

Delegates, events and Linq

Lambdas

- A lambda expression uses a shorter notation to create a function/method.
- The function is anonymous and is often provided inline.
- Used in combination with delegates
- Relies heavily on type inference

```
bool Filter(int number)
{
    return number%2 == 0;
}
```

```
number => number%2 == 0;
```

Lab 7

LINQ-Specific Programming Constructs

- LINQ is basically a strongly-typed query language directly embedded in C#
 - Look and feel of LINQ queries is similar to SQL statements
 - LINQ can be applied to all kinds of data stores
 - Note: the syntax is not identical!
 - Often, it's the opposite of a SQL query
 - It's similar, nothing more!
- To work, a lot of new things had to be added to C# and VB with .NET 3.5
 - Implicitly typed local variables
 - Object/collection initialization syntax
 - Lambda expressions
 - Extension methods

Understanding the Role of LINQ

- LINQ API is an attempt to provide a consistent, symmetrical manner in which programmers can obtain and manipulate “data” in the broad sense of the term
 - With LINQ, we can create query expressions in the language itself
 - These are similar to SQL queries
- Query expressions can be used to interact with numerous types of data
 - Not all relational!
- There are different flavors of LINQ but they all work in the same way

LINQ and Extension Methods

- LINQ is a set of extension methods on `IEnumerable<T>` covered with some syntactic sugar.
 - Roland

Applying LINQ Queries to Primitive Arrays

- We'll start with LINQ to objects on a array

```
static void QueryOverStrings()
{
    // Assume we have an array of strings.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                   "Fallout 3", "Daxter", "System Shock 2"};
}
```

- Assume we want the items without spaces or with a number
 - Can be done without LINQ but will be quite "dirty"
- With LINQ, this becomes much easier
 - All with space in alphabetical order
 - Uses from, in where...
 - Easy to read
 - g is just a name
 - Could be anything

```
IEnumerable<string> subset = from game in currentVideoGames
                             where game.Contains(" ") orderby
                             game select game;
```

Applying LINQ Queries to Primitive Arrays

- Once we have the results in subset, we can then loop over the results

```
IEnumerable<string> subset = from game in currentVideoGames
                             where game.Contains(" ") orderby
                             game select game;
```

```
// Print out the results.
foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
```


It's all possible without LINQ!

- All you can do with LINQ can be done without it as well
 - Loops, ifs...
 - LINQ just makes things easier to write and read!

```
static void QueryOverStringsLongHand()
{
    // Assume we have an array of strings.
    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
        "Fallout 3", "Daxter", "System Shock 2"};

    string[] gamesWithSpaces = new string[5];

    for (int i = 0; i < currentVideoGames.Length; i++)
    {
        if (currentVideoGames[i].Contains(" "))
            gamesWithSpaces[i] = currentVideoGames[i];
    }

    // Now sort them.
    Array.Sort(gamesWithSpaces);

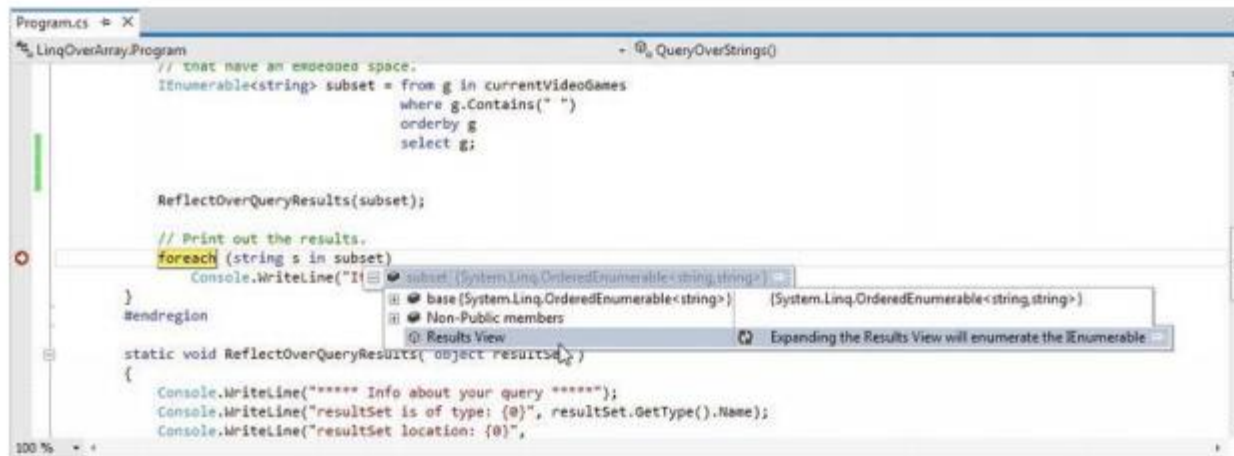
    // Print out the results.
    foreach (string s in gamesWithSpaces)
    {
        if (s != null)
            Console.WriteLine("Item: {0}", s);
    }
    Console.WriteLine();
}
```

The Role of Deferred Execution

- A very important note: LINQ query expressions are not actually evaluated until you iterate over the sequence
 - Deferred execution
 - Allows us to create several queries on the same container and still get back only the latest results

The Role of Deferred Execution

- Visual Studio allows us to trigger the execution of the expression
 - Click on the Results View



The Role of Immediate Execution

- When we need access to the results from code, we can call another extension method
 - `ToArray<T>()`
 - `ToDictionary<TSource,TKey>()`
 - `ToList<T>()`
- Cause the LINQ query to be executed at that point in time
 - If data changes, we don't see that in the result set
 - We get a snapshot of the data

```
static void ImmediateExecution()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    // Get data RIGHT NOW as int[].
    int[] subsetAsIntArray =
        (from i in numbers where i < 10 select i).ToArray<int>();

    // Get data RIGHT NOW as List<int>.
    List<int> subsetAsListOfInts =
        (from i in numbers where i < 10 select i).ToList<int>();
}
```

Returning the Result of a LINQ Query

- It is possible to define a field within a class (or structure) whose value is the result of a LINQ query
 - However, you cannot make use of implicit typing (as the `var` keyword cannot be used for fields) and the target of the LINQ query cannot be instance-level data
- LINQ queries are defined (almost always) in a method or a property
 - Result is stored in a `var`
 - But we can't create `var` on fields
- Returning the result is therefore always using `IEnumerable<T>`

Returning the Result of a LINQ Query

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ Transformations *****\n");
        IEnumerable<string> subset = GetStringSubset();

        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }

        Console.ReadLine();
    }

    static IEnumerable<string> GetStringSubset()
    {
        string[] colors = {"Light Red", "Green",
                           "Yellow", "Dark Red", "Red", "Purple"};

        // Note subset is an IEnumerable<string>-compatible object.
        IEnumerable<string> theRedColors = from c in colors
                                           where c.Contains("Red") select c;

        return theRedColors;
    }
}
```

LINQ syntax

- C# has a quite a few LINQ operators
- Most commonly used ones

Query Operators	Meaning in Life
from, in	Used to define the backbone for any LINQ expression, which allows you to extract a subset of data from a fitting container.
where	Used to define a restriction for which items to extract from a container.
select	Used to select a sequence from the container.
join, on, equals, into	Performs joins based on specified key. Remember, these “joins” do not need to have anything to do with data in a relational database.
orderby, ascending, descending	Allows the resulting subset to be ordered in ascending or descending order.
group, by	Yields a subset with data grouped by a specified value.

LINQ syntax

- Next to that, LINQ has a large number of extension methods as well for which there's no direct operator
 - `Reverse<>()`, `ToArray<>()`, `ToList<>()`
 - `Distinct<>()`, `Union<>()`, `Intersect<>()`
 - `Count<>()`, `Sum<>()`, `Min<>()`, `Max<>()`
 - ...

Obtaining Subsets of Data

- To obtain a specific subset from a container, you can make use of the where operator

```
var result = from item in container where BooleanExpression select item;
```

- Where expects an expression that evaluates to a Boolean

```
static void GetOverstock(ProductInfo[] products)
{
    Console.WriteLine("The overstock items!");

    // Get only the items where we have more than
    // 25 in stock.
    var overstock = from p in products where p.NumberInStock > 25 select p;

    foreach (ProductInfo c in overstock)
    {
        Console.WriteLine(c.ToString());
    }
}
```

```
// Get BMWs going at least 100 mph.
var onlyFastBMWs = from c in myCars
                    where c.Make == "BMW" && c.Speed >= 100 select c;
```

- In the where clause, we can use any C# operator

Projecting New Data Types

- It is also possible to project new forms of data from an existing data source
 - Take an existing item and create a new one
 - Can be done in combination with anonymous types

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
    Console.WriteLine("Names and Descriptions:");
    var nameDesc = from p in products select new { p.Name, p.Description };

    foreach (var item in nameDesc)
    {
        // Could also use Name and Description properties directly.
        Console.WriteLine(item.ToString());
    }
}
```

- Here var is required: this type is only known at compile time
 - Can't be used as return value this way
 - Transform using immediate execution if needed (ToArray)

Sorting Expressions

- A query expression can take an orderby operator to sort items in the subset by a specific value
 - Order is ascending by default
 - Alphabetical, numerical starting at 0

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
    // Get names of products, alphabetized.
    var subset = from p in products orderby p.Name select p;

    Console.WriteLine("Ordered by Name:");
    foreach (var p in subset)
    {
        Console.WriteLine(p.ToString());
    }
}
```

- Can be specified to be descending

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ Aggregation Operations

- LINQ queries can also be designed to perform various aggregation operations on the result set
 - Count()
 - Min()
 - Max()
 - Average()

```
static void AggregateOps()
{
    double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };

    // Various aggregation examples.
    Console.WriteLine("Max temp: {0}",
        (from t in winterTemps select t).Max());

    Console.WriteLine("Min temp: {0}",
        (from t in winterTemps select t).Min());

    Console.WriteLine("Avarage temp: {0}",
        (from t in winterTemps select t).Average());

    Console.WriteLine("Sum of all temps: {0}",
        (from t in winterTemps select t).Sum());
}
```

Practicing LINQ

- Create a collection of a simple type (e.g. `List<int>`) containing some values.
- Experiment with the myriad of LINQ methods available on the collection. For example:
 - Count
 - Where, Select
 - Min, Max, Average
- Now create a collection of a complex type (some class with properties)
- Repeat the experiments.