TEKNOLOGISK INSTITUT

Module 14

LINQ to Objects

Motivation for LINQ

- LINQ = **L**anguage **IN**tegrated **Q**uery
- Several distinct motivations for LINQ
  - Uniform programming model for any kind of data
  - A better tool for embedding SQL queries into type-safe code
  - Another data abstraction layer
  - …
- All of these descriptions to some extent hold true

LINQ Components

- **LINQ to Objects**
- LINQ to XML
- LINQ to Entities
- Parallel LINQ
- …

Starting LINQ to Objects

- Main LINQ features live in `System.Core.dll` in the `System.Linq` namespace

Program.cs

Module_14_Examples.Program | Main(string[] args)

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Module_14_Examples
8  {
9      class Program
10     {
11         static void Main( string[] args )
12         {
13         }
14     }
15 }
16
```

100 %

Solution Explorer - Module 14 Examples

Search Solution Explorer - Module 14 Examples

- Solution 'Module 14 Examples' (1 project)
  - **Module 14 Examples**
    - Properties
    - References
      - Microsoft.CSharp
      - System
      - System.Core
      - System.Data
      - System.Data.DataSetExtensions
      - System.Xml
      - System.Xml.Linq
    - Program.cs

Solution Explorer | Team Explorer

A First Example

- Find all games with more that 18 characters in the title

```
string[] wiiGames = {
    "Super Mario Galaxy",
    "FIFA 09",
    "Guitar Hero III",
    "Wii Sports",
    "Wii Fit",
    "Legend of Zelda: Twilight Princess"
};

IEnumerable<string> query = from g in wiiGames
                            where g.Length >= 18
                            select g;

foreach (string s in query)
{
    Console.WriteLine(s);
}
```

Implicitly Typed Variables

- Query results can be of a multitude of types

```
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
IEnumerable<int> query = from i in numbers where i < 10 select i;
foreach (int i in query)
{
    Console.WriteLine(i);
}
```

- Innocently-looking modifications might change underlying type
- Make all query variables implicitly typed…!

```
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
var query = from i in numbers where i < 10 select i;
foreach (var i in query)
{
    Console.WriteLine(i);
}
```

Enumerable Extension Methods

- The `System.Linq.Enumerable` class provides a lot of extension methods

Program.cs

Module_14_Examples.Program | Main(string[] args)

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Module_14_Examples
8  {
9      class Program
10     {
11         static void Main( string[] args )
12         {
13             int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
14
15             numbers.
16         }
17     }
18 }
19
20
21
22
23
```

Aggregate<>
All<>
Any<>
AsEnumerable<>
AsParallel
AsParallel<>
AsQueryable
AsQueryable<>
Average

(extension) bool IEnumerable<TSource>.All<TSource>(Func<TSource,bool> predicate)
Determines whether all elements of a sequence satisfy a condition.

Exceptions:
System.ArgumentNullException

100 %

Deferred Execution

- Query expressions are not evaluated until they're enumerated!
- This is called *Deferred Execution*

```
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
IEnumerable<int> query = from i in numbers where i < 10 select i;
foreach (int i in query)
{
    Console.WriteLine(i);
}
```

- You can force evaluation through the Visual Studio debugger
  - Use the Results View of the query variable

Immediate Execution

- You can force evaluation by using conversion extension methods

```
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
var query = from i in numbers where i < 10 select i;
foreach (var i in query)
{
    Console.WriteLine(i);
}
```

- There are other such extension methods, e.g.
  - `ToDictionary<T,K>`

LINQ and Generic Collections

- LINQ can query data in various members of `System.Collections.Generic`

```
Stack<int> stack = new Stack<int>(new int[] { 42, 87, 112, 255 });
var query = from i in stack where i < 100 select i;
```

- Moreover, you can query generic collections of custom objects as well

```
List<Car> cars = new List<Car>() {
    new Car{ Color="Silver", Speed=100, Make="VW" },
    new Car{ Color="Tan", Speed=90, Make="BMW" },
    new Car{ Color="Black", Speed=55, Make="VW" },
    new Car{ Color="Rust", Speed=5, Make="Yugo" },
    new Car{ Color="White", Speed=43, Make="Ford" }
};

var query = from c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

LINQ and Nongeneric Collections

- Nongeneric collections lack the `IEnumerable<T>` infrastructure for querying
- This can be provided using the `OfType<T>` extension method

```
ArrayList cars = new ArrayList() {
    new Car{ Color="Silver", Speed=100, Make="BMW" },
    new Car{ Color="Tan", Speed=90, Make="BMW" },
    new Car{ Color="Black", Speed=55, Make="VW" },
    new Car{ Color="Rust", Speed=5, Make="Yugo" },
    new Car{ Color="White", Speed=43, Make="Ford" }
};

IEnumerable<Car> enumerableCars = cars.OfType<Car>();
var query = from c in enumerableCars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

- This method also filters entries not matching type

LINQ and Custom Collections

- LINQ queries can be performed directly on any `IEnumerable<T>` type
  - Even your own types!

```
class Garage : IEnumerable<Car>
{
    public IEnumerator<Car> GetEnumerator()...

    IEnumerator IEnumerable.GetEnumerator()...
}
```

```
Garage g = new Garage();
var query = from c in g
            where c.PetName.StartsWith("F")
            select c;

foreach (var c in query)
{
    Console.WriteLine(c.PetName);
}
```

The `from` Clause

- Range variables and data source are specified in the `from` clause

```
Stack<int> stack = new Stack<int>(new int[] { 42, 87, 112, 255 });
var query = from i in stack where i < 10 select i;
```

- It can define the type of the range variable as well

```
List<Car> cars = new List<Car> {
    new Car{ Color="Silver", Speed=100, Make="BMW" }
    // ...
};
var query = from Car c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

The `where` Clause

- Filtering conditions are specified by a boolean expression in a `where` clause

```
var cars = new List<Car> {
    new Car{ Color="Silver", Speed=100, Make="BMW" }
    // ...
};

var query = from Car c in cars where
            c.Speed > 90 && c.Make == "BMW"
            select c;
```

- This can be any boolean expression
- Can have multiple `where` clauses also

The `select` Clause

- Projections of results are done through the `select` clause

```
IEnumerable<string> query2 = from c in cars
                             where c.Speed > 90 && c.Make == "BMW"
                             select c.Make;
```

- Projections can create new (anonymous) data types

```
// Anonymous type
var query3 = from c in cars
             where c.Speed > 90 && c.Make == "BMW"
             select new { c.Make, c.Color };
```

The orderby Clause

- Results can be sorted using the `orderby` clause
  - The order can be `ascending` (the default) or `descending`

```
var query4 = from c in cars
             where c.Speed >= 55
             orderby c.PetName
             select c;

var query5 = from c in cars
             where c.Speed >= 55
             orderby c.PetName descending, c.Color
             select c;
```

The group Clause

- Results can be grouped using the `GroupBy` clause

```
int[] a1 = { 4, 4, 7, 1, 7, 6, 1 };
IEnumerable<IGrouping<int, int>> r1 = a1.GroupBy(i => i);
foreach (var item in r1)
{
    Console.WriteLine(item.Key);
    foreach (var item2 in item)
    {
        Console.WriteLine(".. " + item2);
    }
}
```

```
4
.. 4
.. 4
7
.. 7
.. 7
1
.. 1
.. 1
6
.. 6
```

Query Operators Resolution

- These query operators are keywords with syntax highlighting and IntelliSense
- But they are resolved as extension methods in the `Enumerable` class

```
var query2 = from g in wiiGames
             where g.Length >= 18
             orderby g.Length, g
             select g.ToUpper();
```

```
var query3 = wiiGames.Where(g => g.Length >= 18)
    .OrderBy(g => g.Length)
    .ThenBy(g => g)
    .Select(g => g.ToUpper());
```

- You can use either syntax or use delegates instead of anonymous methods etc.

Count<T>

- You can compute the number of items in the result set with Count<T>

```
string[] wiiGames = {
    "Super Mario Galaxy",
    "FIFA 09",
    "Guitar Hero III",
    "Wii Sports",
    "Wii Fit",
    "Legend of Zelda: Twilight Princess"
};
var query = from g in wiiGames
            where g.Length >= 18
            select g;
Console.WriteLine("{0} games match the query", query.Count());
```

- This forces an evaluation of the query expression!

Set Operations

- Differences and combinations can easily be found

```
int[] a1 = { 5, 4, 7, 1, 6, 6, 2 };
int[] a2 = { 1, 7, 1, 4, 6, 2, 1, 2, 9 };
var res1 = a1.Except(a2);       // 5
var res2 = a2.Except(a1);       // 9
var res3 = a1.Union(a2);        // 5, 4, 7, 1, 6, 2, 9
var res4 = a1.Intersect(a2);    // 4, 7, 1, 6, 2
var res5 = a1.Distinct();       // 5, 4, 7, 1, 6, 2
```

Singleton Operations

- A single element can be retrieved from a query result
  - `First<T>`
  - `Last<T>`
  - `Single<T>`

```
int[] a1 = { 5, 4, 7, 1, 6, 6, 2 };

var first = a1.First();          // 5
var last = a1.Last();            // 2

var theOnlyOne = a1.Single();    // exception

var a3 = a1.Intersect(new[] { 5 });
var theOnlyOne2 = a3.Single();   // 5
```

- Each of these has an …`OrDefault<T>` version
  - `FirstOrDefault<T>`
  - `LastOrDefault<T>`
  - `SingleOrDefault<T>`
- No exceptions but will simply return either null (reference types) or the default value of the value type. (e.g like 0 for an int.)

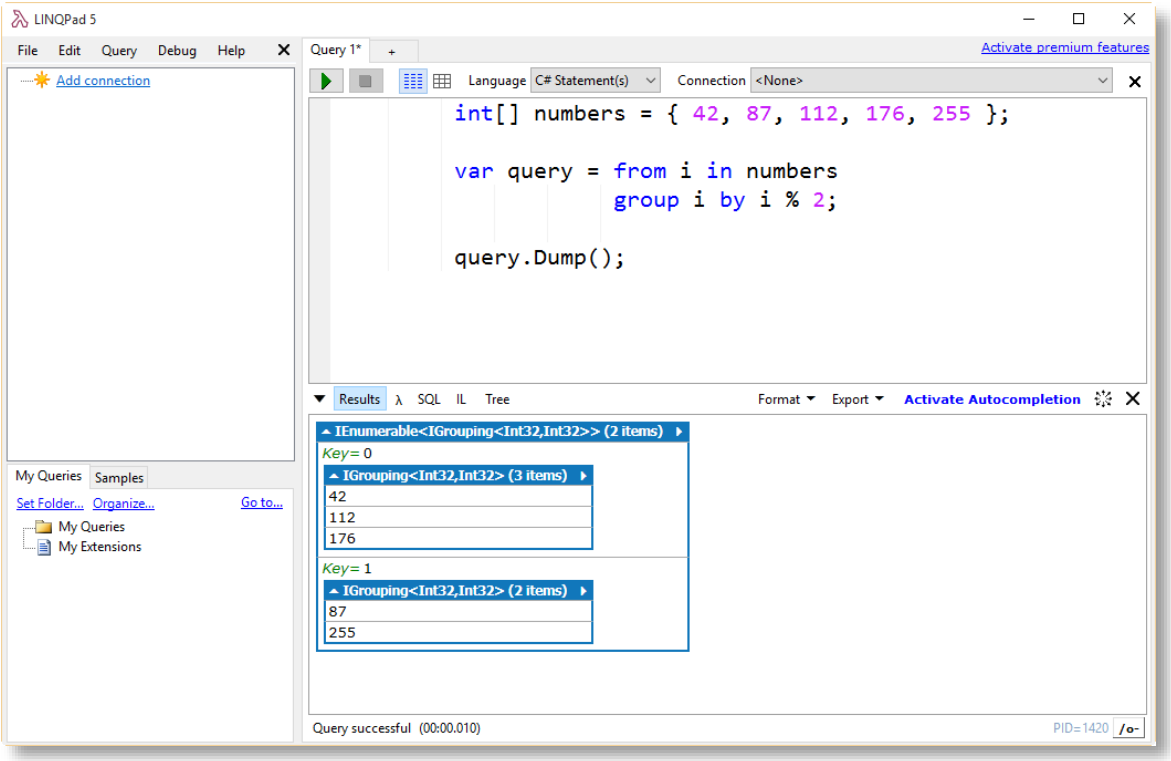
Partitioning Operators

- `Take()` and `Skip()`

```
int[] a1 = { 5, 4, 7, 1, 6, 6, 2 };
var r1 = a1.Take(3);                    // 5, 4, 7
var r2 = a1.Skip(3);                    // 1, 6, 6, 2
var r3 = a1.Skip(3).Take(3);            // 1. 6, 6
```

LINQPad

- LINQPad by Joseph Albahari is indispensable!



- Get it from http://www.linqpad.net