Module

LINQ

Table of Contents

Introduction

- LINQ (Language Integrated Query)
  - Access different data sources with the same syntax
- LINQ Providers
  - Implement the standard query operators for a specific data source
  - Might implement more extension methods
  - DataBase SQL Server : LINQ to SQL
  - Files XML : LINQ to XML
  - Files JSON : LINQ to JSON
  - DataSet ADO.NET : LINQ to DataSet
  - Collections .NET, files .NET and string .NET, … : LINQ to Objects

Comparaison with Lambda Expression

- Sample

```
public class Student
{
        public String FirstName { get; set ; }
        public String LastName { get; set ; }
        public int Age {get ; set;}
        public String Section {get;set;}
        public char Annee {get;set;}
        public List <Course> CourseStudent { get; set;}
        …
}
public class Course
{
        public String Title {get; set;}
        public int ECTS {get: set;}
        …
}
```

Comparaison with Lambda Expression

- Sample(…)

```
public List <Student> GetStudent()
{
        List <Student> IESNStudent = new List <Student>(…);
        IESNStudent.Add(….);
        …;
        return IESNStudent;
}
```

Comparaison with Lambda Expression

```
IEnumerable <Student > IGStudent = AllStudent.GetStudent().Where (
                                    s => s.Section.Equals("IG") ) .Select s;
foreach (Student s in IGStudent) {  ….}
```

- LINQ Query

```
var IGStudent = from s in AllStudent.GetStudent() where s.Section.Equals("IG")
    select s;

foreach (Student s in IGStudent) {  ….}
```

PART 1. LINQ

Contents

- Restriction and Projection Operators
- Ordering Operators
- Grouping Operators
- Join Operators
- Element Operators
- Partitioning Operators
- Set Operators
- Miscellaneous Operators
- Quantifiers Operators
- Aggregate Operators
- Conversion Operators

Restriction and Projection Operators

```
IEnumerable <Student> IESNStudent = AllStudent.GetStudent();


var studentAgeMax20 = from student in IESNStudent
                where student.Age <= 20
                                select student;
var studentIG3 = from student in IESNStudent
                                where student.Section.Equals("IG") && student.Annee =='3'
                                select student;
var … = from student in IESNStudent
                        where student.Section.Equals("IG") && student.Annee == '3'
                        select student.LastName;
```

Restriction and Projection Operators

```
var … = from student in IESNStudent
                where student.Section.Equals("IG") && student.Annee == ‘3’
                select  new { student.FirstName, student.LastName};


var … = from student in IESNStudent
                 where student.Section.Equals("IG")
                   from course in student.CourseStudent
                     where course.ECTS >=5
                  select new  {  course.Title, student.LastName, student.FisrtName} ;
```

Ordering Operators

```
var … = from student in IESNStudent
                where student.Section.Equals("IG")
                 orderby student.LastName
          select student;

var … = from student in IESNStudent
                where student.Section.Equals("IG") && student.Annee == '3'
                orderby student.LastName descending
          select student;

var … = from student in IESNStudent
                orderby student.LastName, student.FirstName
           select student;

var … = (from student in IESNStudent
                orderby student.Lastname
                select student).Reverse();
```

Ordering Operators

```
string[] sections = { "iG", "Ingénieur", "droit", "Marketing" };
var sortedSections = sections.OrderByDescending(a => a, new CaseInsensitiveComparer());
foreach (string section in sortedSections)
    { … }

public class CaseInsensitiveComparer : IComparer<string>
{
     public int Compare(string x, string y)
     {
          return string.Compare(x, y, StringComparison.OrdinalIgnoreCase);
     }
 }
```

Grouping Operators

```
var …   = from student in IESNStudent
          group student by student.Section into section
          orderby section.Count() descending, section.Key
          where section.Count() > 10
        select new { Section = section.Key, Count = section.Count() };

var …   = from student in IESNStudent
          group student by student.Section into section
          orderby section.Count() descending, section.Key
          where section.Count() > 10
        select new { Section = section.Key,
                     Count = section.Count(),
                     ManyStudent = from s in section orderby s.LastName
                                   select s.FirstName + "  " +s.LastName };
```

Join Operators

```
string[] sections = new string[]{ "IG" , "TI", "MASI" };
var … = from section in sections
            join student in IESNStudent on section equals  student.section
        select new { Section = section, Name = student.LastName };

var …= from section in sections
            join student in IESNStudent on section equals  student.Section into StudentSection
        select new {Section= section, Student = StudentSection};

 var …= from section in sections
            join student in IESNStudent on section equals student.Section into StudentSection
                from s in StudentSection
         select new { Section = section, s.LastName };
```

Join Operators

```
var … = from section in sections
        join student in IESNStudent on section equals  student.Section into temp
        from s in temp.DefaultIfEmpty()
    select new { Section = section,
        LastName = (s == null )? "(No students)" : s.LastName
            };
```

Element Operators

- First()
- ElementAt()
- FirstOrDefault()

```
var … = (from student in IESNStudent orderby student.FirstName select student).First() ;

var … = ( from student in IESNStudent where student.Age <18 select student) .ElementAt(1)  ;
```

```
List <Student> students = new List <Student> { } ;

if ( students.FirstOrDefault() == null)
      …;
```

Partitioning Operators

- Take(n)
  - Returns n contiguous elements from the start of a sequence
- Skip(n)
  - Bypasses n elements from the start of a sequence
  - Returns the remaining elements
- TakeWhile(condition)
  - Returns the elements as long as the specified condition is true, skips the remainders
- SkipWhile(condition)
  - Bypasses the elements as long as the specified condition is true, returns the others

Set Operators

- Distinct()
  - Returns distinct elements from a sequence by using the default equality comparer to compare values
- Union()
  - Products the set union of 2 sequences
- Intersect()
  - Produces the set intersection of 2 sequences
- Except()
  - Produces the set difference of 2 sequences

Miscellaneous Operators

```
List<Product> oldProducts = …;
List<Product> newProducts = …;

var oldProductsNames = from p in oldProducts select p.ProductName;

var newProductNames = from p in newProducts select p.ProductName;
```

```
var allNames = oldProdutcsNames.Concat(newProductNames);
```

```
var wordsA = new string[] { "cherry", "apple", "blueberry" };
var wordsB = new string[] { "apple", "blueberry", "cherry" };

bool match = wordsA.SequenceEqual(wordsB);

Console.WriteLine("The sequences match: {0}", match);
```

Quantifiers Operators

```
var productGroups = from p in products
                    group p by p.Category into g
                    where g.All(p => p.UnitsInStock > 0)
                    select new { Category = g.Key, Products = g };

var productGroups = from p in products
                    group p by p.Category into g
                    where g.Any(p => p.UnitsInStock == 0)
                    select new { Category = g.Key, Products = g };
```

Aggregate Operators

- Return a single value
- Types
  - Count()
  - Sum()
  - Min()
  - Max()
  - Average()
  - Aggregate()

Aggregate Operators

```
List<Product> products = …;

 var categoryCounts = from p in products
                              group p by p.Category into g
                        select new { Category = g.Key,
                                     ProductCount = g.Count() };

var categories = from p in products
                        group p by p.Category into g
                  select new { Category = g.Key,
                               TotalUnitsInStock = g.Sum(p => p.UnitsInStock) };

var categories = from p in products
                        group p by p.Category into g
                  select new { Category = g.Key,
                               CheapestPrice = g.Min(p => p.UnitPrice) };
```

Aggregate Operators

```
var … = from p in products
                group p by p.Category into g
                let minPrice = g.Min(p => p.UnitPrice)
          select new { Category = g.Key,
                       CheapestProducts = g.Where(p => p.UnitPrice == minPrice) };

var … = from p in products
                group p by p.Category into g
          select new { Category = g.Key,
                       MostExpensivePrice = g.Max(p => p.UnitPrice) };

var …  = from p in products
                 group p by p.Category into g
                 let maxPrice = g.Max(p => p.UnitPrice)
           select new { Category = g.Key,
                        MostExpensiveProducts = g.Where(p => p.UnitPrice == maxPrice) };
```

Aggregate Operators

```
var categories = from p in products
                 group p by p.Category into g
                 select new { Category = g.Key,
                              AveragePrice = g.Average( p => p.UnitPrice ) };
```

```
var list = Enumerable.Range(5, 4);

Console.WriteLine("Result : {0}", list.Aggregate ( (a, b) => (a + b) ) );

Console.WriteLine("Result : {0}", list.Aggregate( (a, b) => (a * b) ) );
```

Converion Operators

```
List <Student> student =
                ( from student in IESNStudent orderby student.FirstName, student.LastName select student) . ToList();
```

```
object [ ] data = { "Iesn", 100, 200, "Hénallux"};

var query = data.OfType <string>();
```

```
var scoreRecords = new[ ]

    { new {Name = "Alice", Score = 50}, new {Name = "Bob" , Score = 40}, new {Name = "Cathy", Score = 45} };

var scoreRecordsDict = scoreRecords.ToDictionary(sr => sr.Name);

Console.WriteLine("Bob's score: {0}", scoreRecordsDict["Bob"]);
```

Conversion Operators

```
ILookup < string, Student > courseStudent =
                    ( from student in IESNStudent
                            from course in s.CourseStudent()
                      select new { TitleCourse = course.Title, Student = student }
                     ).ToLookup (cs => cs.TitleCourse, cs => cs.Student);

If ( courseStudent.Contains("Environnement de developpement ") )
{
   foreach (var envDev in CourseStudent ["Environnement de developpement "] )
                    …;
}
```

Part 2. LINQ & XML

Contents

- Structure of a XML File
- Some Classes
- How to Create/Save a Document
- Using LINQ
- Using XPath

Structure of a XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
< !–This file represents a fragment of a book store inventory database– >


< Bookstore >
  < Book genre="autobiography" publicationDate="1991" ISBN="1-861003-11-0" >
    < Title > The Autobiography of Benjamin Franklin < /Title >
    < Author >
      < FirstName > Benjamin < /First-name >
      < LastName > Franklin < /Last-name >
    < /Author >
    < Price > 8.99 < /Price >
  < /Book >
  < Book genre="novel" publicationdate="1967" ISBN="0-201-63361-2" >
    …
</Bookstore>
```

Some Classes

- Namespace System.Xml.Linq
- XmlNode
  - Abstract class that represents a single node in an XML document
  - XmlNodeList
- XmlDocument
  - Extends XmlNode
  - Is the W3C DOM (Document Object Model) implementation
  - Provides a tree representation in memory of an XML document, enabling navigation and editing
  - But a little old

Some Classes

- XDocument
  - More modern
- XDeclaration
  - Represents the declaration node (<?xml version='1.0'.>)
- XDocumentType
  - Represents data relating to the document type declaration
- XElement
  - Represents an XML element object
- XAttribute
- XComment

How to Create/Save a Document

```
XDocument doc = new XDocument(
                    new XComment("Students list"),
                    new XElement("ClassList",
                                        new XElement ("Student", new XAttribute("NumId", "1"),
                                                            new XElement("LastName",  "Charlier"),
                                                            new XElement("FirstName", "Isabelle") ),
                                        …
                    ));
doc.Save(@"c:\sample.xml");
```

How to Create/Save a Document

```
<?xml version="1.0" encoding="ISO-8859-15"?>
 <!– Students list -->
<ClassList>
                <Student NumId="1" >
                                <LastName>Charlier</LastName>
                                <FirstName>Isabelle<FirstName>
                </Student>
                <Student NumId=" 2" >
                                <LastName>Scholtes</LastName>
                                <FirstName>Samuel</FirstName>
                </Student>
</ClassList>
```

Using LINQ

```
XElement xml = new XElement("Students",
                 from student in db.Students orderby student.NumId
                select new XElement("Student", new XAttribute("NumId", student.NumId),
                                                    new XElement("FirstName", student.FirstName),
                                                    new XElement("LastName", student.LastName) )
);
xml.Save(@"C:\Students.xml");
```

```
<Students>
      <Student NumId="1">
               <FrstName>Charlier</FirstName>
               <ListName>Isabelle</LastName>
       </Student>
      <Student NumId="2">
               <FrstName>Scholtes</FirstName>
               <ListName>Samuel</LastName>
       </Student>
</Students>
```

Using LINQ

```
XDocument loadedDoc = XDocument.Load(@"C:\Students.xml");

var q = from student in loadedDoc.Descendants("Student")
                where (int)student.Attribute("NumId") < 4
          select (string)student.Element("FirstName") + " "+ (string)student.Element("LastName");


// another way
var xmlSource = students.Load(@"…");

var q = from student in xmlSource.Student
                where student.NumId < 4
                select student.FirstName + " " + student.LastName;
```

Using XPath

- XML Path Language
- Query language for selecting nodes in an XML document
  - Path expressions to navigate
  - Also to compute values from the content
- Used by XSLT
- W3C recommendation
  - XPath 3.0

Using XPath

```
XPathDocument document = new XPathDocument(@"…");

XPathNavigator navigator = document.CreateNavigator();

string expressionXPath = "Students/Student[NumId<4]/FirstName";

XPathNodesIterator nodes = navigator.Select(expressionXPath);

if (nodes.Count != 0) {

                foreach (XPathNavigator node in nodes)
                  …
}
```

Part 3. : LINQ & JSON

Contents

- Structure of a JSON File
- Framework Json.Net
- LINQ to JSON
- Querying in a JSON file with dynamic

Structure of a JSON File


```
{"city":
 {"id":2790471,"name":"Namur",
  "coord":{"lon":4.86746,"lat":50.4669},"country":"BE","population":0},
   "cod":"200", "message": 0.0331,"cnt":7,
    "list":[
        { "dt":1443956400,
          "temp":{"day":23.94,"min":12.64,"max":23.94,"night":16.82,"eve":19.25,"morn":12.64},
                "pressure":998.77,"humidity":78,
                "weather":[ {"id":800, "main":"Clear","description":"sky is clear", "icon":"02d"}],
                "speed":3.07, "deg":199, "clouds":8
        },
        {"dt":1444042800,
                  "temp":{"day....
```

Framework Json.Net

- Framework Json.Net 7.0.1
  - "Popular high-performance JSON framework for .NET"
  - *NuGet Gallery | Json.NET 7.0.1,* https://www.nuget.org/packages/Newtonsoft.Json
  - Moved to GitHub
- APIs
  - Newtonsoft.Json
  - Newtonsoft.Json.Serialization
  - Newtonsoft.Json.Linq
  - …

LINQ TO JSON

```
public async Task<IEnumerable<WeatherForecast>> GetForecast()
{
      HttpClient client = new HttpClient();
      var weather = await client.GetStringAsync( new
          Uri("http://api.openweathermap.org/data/2.5/forecast/daily?q=Namur,BE&mode=Json&units=metric&cnt=7"));
      var rawWeather = JObject.Parse(weather);

      var forecast = rawWeather["list"].Children().Select (d => new WeatherForecast()
      {
            Date = DateTime.Today,
            MinTemp = d["temp"]["min"].Value<double>(),
            MaxTemp = d["temp"]["max"].Value<double>(),
            WeatherDescription = d["weather"].First["description"].Value<string>(),
            WindSpeed = d["speed"].Value<double>()
       } );
       return forecast;
}
```

Querying in a JSON file with dynamic

```
var  jsonString = ….;

dynamic array = JArray.Parse(jsonString);

dynamic element  = array[0];

string title = element.Name;
```