Concord Spark Progress

C# - Constructing LINQ Statements

Introduction to LINQ

C# has many unique features that make it flexible and powerful. One of these features is LINQ – short for Language Integrated Query. This is built right into the C# language, itself – it functions as merely another component of this language.

If you have ever worked with databases, you've probably heard of something called SQL – Structured Query Language. SQL is simply a language where you can write statements that specify what *you want* from the database. SQL is a *declarative* language; you use it to state what you want, but *not how* to get it. Opposite to this idea, is an *imperative* language – such as C# - where you have to specify *how* to retrieve the data and operate on it.

A simple example that shows the difference between a declarative and an imperative language is given below:

"Give me all the hours worked by the employees who have an annual salary over $50,000."

The above statement represents a declarative statement – you say what you want.

"Go to the human resources department, access Workstation 2A, download File workhours.dat onto a USB, then deliver it to my office."

This statement represents an *imperative* language – you specify the steps it takes to do something. Programming languages are imperative languages.

With all this said, C#'s LINQ was based on SQL. SQL coders will definitely be able to make themselves feel right at home in very little time. LINQ can also work with arrays, collections and databases. In our examples in this material, we'll use arrays.

To use LINQ, be sure to have a using directive for namespace `System.Linq`.

LINQ Examples

Suppose, we have an array of integers, defined like this:

```
int[] numbers = { 100, 25, 33, 44, 66, 77, 88, 99 };
```

Let's start with a LINQ query which will select all the numbers from this array:

```
var myNumbers = from number in numbers
                select number;
```

This statement is divided into two parts by the assignment operator - =. The left-hand portion declares a reference variable – `myNumbers` – which will hold a reference to a collection with the LINQ results. We will explain the `var` keyword later.

> A collection in C# is merely an object that can contain multiple objects of the same datatype. `List<T>` is one such type – `List<T>` can also grow according to the number of objects you add.

The right-hand portion is our LINQ *query*. We begin *every LINQ query* with the keyword `from`. Next, we specify a *range* variable. A range variable will represent each item in the data source – `numbers` – in this case. We use the range variable to refer to each element in the data source as we will show in more examples. The range variable has the same datatype as the data source. Since `numbers` contains `int` data, `number` is an `int` variable, as well.

After the range variable, we specify our data source from which we wish to select data from. We do this by coding the keyword `in` followed by the name of the data source.

Lastly, we select the desired data into the list of results by using a select clause. A select clause starts with the keyword `select` followed by the data you wish to add into the results. Typically, a LINQ query will end with a select clause.

Finally, the entire statement is capped off with a semicolon.

Our results, in this case, will contain all the numbers from the array. Normally, we use LINQ to select data with certain criteria as we will show below.

```
IEnumerable<int> numbersGreater = from number in numbers
                                  where number > 50
                                  select number;
```

For starters, let's discuss why we declared the reference to the list of results as `IEnumerable<int>` instead of using the keyword `var`. Every LINQ query you write will generate all the results in a collection of type `IEnumerable<T>` where T is replaced with the type of data you queried – in our case, `int`. When we use the keyword `var` to declare the reference, the compiler *infers* the type. You can use the `var` keyword in many other situations, not just with LINQ query results.

The next thing you may have noticed is the where clause. The where clause begins with the keyword `where`, followed by a condition. Any item in the data source that satisfies the condition will be selected as part of the result. In this case, we're selecting all the values that are greater than 50.

Here's another example:

```
IEnumerable<int> numbersGreater = from number in numbers
                                  where number > 50
                                  orderby number descending
                                  select number;
```

The third line in the LINQ query is what we're interested in; this is an `orderby` clause. Here, we can specify to order the number in either descending or ascending order. Not all types can be ordered as simply as simple data types – such as `decimal`, `char` or `int`. To sort in ascending order, simply omit the keyword `descending` from the `orderby` clause.

If the LINQ query's *range* variable refers to a more complicated object, you can use it to access individual properties, as shown below:

```
var employeesWithHighSalary = from employee in employeeList
                              where employee.YearlySalary >= 30000.45M
                              select employee;
```

Assuming we have defined a class – `Employee` – with instances of it in `employeeList`, the above example will do the following:

- Select all `Employees` whose `YearlySalary` is greater than $30,000.45.

LINQ is a very broad topic, a good source of information would be the MSDN site itself:

MSDN - Introduction to LINQ Queries (C#)
MSDN - LINQ Explanation

Always do your own research, and remember the best way to learn coding is to code something!

Concord Spark Tutoring - Facebook