LINQ
Language Integrated Query

CSIS 3540

Client Server Systems

Class 04

Topics

- Role of LINQ
- LINQ Queries to Arrays
- LINQ Queries to Collections
- LINQ Query Operators

LINQ

- LINQ: Language Integrated Query
  - Support across C#, Vbasic, C++, etc
- Use unifying programming constructs to manipulate data
- LINQ is a Microsoft feature for querying datasources
  - Datasources are anything that implement the IEnumerable<> interface meaning that we can iterate through them
  - Databases and tables – SQL, Access, XML
  - Arrays and Lists (in memory)

LINQ API and Scope

The 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. Using LINQ, you are able to create directly within the C# programming language constructs called *query expressions*. These query expressions are based on numerous query operators that have been intentionally designed to look and feel similar (but not quite identical) to a SQL expression.

The twist, however, is that a query expression can be used to interact with numerous types of data—even data that has nothing to do with a relational database. Strictly speaking, "LINQ" is the term used to describe this overall approach to data access. However, based on where you are applying your LINQ queries, you will encounter various terms, such as the following:

- *LINQ to Objects*: This term refers to the act of applying LINQ queries to arrays and collections.
- *LINQ to XML*: This term refers to the act of using LINQ to manipulate and query XML documents.
- *LINQ to DataSet*: This term refers to the act of applying LINQ queries to ADO.NET `DataSet` objects.
- *LINQ to Entities*: This aspect of LINQ allows you to make use of LINQ queries within the ADO.NET Entity Framework (EF) API.
- *Parallel LINQ* (aka *PLINQ*): This allows for parallel processing of data returned from a LINQ query.

Today, LINQ is an integral part of the .NET base class libraries, managed languages, and Visual Studio itself.

What is a query?

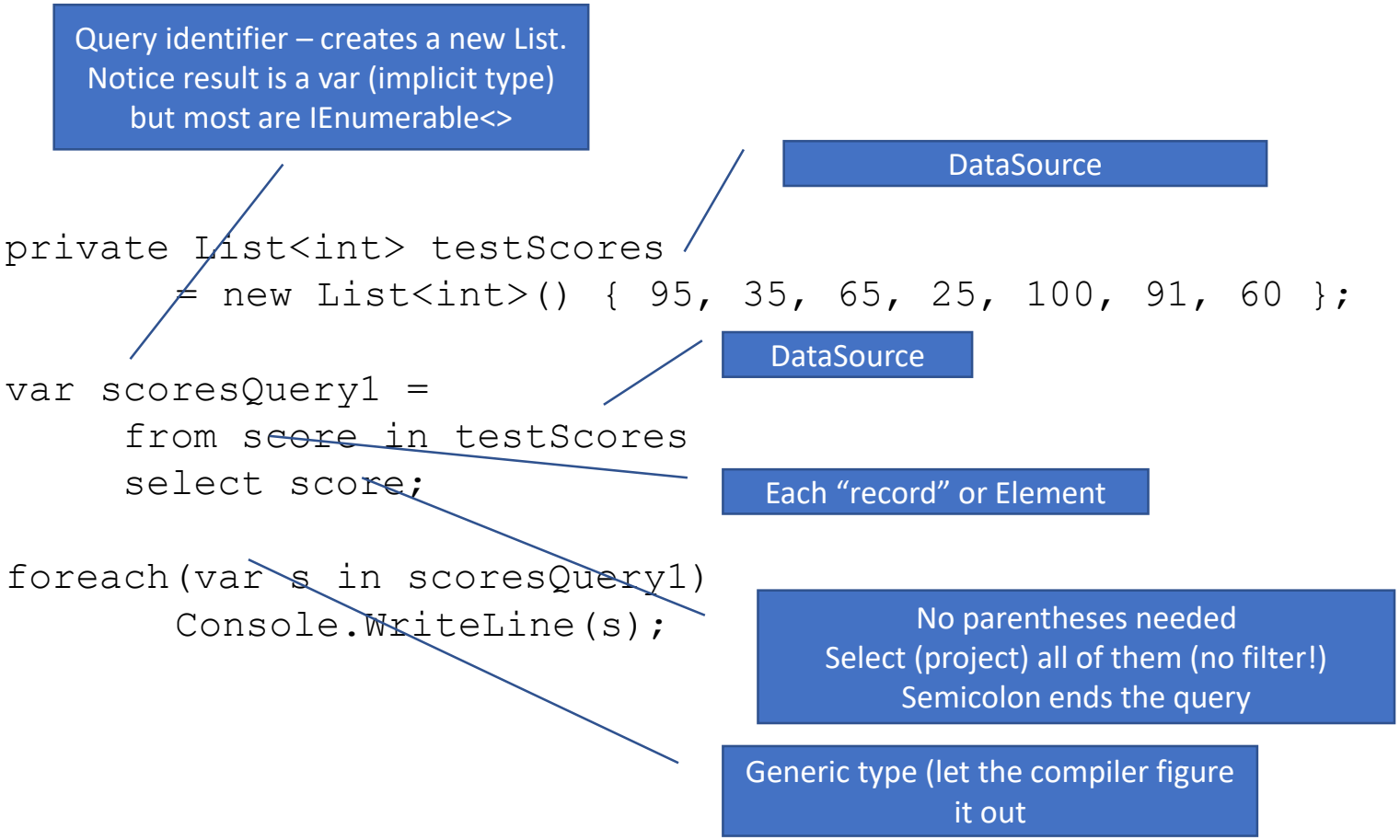
- Asking a question on a collection (set) of data records (Elements)
  - Array
  - List
- Each record (Element) in a datasource consists of one or more fields
- Given a datasource, filter elements based on some comparison
  - A field is compared using >, <, ==
- Then select (project) which elements should show in the result

Common LINQ operators

Table 12-3. *Common LINQ Query Operators*

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.

C# Basic LINQ Query



Filters – the where clause

- Queries can be filtered using the where keyword
- The form is
  - Where *expression*
    - Expression evaluates to Boolean (like an **if** statement)
    - Elements within the datasource record can be used
    - No parentheses are needed
  - **where** score > 70
- Multiple where clauses are allowed, equivalent to &&
- Compound logical expressions using && and || are allowed
- Min(), Max(), Average(), Sum() can be used
  - **where** scores.Min() > 40
  - Assuming scores is an array or list (datasource)

Filters – the where clause

```
private List<int> testScores
      = new List<int>() { 95, 35, 65, 25, 100, 91, 60 };

var scoresQuery1 =
      from score in testScores
      where score > 50
      where score < 90
      select score;

foreach(var s in scoresQuery1)
      Console.WriteLine(s);
```

Filters – the where clause

```
private List<int> testScores
        = new List<int>() { 95, 35, 65, 25, 100, 91, 60 };

var scoresQuery1 =
     from score in testScores
     where score > 50 && score < 90 // note use of compound logic
     select score;

foreach(var s in scoresQuery1)
        Console.WriteLine(s);
```

Subexpressions – the let statement

- The **let** statement allows you to compute intermediate variables which can then be queried/filtered
  - **let** scoresquared = score * score
  - where scoresquared < 500
- This is useful when we have multiple elements per record
  - Example: multiple test scores
    - let scoresum = score1 + score2
    - where scoresum < 180

Example let statement

```
private List<int> testScores
      = new List<int>() { 95, 35, 65, 25, 100, 91, 60 };

var scoresQuery1 =
     from score in testScores
     let adjustedScore = score / 10
     where adjustedScore > 5
     select adjustedScore; // new list of adjusted scores

foreach(var s in scoresQuery1)
      Console.WriteLine(s);
```

Sorting and Organizing: Orderby and Group

- orderby sorts the query output by the element(s) named
  - orderby name, price
- group <collection> by <expression> into <element>
  - Groups by the value of a an expression and creates a special subquery with an element.Key

Sorting and Organizing: Orderby and Group

```
private List<int> testScores
      = new List<int>() { 95, 35, 65, 25, 100, 91, 60 };

var scoresQuery1 =
     from score in testScores
     let adjustedScore = score / 10
     where adjustedScore > 5
     orderby adjustedScore
     select adjustedScore; // new list of sorted adjusted
scores

foreach(var s in scoresQuery1)
      Console.WriteLine(s);

// output: 2,3,6,6,9,9,10
```

Sorting and Organizing: Orderby and Group

```
Class Student { string Name; int Score;}
List<Student> students = new List<Student> { {“me”, 30},{“you”,20} }

// sorts by student score
var studentQuery1 =
                from student in students
                orderby student.Score
                select student;

// groups by first letter in student name
 var studentQuery2 =
                from student in students
                group student by student.Name[0] into studentGroup
                orderby studentGroup.Key
                select studentGroup;

 foreach (var studentGroups in studentQuery2)
            {
                Console.WriteLine(studentGroups.Key);
                foreach (var s in studentGroups)
                    Console.WriteLine(" " + s.Name);
            }
```

Join

- Like a SQL join
- Joins two collections based on some key
- Can be done with multiple from statements and a where clause

```
Syntax: from <object> in <collection join <object2> in <collection2>
        on <object>.key equals <object2>.key


Example: myCars and yourCars are lists of strings

var carJoin = from myCar in myCars
                          join yourCar in yourCars
                          on myCar equals yourCar
                          select myCar;
```

LINQ primitives on datasources

- System.Linq.Enumerable class provides a set of methods that do not have a direct C# query operator shorthand notation but are instead exposed as extension methods.
  - transform a result set in various
    - Reverse<>()
    - ToArray<>()
    - ToList<>()
    - others
  - perform various set operations
    - Distinct<>()
    - Union<>()
    - Intersect<>()
    - others
  - aggregate results
    - Count<>()
    - Sum<>()
    - Min<>()
    - Max<>()
    - Average<>()
    - others
- All query expressions can also be invoked as methods
- Can embed a function as an argument
- See FunWithLinqExpressions

Linq Methods

- Method syntax can be used in place of query syntax
- Methods can be chained in a pipeline
- See LinqMethods

```
/// <summary>
        /// Query a list of strings using standard Linq operators.
        /// This is easy to read and is preferred.
        /// </summary>
        /// <param name="list"></param>
        static void QueryStringWithOperators(IEnumerable<string> list)
        {
            Console.WriteLine("***** Using LINQ Query Operators *****");

            var subset = from item in list
                         where item.Contains(" ")
                         orderby item
                         select item;

            foreach (string item in subset)
                Console.WriteLine("Sorted Item: {0}", item);
        }

        /// <summary>
        /// Query the list using Linq methods and lambdas as args to the methods
        /// </summary>
        /// <param name="list"></param>
        static void QueryStringsWithLambdas(IEnumerable<string> list)
        {
            Console.WriteLine("***** Using LINQ Methods Syntax with lambdas *****");

            // Build a query expression using extension methods
            // which is granted to the Enumerable type.
            //  item => item seems a bit silly, but a Func is needed as arg.

            var subset = list
                .Where(item => item.Contains(" "))
                .OrderBy(item => item)
                .Select(item => item);

            // Print out the results.
            foreach (var item in subset)
                Console.WriteLine("Sorted Item: {0}", item);
            Console.WriteLine();

        }
```

LINQ Features

- LINQ queries return IEnumerable<>
  - Array, List, and other collections
  - See LinqRetValues, LinqOverArray, LinqOverCollections
- We don't usually know the return type, so use implicit (var) rather than IEnumerable<T>
- Deferred execution
  - query not actually implemented until iterated over the sequence
  - think of LINQ as a definition, the iteration executes it
  - ToArray(), ToList() can make the query execute immediately
- Easier than using IEnumerable<T> and lambda expressions!
- See LinqMethods

LINQ under the covers

- Query expressions are created using various C# query operators.
- Query operators are simply shorthand notations for invoking extension methods defined by the System.Linq.Enumerable type.
- Many methods of Enumerable require delegates (Func<> in particular) as parameters.
- Any method requiring a delegate parameter can instead be passed a lambda expression.
- Lambda expressions are simply anonymous methods in disguise (which greatly improve readability).
- Anonymous methods are shorthand notations for allocating a raw delegate and manually building a delegate target method.
  - See LinqMethods
- BASICALLY USE LINQ!!!
  - it uses lambdas which use anonymous methods which use delegates underneath the covers
  - This will be KEY to constructing DB Queries.

References

- Introduction to LINQ Queries
  - https://msdn.microsoft.com/en-us/library/bb397906.aspx
- Basic Queries
  - https://msdn.microsoft.com/en-us/library/bb397927.aspx
- Examples
  - https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b/viewsamplepack