Programming in .NET

Microsoft Development Center Serbia programming course

Lesson 6 - LINQ

LINQ, or *Language Integrated Query* as the full name of the feature stands, is a new and very powerful way to work with certain data types in C# programs. They allow the programmer to use declarative model of programming. Instead of explicitly saying to the compiler what needs to happen in order to produce some result, the programmer effectively says what is the result of the operation he is interested in, and leaves the compiler to find the best way to make it happen.

LINQ consists of a set of *query expressions* that are used mostly on sequences. You've covered sequences in previous lessons when talking about iterators. For the purposes of LINQ, a sequence is anything that implements `IEnumerable` or its generic cousin `IEnumerable<T>`. This means that LINQ can be used on any generic collection type within a C# program.

LINQ exists in several distinct "flavors" within the .NET Framework. First, most common, is *LINQ to Objects*. This flavor of LINQ works with sequences that are in memory, and usually use delegates to perform its work. Second most common is *LINQ to Entities* or LINQ to SQL. This flavor works by abstracting the data stored within a relational database in an object model, and then performs translation into Structured Query Language (SQL).

Before we dive deeper into LINQ territory, there are a couple of C# features that need to be covered, most importantly *anonymous types* and type inference, as well as expression trees.

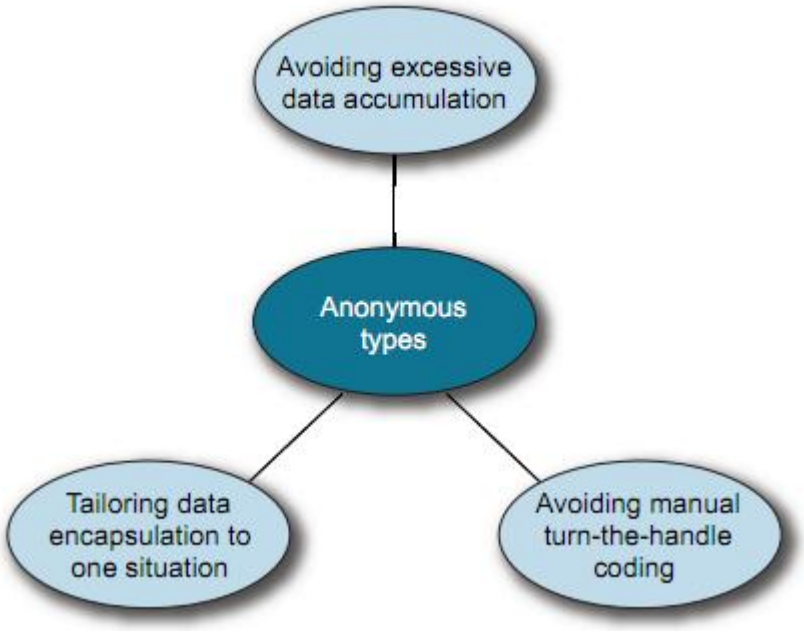
Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. You create anonymous types by using the `new` operator together with an object initializer.

```
// define a family member by his/her name and age
var result = new { Name = "Merry", Age = 12 };

// accessing properties is the same as with any other object
Console.WriteLine(result.Name); // prints out: Merry
```

```
// it can be used in implicitly typed arrays as well
// so let's define a whole family
 var family = new []{
     new { Name = "Merry", Age = 12 },
     new { Name = "Brad", Age = 15 },
     new {  Name = "Vivian", Age = 34 },
     new { Name = "John", Age = 35 }
 };
```

The figure below outlines the uses you can have from using anonymous types.



Working with anonymous types means that the programmer doesn't know the actual type of the variable that will contain the anonymous type; the type will be generated at compile type. In order to overcome this problem, C# has another feature which is called *typed inference*.

Type inference, also called implicitly typed local variables, allows the programmer to let the complier deduce the type of any given variable by looking at the right-hand side of the assignment operation. Looking at the example above:

```
// define a family member by his/her name and age
var result = new { Name = "Merry", Age = 12 };
```

We see the `var` keyword being used before the result identifier. This will instruct to complier to deduce the actual type of the variable by evaluating the expression on the right side of the assignment. If course, this can be used for any type, not just anonymous ones. Consider:

```
// the following two statements are identical
var result = "Hello World!";
string result = "Hello World!";

// of course, we can use more complex types
public class ClassA { public Tuple<int,string> PropA {get;set;} }

var result = new ClassA();
result.PropA.Item1 = 1;
result.PropA.Item2 = "John";
```

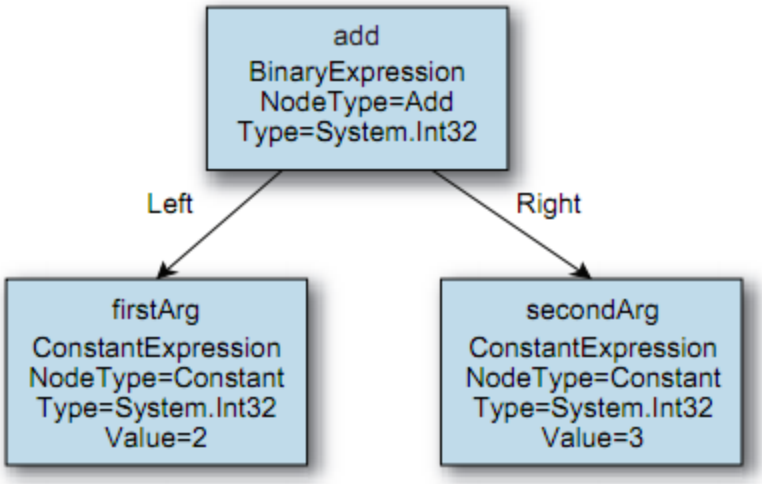
Expression trees in C# allow us to treat a piece of C# code as data, not code. This concept may sound strange, however it is really quite simple and it has been around for a while in various different languages. One of the most popular languages used for web development these days, JavaScript, had an `eval()` function from its first version, which allowed a JavaScript programmer to define a string containing some code and pass it to the `eval()` function at some point in the program's workflow for execution.

Expression trees do the same for C# in a very high-level and abstract way that makes it more of a joy to work with. C# compiler provides a built-in way of converting lambdas into expression trees, as well as a programmatic API to do so. Let us take a look at the latter first.

As the name suggests, they're trees of objects, where each node in the tree is an expression in itself. Different types of expressions represent the different operations that can be performed in code: binary operations, such as addition; unary operations, such as taking the length of an array; method calls; constructor calls; and so forth. The `System.Linq.Expressions` namespace contains the various classes that represent expressions. All of them derive from the `Expression` class, which is abstract and mostly consists of static factory methods to create instances of other expression classes. The simplest possible example is defining an addition of two integer constants:

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);
Console.WriteLine(add);
```

The graph below shows the tree in a graphical way.



The above example become s more useful and impressive when we see that expression trees can be used as building blocks to create, and then compile and execute a delegate (lambda expression):

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);
Expression.Lamda<Func<int>> lambda = Expression.Lambda<Func<int>>(add);
Func<int> compiled =  lambda.Compile();
Console.WriteLine(compiled());
```

Though powerful and impressive, this way of working with expression trees is prone to errors and somewhat long in the tooth. Luckily, the C# compiler can do a lot of the "heavy lifting" for us. Let's use the compiler to do most of the tedious work above:

```
Expression<Func<int>> lambda = () => 2+3;
Func<int> compiled = lambda.Compile();
// Outputs 5 to the console
Console.WriteLine(compiled());
```

Of course, these are simple examples. You can have much more complex ones, as nearly any lambda expression can be converted, either programmatically or by compiler, into an expression tree. There is one important caveat there: the lambda expression must evaluate into a single expression. That means that you cannot convert lambdas with a body (multi-statement lambdas).

Using LINQ query expressions

Before we go into the details of the LINQ operators and expressions, we need to define a data source we will work with. The code sample below shows off two sample types, `Defect` and `User`, as well as a sample class `SampleData` that exposes two properties that are sequences.

```
// Defect
public class Defect {
      public int ID {get; set; }
      public string Title {get; set; }
      public int Severity {get; set; }
      public User User {get; set; }
}

// User
public class User {
      public int ID {get; set; }
      public string Name {get; set; }
}

// Sample defects
public class SampleData {
      public List<Defect> Defects {get; set; } // all defects
      public List<User> Users {get; set; } // all users

      public SampleData()
      {
            this.Defects = new List<Defect> {
                  new Defect { ID = 1, Title = "Defect 1", Severity= 0 },
                  new Defect { ID = 2, Title = "Defect 2", Severity=4 } };

            this.Users = new List<User> { new User { ID = 1, Name = "Test User" } };
      }
}
```

Consider the following LINQ expression that uses the data source we defined earlier:

```
var result = from defect in SampleData.Defects
             select defect.Title;
```

This query will simply return each element from the sequence (called a source) into a new sequence named result in the above example. Each query expression in LINQ starts in the same way, by defining a source sequence and a range variable. In the above example, a range variable is the defect variable. It will exist inside the query expression, and is an important concept because it is used further in other LINQ operators. The source is any sequence that LINQ can stream through.

All of the query expressions above will be translated into method calls. These methods are just extension methods that are implemented on top of `IEnumerable<T>`. That means that any sequence will have

them, provided that appropriate namespace is imported. The above query expression will be rewritten into the following series of method calls by the compiler:

```
var result = SampleData.Defects.Select(defect => defect);
```

The parameter to the `Select` method (LINQ operator) will be *a lambda expression* that will encapsulate the range variable and determine its type. This way, we can use it to filtering, aggregations, joins between sequences etc.

One aspect of LINQ that is important to note at the outset is *deferred execution*. This means that once the query is defined, it will not be executed immediately and no data will be returned immediately. In order to execute the query and return data, the resulting sequence needs to be iterated over. This can be done by using the `foreach` operator, or by converting the result into an explicit collection (usually via the `ToList()` or `ToArray()` methods).

Projections

Projections or returning data from LINQ queries is done using the select expression. This expression gets translated into a call to the `Select` method with a lambda expression that uses the range variable and returns certain items or a completely new type (anonymous or otherwise).

```
// return just the title of the Defect
var result = from defect in SampleData.Defects
             select defect.Title;
```

The above example will return a new sequence of strings (`IEnumerable<string>` to be precise). We can also use projections to return completely new types. Let us construct a new type that will have just the `ID` and the `Title` of the defect:

```
// return the title of the Defect and the ID
var result = from defect in SampleData.Defects
             select new { ID = defect.ID, Title = defect.Title };
```

It is important to note *degenerate queries*. For example, the following query expression just selects all the defects in the system:

```
var result = from defect in SampleData.Defects
             select defect
```

This is known as a *degenerate query expression*. The compiler deliberately generates a call to `Select` even though it seems to do nothing:

```
SampleData.AllDefects.Select(defect => defect)
```

There's a big difference between this and using SampleData.AllDefects as a simple expression though. The items returned by the two sequences are the same, but the result of the Select method is just the sequence of items, not the source itself. The result of a query expression is never the same object as the source data, unless the LINQ provider has been poorly coded. This can be important from a data integrity point of view—a provider can return a mutable result object, knowing that changes to the returned data sequence won't affect the master even in the face of a degenerate query.

Predicates (filtering)

In order to perform filtering, we use the where query expression. The compiler translates this into a call to the Where method with a lambda expression, which uses the appropriate range variable as the parameter and the filter expression as the body. The filter expression is applied as a predicate to each element of the incoming stream of data, and only those that return true are present in the resulting sequence. Using multiple where clauses results in multiple chained `Where` calls—only elements that match all of the predicates are part of the resulting sequence.

```
// using query expressions, filter the Defects by severity
var result = from defect in SampleData.Defects
             where defect.Severity > 1
             select defect;

// same as above, using methods
var result2 = SampleData.Defects.Where(defect=>defect.Severity>1).Select(defect=>defect);
```

Another way to return data is to return either the first element that satisfies a certain predicate or just one element of the sequence that satisfies a certain predicate.

Grouping and Ordering

Grouping is largely intuitive, and LINQ makes it simple. To group a sequence in a query expression, all you need to do is use the group ... by clause.

Ordering is achieved using the `orderby` operator. The query can be ordered by one or several properties, in either ascending or descending order.

```
// order the defects by Severity from highest to lowest
var result = from defect in SampleData.Defects
             orderby defect.Severity descending
             select defect.ID;

foreach (vart item in result){
      Console.WriteLine(item);
}
```

Joining

Joining is one of the concepts that are at the heart of SQL language, and it is one concept that most users of a relational database have come across. What it does, in short, is that it joins two sets by matching values of certain elements of the set. LINQ joins work in a similar fashion, only they work on sequences instead of tables (like a relational store). There are several types of joins in LINQ. We will cover only one, the inner join that is closest one to the SQL join.

Inner joins involve two sequences. One key selector expression is applied to each element of the first sequence and another key selector (which may be totally different) is applied to each element of the second sequence. The result of the join is a sequence of all the pairs of elements where the key from the first element is the same as the key from the second element. Two sequences being joined can be same or different; the only thing that must match is that the key selector must result in the same type for both keys.

```
//join sample
var joined = from user in SampleData.Users
             join defect  in SampleData.Defects
             on user.ID equals defect.User.ID
             select new { user.ID, defect.Title };

foreach (var item in joined) {
```

```
      Console.WriteLine("{0}\t{1}", item.ID, item.Title);
}
```

Using LINQ to SQL

In order to use LINQ to SQL, we need to define a database and some data in it. To make matters simple, we will assume the same schema (shape of the data) as our examples so far. The database has one table called Defects, and it looks like below:

To work with this database from LINQ to SQL, we need to define a *data context* first, and then a type that will represent our table in the code. We can use the same type we've used so far. Data context in this sense is a gate towards the database; it will expose sequences that represent tables as properties, and will keep track of changes to the objects so it can later update the data. All of this is beyond the scope of this lesson; we will use LINQ to SQL to illustrate the translation of LINQ query expressions into different languages, in this case SQL language.

Let us revisit the previous example of finding all of the defects with Severity higher than 1:

```
// note that we are using data context here
var result = from defect in dataContext.Defects
             where defect.Severity > 1
             select defect;

// write out results
foreach (Defect item in result){
      Console.WriteLine(item.Title);
}
```

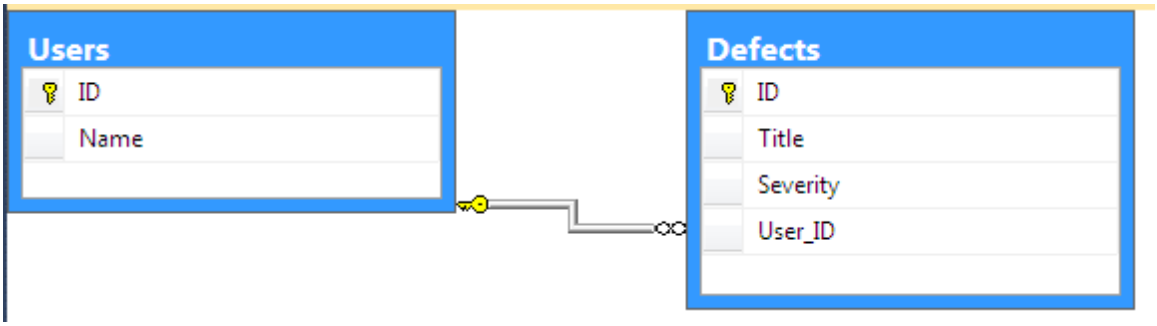
It pays to draw closer attention to what is happening here. In the example above, the query looks similar, if not exactly the same as in previous examples. However, the LINQ to SQL implementation of the where operator will not "simply" take the lambda expression passed in and apply it to the sequence. First of all,

there is no sequence yet; it first needs to connect to the database to get the data. However, pulling this data into the working memory could be potentially highly ineffective as it cannot know the size of the final data set. Also, relational store (Microsoft SQL Server in this case) have their own query language, which means that the predicate can be pushed to the data store for evaluation.

In order to do this, the Where method in LINQ to SQL does not take as a parameter a `Func<bool,TSource>` delegate, but rather an Expression of type `Expression<Func<bool,TSource>>`. Treating programming code as data here allows the LINQ to SQL to parse the Expression, and then translate the encapsulated predicate into an equivalent SQL statement and then send that to the server. In this, rather simple, case, the resulting SQL statement will be

```
SELECT * FROM Defects WHERE Severity > 1
```

This workflow is the same for more complex operators as well. Let us take a look at the join operator in LINQ. For this, we will need to add another table to the database that will be tied to the Defect one. Again, in order to avoid complexity, I will add a table for the type we are already using within code. The diagram is shown below.



The table Users will be accessible as a sequence off the data context, similar to the Defects sequence. Let us see two ways in which we can perform a join.

First, consider the explicit join:

```
      var result = from defect in dataContext.Defects
                   join user in dataContext.Users on defect.User_ID equals user.ID
                   select new { defect.Title, defect.User.Name };

      // show the generated SQL statement on standard output
      dataContext.Log = Console.Out;

      foreach (var item in result) {
            Console.WriteLine(item);
      }
```

This join looks a lot like the one we did with collections (in memory). We can see the SQL statement that was generated by this operation:

```
SELECT [t0].[Title], [t2].[Name]
FROM [dbo].[Defects] AS [t0]
INNER JOIN [dbo].[Users] AS [t1] ON [t0].[User_ID] = [t1].[ID]
INNER JOIN [dbo].[Users] AS [t2] ON [t2].[ID] = [t0].[User_ID]
```

There is also an implicit join:

```
      var result = from defect in dataContext.Defects
                   select new { defect.Title, defect.User.Name };

      // show the generated SQL statement on standard output
      dataContext.Log = Console.Out;

      foreach (var item in result) {
            Console.WriteLine(item);
      }
```

The difference here is staggering. It is only one line. We have omitted the explicit join operator and just left the projection of a property that forms a relationship between these two tables in the database. The same code will work, and LINQ to SQL is "smart" enough to realize that it needs a join in order to reach the needed data.

```
SELECT [t0].[Title], [t1].[Name]
FROM [dbo].[Defects] AS [t0]
INNER JOIN [dbo].[Users] AS [t1] ON [t1].[ID] = [t0].[User_ID]
```

Most LINQ "flavors" (also known as "providers") that move away from working with in-memory sequences and try to provide query expressions' support on top of other data sources follow this pattern. They work with expression trees that they then translate into the needed target language. That language can be anything, really. For instance, there is an implementation of LINQ for Active Directory, that is, LDAP servers; it will translate most of the LINQ operators into LDAP queries to find the users in directories etc.

Recap

The purpose of this lesson was to introduce LINQ operators, and to show basic query expressions and how to work with them. LINQ greatly simplifies working with sequence-based data in C# program. IT can also be used, via its extensibility, to access other data sources that don't necessarily have to reside in memory of the currently executing program; we saw a brief example of this with LINQ to SQL. By using extensions to C# language like anonymous types, type inference and expression trees LINQ enables the programmer to start writing code in a more declarative manner.