What is LINQ?

Language-Integrated Query, or LINQ, is a way to query a set of data with the use of *extension methods*. These extension methods can only be accessed by adding the `using System.Linq;` statement. In the following examples, you'll see how to use LINQ on a `List` of `Person` objects. The following material builds upon the Working with Arrays and Collections lesson and the Extension Methods section of the Defining and Calling Methods lesson.

As you follow along in these examples, use this `List<Person>` collection and `Person` class:

```
public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        // Write your code here
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation = "Dev"
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation = "Manag
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation = "De
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev", Ag
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation = "De

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Finding Items in Collections

At some point, you're going to need to find items in a collection that meet specific criteria. LINQ provides you with extension methods like `Where`, `Skip`, and `Take` to make this easy.

Where

The `Where` extension method is the most commonly used. It can be used to filter elements from a collection based on certain criteria. For example, say you wanted to filter the list of people based on whether or not they are above the age of 30, it would look something like this:

```
//There will be two Persons in this variable: the "Steve" Person and the "Jane" Person
var peopleOverTheAgeOf30 = people.Where(x => x.Age > 30);
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        //There will be two Persons in this variable: the "Steve" Person and the "Jane"
Person
        var peopleOverTheAgeOf30 = people.Where(x => x.Age > 30);
        foreach (var person in peopleOverTheAgeOf30)
        {
            Console.WriteLine(person.FirstName);
        }
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
```

```
    public int Age { get; set; }
}
```

As you can see, the `Where` method takes in a lambda expression as a *predicate* (a statement evaluating to either `true` or `false`) to be applied to each item in the list of people. In this scenario, every person's `Age` property is checked to see if it is greater than 30. If the result of the expression is `true`, the current item is added to an an object of type `IEnumerable<T>`. This new `IEnumerable<T>` will be an "IEnumerable of Person" `IEnumerable<Person>`, because the `List` it came from was made of `Person` objects.

Skip

Sometimes you will want to ignore the first items in a collection and include only what remains. You can do this by using the aptly-named `Skip` method.

```
//Will ignore Eric and Steve in the list of people
IEnumerable<Person> afterTwo = people.Skip(2);
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        //Will ignore Eric and Steve in the list of people
        IEnumerable<Person> afterTwo = people.Skip(2);
        foreach (var person in afterTwo)
        {
            Console.WriteLine(person.FirstName);
        }
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });
```

```
        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Take

The opposite of the `Skip` extension method is the `Take` extension method. `Take` will return the first items in the collection including a number of items equal to the number passed as an argument to the method.

```
//Will only return Eric and Steve from the list of people
IEnumerable<Person> takeTwo = people.Take(2);
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        //Will only return Eric and Steve from the list of people
        IEnumerable<Person> takeTwo = people.Take(2);
        foreach (var person in takeTwo)
        {
            Console.WriteLine(person.FirstName);
        }
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
```

```
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Note {.note}
You can *chain* the `Skip` and `Take` methods to skip one or more items and take the next ones.

Changing Each Item in Collections

Sometimes the items in your collections will have too many or not enough properties; at other times those items will just be of the wrong type. Using the `Select` method from LINQ, you can change the *type* of the items in your collections by creating new items or selecting one member of the items. In the following example, `Select` is used to select the `FirstName` from each `Person` in the `List<Person>` collection. This will return a collection of `string` objects in the form of an `IEnumerable<string>` .

```
IEnumerable<string> allFirstNames = people.Select(x => x.FirstName);
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        IEnumerable<string> allFirstNames = people.Select(x => x.FirstName);
        foreach (var firstName in allFirstNames)
        {
            Console.WriteLine(firstName);
        }
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();
```

```
        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

The contents of `allFirstNames` will now contain the strings `"Eric", "Steve", "Brendan", "Jane", "Samantha"` .

You can also create a brand new object based on properties you select. For this example, I'll create another model `FullName` and set the `First` and `Last` properties based on the `FirstName` and `LastName` properties of the `Person` model.

```
public class FullName
{
    public string First { get; set; }
    public string Last { get; set; }
}
```

The below line is how we can create a new `FullName` object for each `Person` in the `people` list.

```
IEnumerable<FullName> allFullNames = people.Select(x => new FullName { First =
x.FirstName, Last = x.LastName });
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
```

```
        var people = GenerateListOfPeople();

        IEnumerable<FullName> allFullNames = people.Select(x => new FullName { First =
x.FirstName, Last = x.LastName });
        foreach (var fullName in allFullNames)
        {
            Console.WriteLine($"{fullName.Last}, {fullName.First}");
        }
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}

public class FullName
{
    public string First { get; set; }
    public string Last { get; set; }
}
```

Finding One Item in Collections

Up to this point, our results are always collections of objects. While useful, there will be times you will want to get just one object from the collection. In this lesson, you will see the `FirstOrDefault`, `LastOrDefault`, and `SingleOrDefault` extension methods from LINQ. These are three of the more commonly used LINQ methods for obtaining one item from a collection.

FirstOrDefault

The `FirstOrDefault` extension method will return the first element of a set of data. If there are no elements that match your criteria, the result will be the default value type for the object (usually `null`).

```
Person firstOrDefault = people.FirstOrDefault();
Console.WriteLine(firstOrDefault.FirstName); //Will output "Eric"
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        Person firstOrDefault = people.FirstOrDefault();
        Console.WriteLine(firstOrDefault.FirstName);
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

The `FirstOrDefault` method can also be used to filter the list to return the first element that matches your expression's criteria.

```
var firstThirtyYearOld1 = people.FirstOrDefault(x => x.Age == 30);
var firstThirtyYearOld2 = people.Where(x => x.Age == 30).FirstOrDefault();
Console.WriteLine(firstThirtyYearOld1.FirstName); //Will output "Brendan"
Console.WriteLine(firstThirtyYearOld2.FirstName); //Will also output "Brendan"
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        var firstThirtyYearOld1 = people.FirstOrDefault(x => x.Age == 30);
        var firstThirtyYearOld2 = people.Where(x => x.Age == 30).FirstOrDefault();
        Console.WriteLine(firstThirtyYearOld1.FirstName); //Will output "Brendan"
        Console.WriteLine(firstThirtyYearOld2.FirstName); //Will also output "Brendan"
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

The two expressions above return the same item whether the predicate is in the `Where` or the `FirstOrDefault`. Skipping the `Where` is more concise, and can improve readability of your code.

Note {.note}
The "OrDefault" means, if no elements in the queried data match the expression passed into the method, the returning object will be `null`.

```
List<Person> emptyList = new List<Person>(); // Empty collection
Person willBeNull = emptyList.FirstOrDefault(); // None - default of null used

List<Person> people = GenerateListOfPeople();
Person willAlsoBeNull = people.FirstOrDefault(x => x.FirstName == "John");  No John -
default of null used

Console.WriteLine(willBeNull == null); // true
Console.WriteLine(willAlsoBeNull == null); //true



using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        List<Person> emptyList = new List<Person>();
        Person willBeNull = emptyList.FirstOrDefault();

        List<Person> people = GenerateListOfPeople();
        Person willAlsoBeNull = people.FirstOrDefault(x => x.FirstName == "John");

        Console.WriteLine(willBeNull == null); // true
        Console.WriteLine(willAlsoBeNull == null); //true
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
```

```
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Note {.note}
There is a `First` extension method that will function the same as `FirstOrDefault` but will return a `System.InvalidOperationException` if there are no elements that match your criteria.

LastOrDefault

The `LastOrDefault` extension method works the same way that `FirstOrDefault` does, however, as the name suggests, instead of returning the first item of the collection, it returns the last.

```
Person lastOrDefault = people.LastOrDefault();
Console.WriteLine(lastOrDefault.FirstName); //Will output "Samantha"
Person lastThirtyYearOld = people.LastOrDefault(x => x.Age == 30);
Console.WriteLine(lastThirtyYearOld.FirstName); //Will output "Brendan"
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        Person lastOrDefault = people.LastOrDefault();
        Console.WriteLine(lastOrDefault.FirstName);
        Person lastThirtyYearOld = people.LastOrDefault(x => x.Age == 30);
        Console.WriteLine(lastThirtyYearOld.FirstName);
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });
```

```
        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Note {.note}
As `First` is to `FirstOrDefault`, there is also a `Last` method, which will return a `System.InvalidOperationException` if there are no elements that match your criteria.

SingleOrDefault

The `SingleOrDefault` extension method will return the only occurrence of an item matching your expression. If none match your criteria, the default value of your type will be returned. `SingleOrDefault` functions much like `FirstOrDefault`, but if more than one item matches your predicate, a `System.InvalidOperationException` will be thrown.

```
Person single = people.SingleOrDefault(x => x.FirstName == "Eric"); //Will return the
Eric Person obejct
Person singleDev = people.SingleOrDefault(x => x.Occupation == "Dev"); //Will throw the
System.InvalidOperationException
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        Person single = people.SingleOrDefault(x => x.FirstName == "Eric");
        Console.WriteLine(single.FirstName);
        // Uncomment the next line to see it throw an exception
        // Person singleDev = people.SingleOrDefault(x => x.Occupation == "Dev");
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
```

```
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Note {.note}
The `Single` extension method functions the same as `SingleOrDefault`, however, in addition to throwing an exception for too many items matching the expression, it will also throw a `System.InvalidOperationException` if there are no items matching.

Finding Data About Collections

There are extension methods that allow you to determine if or how many items will satisfy an expression.

Count

The `Count` extension method returns the number of items in the data over which you're iterating as an `int`.

```
int numberOfPeopleInList = people.Count(); //Will return 5
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();
```

```
        int numberOfPeopleInList = people.Count();
        Console.WriteLine(numberOfPeopleInList);
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

This method will also allow you to pass in a predicate expression, and return the number of matching items as an `int`.

```
int peopleOverTwentyFive = people.Count(x => x.Age > 25); //Will return 3
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        int peopleOverTwentyFive = people.Count(x => x.Age > 25);
        Console.WriteLine(peopleOverTwentyFive);
    }

    public static List<Person> GenerateListOfPeople()
```

```
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Any

The `Any` extension method checks a set of data and return a boolean value indicating whether any items in the collection match your predicate. This is commonly used when checking if a list has any elements before performing some other action on the list.

```
bool thereArePeople = people.Any(); //This will return true
bool thereAreNoPeople = emptyList.Any(); //This will return false
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();
        var emptyList = new List<Person>();

        bool thereArePeople = people.Any();
        Console.WriteLine(thereArePeople);
        bool thereAreNoPeople = emptyList.Any();
        Console.WriteLine(thereAreNoPeople);
    }
```

```
    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Note {.note }
Using `Any` is a clearer way of checking if elements exist than checking if the count of a list is greater than 0 as shown in this example:

```
if (people.Count() > 0) //This works
{
    //perform some action(s)
}
if (people.Any()) //This is better
{
    //perform some action(s)
}
```

All

The `All` extension method is similar to `Any`, but it returns a boolean only when all elements in the collection satisfy your expression.

```
bool allDevs = people.All(x => x.Occupation == "Dev"); //Will return false

bool everyoneAtLeastTwentyFour = people.All(x => x.Age >= 24); //Will return true
```

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var people = GenerateListOfPeople();

        bool allDevs = people.All(x => x.Occupation == "Dev");
        Console.WriteLine(allDevs);
        bool everyoneAtLeastTwentyFour = people.All(x => x.Age >= 24);
        Console.WriteLine(everyoneAtLeastTwentyFour);
    }

    public static List<Person> GenerateListOfPeople()
    {
        var people = new List<Person>();

        people.Add(new Person { FirstName = "Eric", LastName = "Fleming", Occupation =
"Dev", Age = 24 });
        people.Add(new Person { FirstName = "Steve", LastName = "Smith", Occupation =
"Manager", Age = 40 });
        people.Add(new Person { FirstName = "Brendan", LastName = "Enrick", Occupation =
"Dev", Age = 30 });
        people.Add(new Person { FirstName = "Jane", LastName = "Doe", Occupation = "Dev",
Age = 35 });
        people.Add(new Person { FirstName = "Samantha", LastName = "Jones", Occupation =
"Dev", Age = 24 });

        return people;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Occupation { get; set; }
    public int Age { get; set; }
}
```

Converting Results to Collections

The results of many of the LINQ queries so far has been `IEnumerable<T>`, however, you sometimes want a different type than that. LINQ provides you with extension methods that allow you to convert collections to other collection types, and these methods work with any type implementing an `IEnumerable` interface.

ToList

The `ToList` extension method allows you to convert an `IEnumerable<T>` to a `List<T>`, where `T` will be the same type received. When you filter a list using the `Where` extension method, the result is an `IEnumerable<T>`. This is fine if you just want to iterate over the items once, however, you will often need to manipulate those items. This is where the `ToList` extension method comes in to play.

```
List<Person> listOfDevs = people.Where(x => x.Occupation == "Dev").ToList();
```

ToArray

Similar to `ToList`, there is also a `ToArray` extension method that will result in an array rather than a list.

```
Person[] arrayOfDevs = people.Where(x => x.Occupation == "Dev").ToArray(); //This will retur
```