LINQ

Language-INtegrated Query

Agenda

- What is LINQ?
- The fundamental building blocks of LINQ
- LINQ to Objects

What is LINQ?

- LINQ is a uniform programming model for any kind of data. And enables you to query and manipulate data with a consistent model that is independent from data sources.
- LINQ defines a set of method names (called standard query operators, or standard sequence operators), along with translation rules from so-called query expressions to expressions using these method names, lambda expressions and anonymous types.
- LINQ query:

```
var adultNames = from person in people
                 where person.Age >= 18
                 select person.Name;
```

Why Use LINQ?

- The two most common sources of non-OO information are relational databases and XML.
- Rather than add relational or XML-specific features to our programming languages and runtime, with the LINQ project Microsoft have taken a more general approach and added general-purpose query facilities to the .NET Framework that apply to all sources of information.
- Language-integrated query (LINQ) allows ***query expressions*** to benefit from the rich metadata, compile-time syntax checking, static typing and IntelliSense that was previously available only to imperative code.

THE FUNDAMENTAL BUILDING BLOCKS OF LINQ

Features in the language that is necessary to enable LINQ

Foundation

- The fundamental building blocks of LINQ are:
  - Generics
  - Anonymous methods
  - Implicit typing of local variables
  - Lambda expressions and expression trees
  - Extension methods

Generics

- Generics provide a facility for creating data structures that are specialized to handle specific types when declaring a variable.
- Programmers define these parameterized types so that each variable of a particular generic type has the same internal algorithm but the types of data and method signatures can vary based on programmer preference.
- **`List<T>`** is just a list of items of whatever type is specified
  - `List<string>` is a list of strings.
  - `List<int>` is a list of integers.
  - `List<MyClass>` is a list of objects of type MyClass.

Anonymous Methods

- Anonymous methods allow you to specify the method for a delegate instance inline as part of the delegate instance creation expression.

```
// Create a handler for a click event.
button1.Click += delegate(Object o, RoutedEventArgs e)
    { MessageBox.Show("Click!"); };
```

```
// Create a delegate.
delegate void Del(int x);

// Instantiate the delegate using an anonymous method.
Del d = delegate(int k) { /* ... */ };
```

- By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method.

Generics Delegate Types

- The generic mechanism is also used to declare some very useful generic delegate types:
  - Predicate
  - Comparison
  - Action
  - Func

- And these types are often used together with Anonymous Methods (and Lambda expressions).

Predicate

- BCL defines a predicate type:

```
public delegate bool Predicate<T>(T obj)
```

- Predicates are usually used in filtering and matching:

```
Predicate<int> isEven = delegate(int x) { return x % 2 == 0; };

Console.WriteLine(isEven(1));
Console.WriteLine(isEven(4));
```

Comparison

- BCL defines a Comparison type:

```
public delegate int Comparison<in T>(T x, T y)
```

Where return value:

Value	Meaning
Less than 0	x is less than y.
0	x equals y.
Greater than 0	x is greater than y.

Comparison Example

```
static void SortAndShowFiles(string title, Comparison<FileInfo> sortOrder)
{
  FileInfo[] files = new DirectoryInfo(@"C:\").GetFiles();
  Array.Sort(files, sortOrder);
  Console.WriteLine(title);
  foreach (FileInfo file in files)
  {
    Console.WriteLine (" {0} ({1} bytes)", file.Name, file.Length);
  }
}
...
SortAndShowFiles("Sorted by name:", delegate(FileInfo f1, FileInfo f2)
  { return f1.Name.CompareTo(f2.Name); }
);

SortAndShowFiles("Sorted by length:", delegate(FileInfo f1, FileInfo f2)
  { return f1.Length.CompareTo(f2.Length); }
);
```

Action

- BCL defines four Action types:

```
delegate void Action<T>( T obj )
..
delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2,
                                     T3 arg3, T4 arg4 )
```

- An Action encapsulates a method that has from 1 to 4 parameters and does not return a value.
- You can use the Action<T> delegate to pass a method as a parameter without explicitly declaring a custom delegate.

Func

- There are five generic Func delegate types in BCL:

```
TResult Func<TResult>()
TResult Func<T,TResult>(T arg)
..
TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2,
                                  T3 arg3, T4 arg4)
```

- Encapsulates a method that has from 0 to 4 parameters and returns a value of the type specified by the TResult parameter.

- E.g.:

```
 Func<string,double,int>
```

is equivalent to a delegate type of the form

```
public delegate int SomeDelegate(string arg1, double arg2)
```

Implicit Typing of Local Variables

- You can ask the compiler to infer the types of **local** variables for you:
  - Just replace the type part of a normal local variable declaration with var.

```
MyType variableName = someInitialValue;
↓
var variableName = someInitialValue;
```

- The compiler simply takes the compiletime type of the initialization expression and makes the variable have that type too.
- The variable is still statically typed!
  - You just haven’t written the name of the type in your code.

Lambda Expressions

- A lambda expression is an **anonymous function** that can contain expressions and statements, and can be used to create:
  - delegates or
  - expression tree types.
- All lambda expressions use the lambda operator:

  =>

  which is read as "goes to".
- The left side of the lambda operator specifies the input parameters (if any)
- The right side holds the expression or statement block.
- The lambda expression x => x * x is read "x goes to x times x."

Lambda Expressions

- Using an anonymous method to create a delegate instance:

```
 Func<string,int> returnLength;
returnLength = delegate (string text) { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

- A long-winded lambda expression:

```
returnLength = (string text) => { return text.Length; };
```

- If you can express the whole of the body in a single expression

```
returnLength = (string text) => text.Length;
```

- If the compiler can guess the parameter types:

```
returnLength = (text) => text.Length;
```

- When the lambda expression only needs a single parameter, and that parameter can be implicitly typed:

```
returnLength = text => text.Length;
```

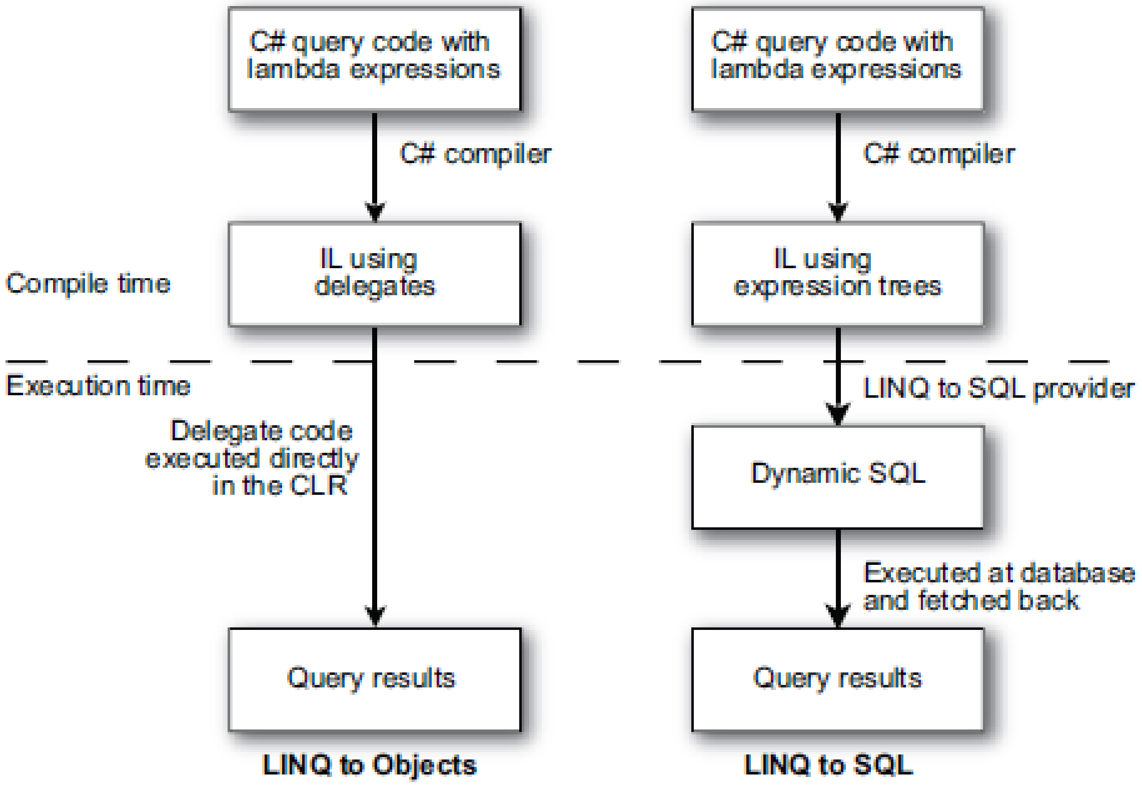
Expression Trees

- Expression trees represent code in a tree-like data structure, where each node is an expression.
- When a lambda expression is assigned to a variable of type `Expression<TDelegate>` the compiler emits code to build an expression tree that represents the lambda expression.
- The C# compiler can only generate expression trees from expression lambdas (single-line lambdas).
- Example:

```
Expression<Func<int, bool>> myLambda = num => num < 5;
```

The Use of Expression Trees

- Both LINQ to Objects and LINQ to SQL start with C# code and end with query results. The ability to execute the code remotely as LINQ to SQL does comes through expression trees.


C# query code with lambda expressions
C# compiler
IL using delegates
Compile time
Execution time
Delegate code executed directly in the CLR
Query results
LINQ to Objects
C# query code with lambda expressions
C# compiler
IL using expression trees
LINQ to SQL provider
Dynamic SQL
Executed at database and fetched back
Query results
LINQ to SQL

Extension Methods

- Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- Extension methods are a special kind of **static** method, but they are called as if they were instance methods on the extended type.
- Their first parameter specifies which type the method operates on, and the parameter is preceded by the **this** modifier.
- Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

```
public static class MyExtensions
{
  public static int WordCount(this String str)
```

Using Extension Methods

- For client code written in C# there is no apparent difference between calling an extension method and the methods that are actually defined in a type.
- Extension methods are defined as static methods but are called by using instance method syntax.

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

- To enable extension methods for a particular type, just add a using directive for the namespace in which the methods are defined.

Extension Methods and Encapsulation

- Extension methods cannot access private variables in the type they are extending!
- Therefore, the principle of encapsulation is not being violated.

LINQ TO OBJECTS

LINQ

- Defines a set of general purpose ***standard query operators*** that allow:
  - traversal,
  - filter, and
  - projection

  operations to be expressed in a direct yet declarative way in any .NET-based programming language.
- The standard query operators are defined as extension methods in the type **System.Linq.Enumerable**.
- Almost all standard query operators are defined in terms of the **IEnumerable<T>** interface.
- This means that every **IEnumerable<T>**-compatible information source gets the standard query operators simply by adding the following using statement in C#:

```
using System.Linq;
```

LINQ

- The developer is free to use:
  - named methods,
  - anonymous methods, or
  - lambda expressions

  with query operators.
- Lambda expressions have the advantage of providing the most direct and compact syntax for authoring.
- But more importantly:
  - lambda expressions can be compiled as either code or data, which allows lambda expressions to be processed at runtime by optimizers, translators, and evaluators.

LINQ Example

```
string[] names = { "Burke", "Connor", "Frank","Everett", "Albert", "George"};

var query = from s in names
            where s.Length == 5
            orderby s
            select s.ToUpper();

foreach (string item in query)
    Console.WriteLine(item);

// Is equivalent to
IEnumerable<string> query2 = names
                            .Where(s => s.Length == 5)
                            .OrderBy(s => s)
                            .Select(s => s.ToUpper());

foreach (string item in query2)
    Console.WriteLine(item);
```

Query Syntax: from

- Every query expression starts off in the same way - stating the source of a sequence of data:

```
      from element in source
```

  - The ***element*** part is just an identifier
  - The ***source*** part is just a normal expression.

Query Syntax: select

- Query expressions always end with either a select clause or a group clause:

  ```
  select expression
  ```

- The select clause is known as a ***projection***.

- Minimal (useless) query:

  ```
  var query = from name in names
              select name;
  ```

Query Syntax: OrderBy

- The **OrderBy** and **OrderByDescending** operators can be applied to any information source and allow the user to provide a key extraction function that produces the value that is used to sort the results.

```
    var s1 = names.OrderBy(s => s);
    var s2 = names.OrderByDescending(s => s);
```

- OrderBy and OrderByDescending also accept an optional comparison function that can be used to impose a partial order over the keys.

```
    var s3 = names.OrderBy(s => s.Length);
    var s4 = names.OrderByDescending(s => s.Length);
```

Query Syntax: ThenBy

- To allow multiple sort criteria, both **OrderBy** and **OrderByDescending** return **OrderedSequence<T>** rather than the generic **IEnumerable<T>**.
- Two operators are defined only on **OrderedSequence<T>**, namely **ThenBy** and **ThenByDescending** which apply an additional (subordinate) sort criterion.

```
var s1 = names.OrderBy(s => s.Length).ThenBy(s => s);
```

Query Syntax: reverse

- **Reverse** simply enumerates over a sequence and yields the same values in reverse order.
- Unlike **OrderBy**, **Reverse** doesn't consider the actual values themselves in determining the order, rather it relies solely on the order the values are produced by the underlying source.

Query Syntax: …

Self study

LINQ Extensibility

- LINQ allows third parties to augment the set of standard query operators with new domain-specific operators that are appropriate for the target domain or technology.
  - LINQ to SQL
  - LINQ to Entities
  - LINQ to XML

  - LINQ to Google
  - LINQ to CSV
  - LINQ to Twitter
  - LINQ to …

References and Links

- LINQ: .NET Language-Integrated Query (Don Box, Anders Hejlsberg)
  http://msdn.microsoft.com/en-us/library/bb308959.aspx
- **.NET Framework Developer Center > Learn > LINQ**
  http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
- **101 LINQ Samples**
  http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b
- C# in Depth
  http://csharpindepth.com/