
    Address = "Budapest Bécsi út 96/B"
};
```

- **We can also set the properties using the same syntax with regular everyday classes as well (object initializer)**

Adding methods





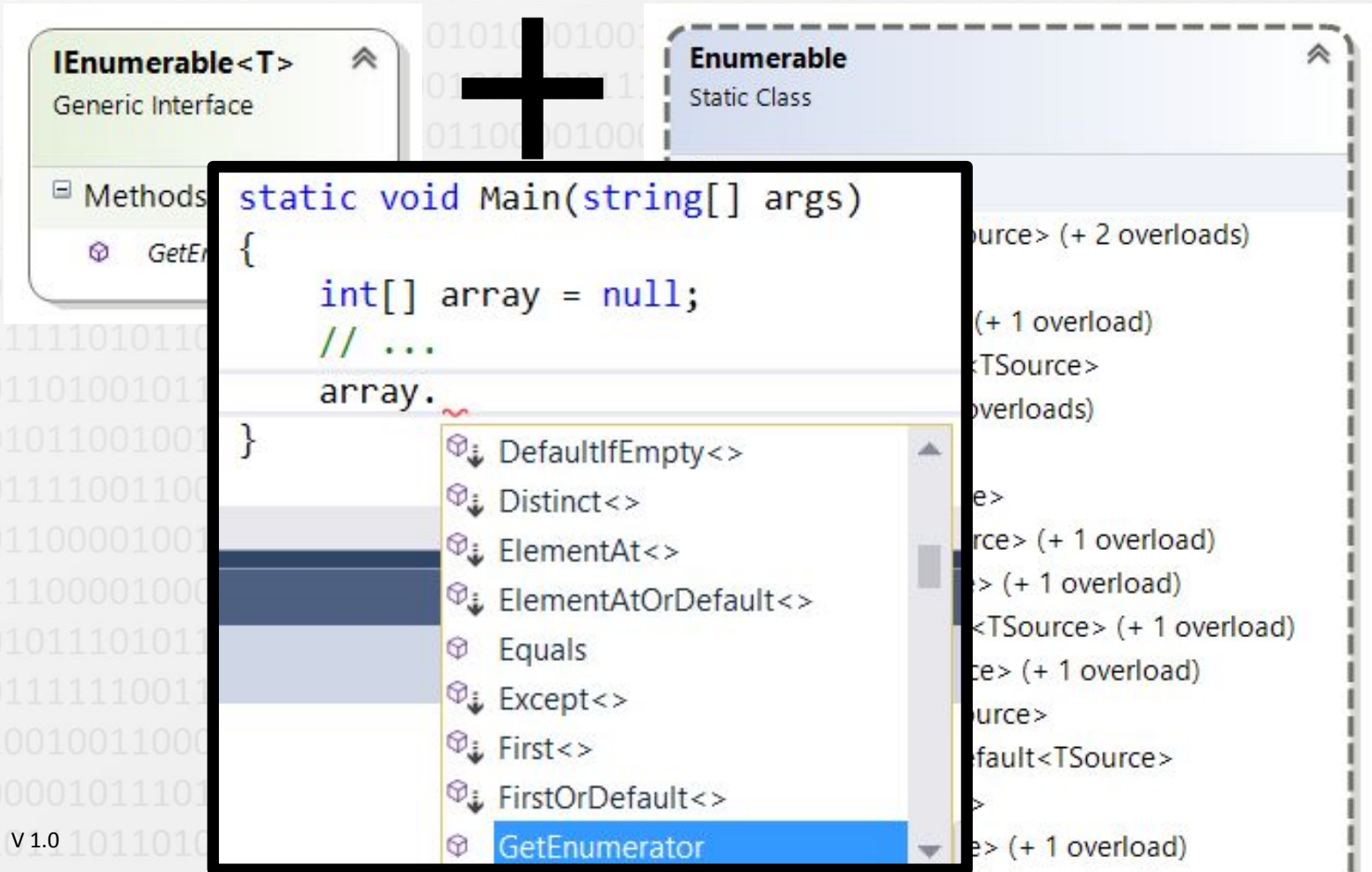
?

We want to add new methods to existing collections, BUT

- **… modifying the interface definition will have unwanted consequences (must be backwards-compatible)**
- **… we cannot add implemented methods to interfaces**

Extension methods

```
public static class Enumerable
{
    public static bool Contains<TSource>(this IEnumerable<TSource> source, TSource value);
    public static int Count<TSource>(this IEnumerable<TSource> source);
    public static TSource ElementAt<TSource>(this IEnumerable<TSource> source, int index);
```



With extension methods…

- We can call the same LINQ method for the typical operations, independent from the actual collection that we have
- The method that we call is an extension method of the interface, it will perform the same operation (filter/sort/etc) independent from the type of the collection
- List<T>, XML file IEnumerable extension methods Func<T, xxxxxx> parameter types Practice, second week
- Database IQueryable<T> extension methods Expression<Func<T, xxxxxx>> parameter types (same syntax) Practice, fourth week

With extension methods…

```
public static void ToConsole<T>(this IEnumerable<T> input, string str)
{
    Console.WriteLine("*** BEGIN " + str);
    foreach (T item in input)
    {
        Console.WriteLine(item.ToString());
    }
    Console.WriteLine("*** END " + str);
    Console.ReadLine();
}
```

```
var q3 = from person in people
         let minlen = people.Min(x => x.Name.Length)
         let maxlen = people.Max(x => x.Name.Length)
         where person.Name.Length == minlen || person.Name.Length == maxlen
         select new { person.Name, person.Name.Length };
q3.ToConsole("Q3");
```

LINQ operators

- **Extension methods for the IEnumerable<T> interface**
  - ... LINQ To Objects, LINQ To XML, LINQ to Entities, ...
- **They work on IEnumerable<T>, the output is:**
  - Rarely T or X (selection, conversion …)
  - But more often IEnumerable<T> or IEnumerable<X>, they can be chained
  - Very often the parameters are methods = lambda expressions
- **They perform common operations**
  - **Selecting an item from the collection** – first, last, single, ...
  - **Filter** – according to conditions...
  - **Sort** – ascending, descending, reverse, ...
  - **Handling of sets** – union, intersect, ...
  - **Aggregation (~calculations)** – max, min, ...
  - **Groupings**

Using lambda expressions with LINQ methods

- **The lambda expressions…**
  - Are parameters for the LINQ extension methods
  - We must separate: parameter and result of the extension method; parameter and result of the lambda expression
  - E.g. when we use a filter (Where) :
    - The extension method works with List<int> (=IEnumerable<T>)
    - The parameter for the extension method is a lambda of Func<T, bool>
    - The result of the extension method is IEnumerable<T>
    - The parameter of the lambda expression is T (one element: int)
    - The result of the lambda expression is bool (is the element wanted?)
    - The lambda expression (or: the method described with the expression) is called for all elements of the collection
    - The elements with TRUE as the result of the lambda expression will be in the resulting IEnumerable<T>

LINQ operator examples

```
int[] first = new int[] { 2, 4, 6, 8, 2, 1, 2, 3 };
int[] second = new int[] { 1, 3, 5, 7, 1, 1, 2, 3 };
string[] strArray = new string[] { "Béla", "Jolán",
  "Bill", "Shakespeare", "Verne", "Jókai" };
List<Student> students = new List<Student>();
students.Add(new Student("Első Egon", 52));
students.Add(new Student("Második Miksa", 97));
students.Add(new Student("Harmadik Huba", 10));
students.Add(new Student("Negyedik Néró", 89));
students.Add(new Student("Ötödik Ödön", 69));
```

LINQ operator examples - sets

- Concatenate two collections (not as sets = repeated elements) :

```
var allNumbers = first.Concat(second);
```

- Check for existance:

```
bool doesContainFour = first.Contains(4);
```

- Leave out repeated elements (convert to set):

```
var onlyDifferentNumbers = first.Distinct();
```

- Intersection of sets:

```
var sameItems = first.Intersect(second);
```

- Union of sets:

```
var unionOfSets = first.Union(second);
```

- Difference of sets:

```
var diffOfSets = first.Except(second);
```

LINQ operator examples - sort

- OrderBy
  - As a parameter it wants a method (lambda) that will determine the key from an element (key = the data that will be used for sorting – this must be IComparable)
  - The result is always IEnumerable<T>
  - Int array, sort by the numbers themselves:

```
var result = first.OrderBy(x => x);
```

  - String array, sort by the length of the strings:

```
var result = strArray.OrderBy(x => x.Length);
```

  - List of students, order by names:

```
var result = students.OrderBy(x => x.Name);
```

  - Exception, because its not Student : IComparable

```
var result = students.OrderBy(x => x);
```

LINQ operator examples – filter, count

- **Where / Count**
  - **The parameter lambda expression has a bool result**
  - **The result of the *Where* is a collection of elements where this lambda results in TRUE**
  - **The result of the *Count* is the number (int!), and it can be called without parameters total number of elements**
  - **Int array, the odd numbers:**

```
var result = first.Where(x => x % 2 == 1);
```

  - **String array, the four-letter elements:**

```
int result= strArray.Count(x => x.Length == 4);
```

LINQ operator examples – filter, selection

- List of students, where the number of credits is a prime number:

```
var result = students.Where(x =>
{
    if (x.Credits <= 1) return false;
    for (int i = 2; i <= Math.Sqrt(x.Credits); i++)
        { if (x.Credits % i == 0) return false; }
    return true;
});
// Második Miksa - 97, Negyedik Néró – 89
```

- Select a property / conversion:

```
var nameCollection = students.Select(x => x.Name);
var jsonCollection = students.Select(x => x.ToJson());
```

LINQ operator examples – chaining, query syntax

- List of students, only the odd credit numbers, sorted by names descending, show only the uppercase names:

```
var result= students.Where(x => x.Credits % 2 == 1)
  .OrderBy(x => x.Name)
  .Reverse()
  .Select(x => x.Name.ToUpper());
```

- Same result, same intermediate code, DECLARATIVE approach:

```
var result = from x in students
        where x.Credits % 2 == 1
        orderby x.Name descending
        select x.Name.ToUpper();
```

LINQ operator examples - aggregation

- **Aggregate methods**

```
int totalSum = first.Sum(); //28
double averageOfItems = second.Average(); //2.875
int sumOfEvenItems = first
  .Where(x => x % 2 == 0).Sum(); //24
int sumOfOddItems = second
  .Where(x => x % 2 == 1).Sum(); //4
```

- **The example above is common: I want to group my collection according to some feature, and execute the same aggregate for all groups**
  - We have to use multiple similar statements…
  - It's possible to call Sum/Average with a Func<T, bool> parameter, but it doesn't help much …
- **Instead, let's use automatic group creation: GroupBy**

LINQ operator examples - GroupBy

- Grouping, according to parity (even/odd) :

```
var groups = first.GroupBy(x => x % 2);
      // IEnumerable<IGrouping<TKey, TElement>>
foreach (var g in groups)
{
    Console.WriteLine("Remainder: " + g.Key +
    ", Number of items: " + g.Count());
}
```

- This is a lot better with a query syntax

```
var result = from x in first
  group x by x % 2 into g
  select new {Remainder=g.Key, NumItems=g.Count()};
```

LINQ operator examples – inner join

- If two collections have a common data field, then we can join them together (e.g. practice and Db-Linq and Databases lesson)
- This is typically a feature only used with query syntax, would be too complex without (a generic method with 4 generic type parameters, 2 collections and 3 lambda expressions…)
-
```
var result = from firstItem in firstCollection
             join secondItem in secondCollection
             on firstItem.X equals secondItem.Y
```
- For the rest of the query, every firstItem will be connected to the appropriate secondItem element, and both is useable (e.g. we join the appropriate brand for a car)
-
```
var result = from car in carCollection
             join brand in brandCollection
             on car.brandId equals brand.Id
```

JSON Linq

```
// JObject obj = JObject.Parse(json);
// obj["property"]?.ToString();
JArray array = JArray.Parse(json);
Console.WriteLine(array[0].ToString());
Color firstColor = array[0].ToObject<Color>();
Console.WriteLine($"FIRST COLOR: {firstColor.Red} - {firstColor.Green}
  {firstColor.Blue}");

var q = from color in array.Children()
        group color by color["Red"] into grp
        orderby grp.Key
        select new { RedValue = grp.Key, PixelCount = grp.Count() };
foreach (var item in q) Console.WriteLine(item);
Console.ReadLine();
```

LINQ to XML, XLINQ

- X* classes: strong LINQ support!
  - Lots of methods have IEnumerable<T> results that can be used with LINQ extension methods

Pl. XElement xe2:

- xe2.Descendants()
  - All children (including children of children) nodes
- xe2.Descendants("note")
  - Children (including children of children) nodes with the specified name
- xe2.Elements()
  - All immediate children nodes
- xe2.Elements("note")
  - Immediate children nodes with the specified name
- xe2.Attributes(), xe2.Ancestors() …

LINQ to XML

```
<people>
  <person id="43984">
    <name>Joe</name>
    <age>25</age>
    <phone>0618515133</phone>
  </person>
...
</people>
```

```
XDocument XDoc = ...
var q = from node in XDoc.Descendants("person")
        where node.Element("name").Value.StartsWith("J")
        select node;
```

Lab exercises and examples

Example

http://users.nik.uni-obuda.hu/prog3/_data/people.xml

```
<person>
  <name>Dr. Vámossy Zoltán</name>
  <email>vamossy.zoltan@nik.uni-obuda.hu</email>
  <dept>Alkalmazott Informatikai Intézet</dept>
  <rank>egyetemi docens, intézetigazgató-helyettes, TDK-fel
  <phone>+36 (1) 666-5550</phone>
  <room>BA.3.12</room>
</person>
```

Example

List those who are NOT working in the BA building (Alternatives: use the XML directly or with a data class representation?)

```
XDocument XDoc =
XDocument.Load("http://users.nik.uni-obuda.hu/prog3/_data/people.xml");
var q0 = from node in XDoc.Descendants("person")
                let room=node.Element("room").Value
                where !room.StartsWith("BA")
                select node.Element("name").Value;
foreach (var item in q0) {
    Console.WriteLine(item);
}
```

Example

- **By using a data class, we first convert the XML node into a typed instance, so we move back from XML processing to Linq To Objects methods**

```
class Person
{
    public static Person Parse(XElement node)
    {
        return new Person()
        {
            Name = node.Element("name")?.Value,
            Email = node.Element("email")?.Value,
            Dept = node.Element("dept")?.Value,
```

```
// IEnumerable<X> vs List<X>
public static IEnumerable<Person> Load(string url)
{
    XDocument XDoc = XDocument.Load(url);
    return XDoc.Descendants("person").
        Select(node => Person.Parse(node));
}
```

Example – Extension Method

```
static class MyExtensions
{
    public static void ToConsole<T>(
        this IEnumerable<T> input, string str)
    {
        Console.WriteLine("*** BEGIN " + str);
        foreach (T item in input)
        {
            Console.WriteLine(item.ToString());
        }
        Console.WriteLine("*** END " + str);
        Console.ReadLine();
    }

}
```

```
IEnumerable<Person> people = Person.
    Load("http://users.nik.uni-obuda.hu/prog3/_data/
      people.xml");
people.Select(person => person.Name).
    ToConsole("ALL WORKERS");
```

Example – Number of workers; paginated list

```
string dept = "Alkalmazott Informatikai Intézet";
int num = people.
    Where(person => person.Dept == dept).
    Count();
int num2 = people.
    Count(person => person.Dept == dept);
```

```
int current = 0; int pagesize = 15;
while (current < num)
{
    var q2 = people.
        Where(person => person.Dept == dept).
        Skip(current).
        Take(pagesize).
        Select(person=>person.Name);
    q2.ToConsole("Q2 / page");
    current += pagesize;
}
```

Example – Shortest and longest names

```
// 3. people with the longest/shortest name
// Query vs Method syntax???
var q3 = from person in people
         let minlen = people.Min(x => x.Name.Length)
         let maxlen = people.Max(x => x.Name.Length)
         where person.Name.Length == minlen ||
                  person.Name.Length == maxlen
         select new { person.Name, person.Name.Length };
q3.ToConsole("Q3");
```

Example – Groups; Biggest group

```
// 4. number of people per department
var q4 = from person in people
         group person by person.Dept into g
         select new { Dept = g.Key, Cnt = g.Count() };
q4.ToConsole("Q4");
```

```
// 5. biggest dept
// ElementAt, First, Last, Single, ...OrDefault
var oneDept = q4.
    OrderByDescending(rec=>rec.Cnt).
    FirstOrDefault();
var oneDept_alter = q4.
    Aggregate((i, j) => i.Cnt > j.Cnt ? i : j);
Console.WriteLine(oneDept.ToString());
Console.WriteLine(oneDept_alter.ToString());
```

Practice

http://users.nik.uni-obuda.hu/prog3/_data/war_of_westeros.xml

```
<battle>
  <name>Battle of the Golden Tooth</name>
  <year>298</year>
  <outcome>attacker</outcome>
  <type>pitched battle</type>
  <majordeath>1</majordeath>
  <majorcapture>0</majorcapture>
  <season>summer</season>
  <location>Golden Toot
  <region>The Westerlan
  <attacker>...</attack
  <defender>...</defend
</battle>
```

```
<defender>
  <king>Robb Stark</king>
  <commanders>
    <commander>Clement Piper</commander>
    <commander>Vance</commander>
  </commanders>
  <house>Tully</house>
  <size>4000</size>
</defender>
```

Practice

In the war of five kings …

1. **How many houses participated?**
2. **List the battles with the „ambush” type!**
3. **How many battles are there where the defending army won and there was a major capture?**
4. **How many battles were won by the Stark house?**
5. **Which battles had more than 2 participating houses?**
6. **Which are the 3 most violent regions?**
7. **Which one is the most violent region?**
8. **In the 3 most violent region, which battles had more than 2 participating houses? (Q5 join Q6)**
9. **List the houses ordered descending by the number of battles won!**
10. **Which battle had the biggest known army?**
11. **List the three commanders who attacked the most often!**

Források

- Lambda expressions: http://msdn.microsoft.com/en-us/library/bb397687.aspx
- Lambda expressions: http://geekswithblogs.net/michelotti/archive/2007/08/15/114702.aspx
- Why use Lambda expressions: http://stackoverflow.com/questions/167343/c-lambda-expression-why-should-i-use-this
- Recursive lambda expressions: http://blogs.msdn.com/b/madst/archive/2007/05/11/recursive-lambda-expressions.aspx
- Standard query operators: http://msdn.microsoft.com/en-us/library/bb738551.aspx
- Linq introduction: http://msdn.microsoft.com/library/bb308959.aspx
- 101 Linq samples: http://msdn.microsoft.com/en-us/vcsharp/aa336746
- Lambda: Reiter István: C# jegyzet (http://devportal.hu/content/CSharpjegyzet.aspx) , 186-187. oldal
- Linq: Reiter István: C# jegyzet (http://devportal.hu/content/CSharpjegyzet.aspx) , 250-269. oldal

- Fülöp Dávid XLinq prezentációja
- Linq to XML in 5 minutes: http://www.hookedonlinq.com/LINQtoXML5MinuteOverview.ashx
- Access XML data using Linq: http://www.techrepublic.com/blog/programming-and-development/access-xml-data-using-linq-to-xml/594
- Simple XML parsing examples: http://omegacoder.com/?p=254 , http://gnaresh.wordpress.com/2010/04/08/linq-using-xdocument/
- XML: Reiter István: C# jegyzet (http://devportal.hu/content/CSharpjegyzet.aspx) , 224. oldal (A könyv az XMLReader/Writer, illetve az XmlDocument használatát mutatja be)