Extension Methods

The compiler converts the LINQ query to invoke method calls instead of the LINQ query. LINQ offers various extension methods for the `IEnumerable<T>` interface, so you can use the LINQ query across any collection that implements this interface. An *extension method* is defined as a static method whose first parameter defines the type it extends, and it is declared in a static class.

One of the classes that define LINQ extension methods is `Enumerable` in the namespace `System.Linq`. You just have to import the namespace to open the scope of the extension methods of this class. A sample implementation of the `Where` extension method is shown in the following code. The first parameter of the `Where` method that includes the `this` keyword is of type `IEnumerable<T>`. This enables the `Where` method to be used with every type that implements `IEnumerable<T>`. A few examples of types that implement this interface are arrays and `List<T>`. The second parameter is a `Func<T, bool>` delegate that references a method that returns a Boolean value and requires a parameter of type `T`. This predicate is invoked within the implementation to examine whether the item from the `IEnumerable<T>` source should be added into the destination collection. If the method is referenced by the delegate, the `yield return` statement returns the item from the source to the destination:

```
public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, bool> predicate)
{
  foreach (TSource item in source)
    if (predicate(item))
      yield return item;
}
```

NOTE *A predicate is a method that returns a Boolean value.*
Because `Where` is implemented as a generic method, it works with any type that is contained in a collection. Any collection implementing `IEnumerable<T>` is supported.
NOTE The extension methods here are defined in the namespace `System.Linq` in the assembly `System.Core`.

Deferred Query Execution

During runtime, the query expression does not run immediately as it is defined. The query runs only when the items are iterated.
Let's have a look once more at the extension method `Where`. This extension method makes use of the `yield return`statement to return the elements where the predicate is true. Because the `yield return` statement is used, the compiler creates an enumerator and returns the items as soon as they are accessed from the enumeration:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                      Func<T, bool> predicate)
{
```

```
  foreach (T item in source)
  {
    if (predicate(item))
    {
     yield return item;
    }
  }
}
```

This has a very interesting and important effect. In the following example a collection of `string` elements is created and filled with first names. Next, a query is defined to get all names from the collection whose first letter is `J`. The collection should also be sorted. The iteration does not happen when the query is defined. Instead, the iteration happens with the `foreach` statement, where all items are iterated. Only one element of the collection fulfills the requirements of the `where`expression by starting with the letter J: `Juan`. After the iteration is done and `Juan` is written to the console, four new names are added to the collection. Then the iteration is done again:

```
var names = new List<string> {"Nino","Alberto","Juan","Mike","Phil" };

var namesWithJ = from n in names
                 where n.StartsWith("J")
                 orderby n
                 select n;

WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  WriteLine(name);
}
WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  WriteLine(name);
}
```

Because the iteration does not happen when the query is defined, but does happen with every `foreach`, changes can be seen, as the output from the application demonstrates:

```
First iteration
Juan

Second iteration
Jack
Jim
John
Juan
```

Of course, you also must be aware that the extension methods are invoked every time the query is used within an iteration. Most of the time this is very practical, because you can detect changes

in the source data. However, sometimes this is impractical. You can change this behavior by invoking the extension methods `ToArray`, `ToList`, and the like. In the following example, you can see that `ToList` iterates through the collection immediately and returns a collection implementing `IList<string>`. The returned list is then iterated through twice; in between iterations, the data source gets new names:

```
var names = new List<string> {"Nino","Alberto","Juan","Mike","Phil" };
var namesWithJ = (from n in names
                  where n.StartsWith("J")
                  orderby n
                  select n).".ToList();"

WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  WriteLine(name);
}
WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  WriteLine(name);
}
```

The result indicates that in between the iterations the output stays the same although the collection values have changed:

```
First iteration
Juan

Second iteration
Juan
```