Libraries, Collections, Lambdas & LINQ and Generics

Libraries

Throughout the examples used with **dotnetfiddle.net** the **using** has been put at the top to include a **namespace**, this is taking advantage of the **.NET Framework** which features many kinds of "**namespace**" that can be used in an application that add features both complex and simple so many things don't need to be done again if they've been done already and many of these are contained in the **.NET Framework class** Libraries such as **System** and **System.Text** used in previous examples.

To use another **namespace** in **dotnetfiddle.net** enter the following:

```
using System;
using System.Text;
using System.Xml;

public class Demo
{
	public class Person
	{
		public string Forename { get; set; }
		public string Surname { get; set; }
	}

	public static void Main()
	{
		Person item = new Person()
		{
			Forename = "John",
			Surname = "Smith"
		};
		StringBuilder output = new StringBuilder();
		using(XmlWriter writer = XmlWriter.Create(output))
		{
			writer.WriteStartElement("person");
			writer.WriteStartAttribute("forename");
			writer.WriteValue(item.Forename);
			writer.WriteEndAttribute();
			writer.WriteStartAttribute("surname");
			writer.WriteValue(item.Surname);
			writer.WriteEndAttribute();
			writer.WriteEndElement();
		}
		Console.WriteLine(output.ToString());
	}
}
```

Namespaces used in the previous examples were **System**, **System.Text** and **System.Xml** – the first is where the main and common features such as the **Console** Methods that have been used like **Console.WriteLine**. **System.Text**" is used for **StringBuilder** which has also been used before, in this case it's used in combination with **System.Xml** to create an **XmlWriter** – this is used to create an XML document which is a way of storing data that can be read easily by a program or in the case written. There's another use of **using** here to create the **XmlWriter** to make sure it's created at the top part and then closed or Disposed when finished.

Another example is to use **XmlReader** to read in XML and take advantage of another **namespace** which is **System.IO** which allows input and output methods to be used in this case a **StringReader** to read some existing XML as an input for an application.

In **dotnetfiddle.net** enter the following example:

```
using System;
using System.IO;
using System.Xml;

public class Demo
{
	public class Person
	{
		public string Forename { get; set; }
		public string Surname { get; set; }
	}

	public static void Main()
	{
		Person item = new Person();

		string input =
		"<person forename='John' surname='Smith'/>";

		using(XmlReader reader = XmlReader.Create(
		new StringReader(input)))
		{
			reader.ReadToFollowing("person");
			item.Forename = reader.GetAttribute("forename");
			item.Surname = reader.GetAttribute("surname");
		}
		Console.WriteLine("Forename:" + item.Forename);
		Console.WriteLine("Surname:" + item.Surname);
	}
}
```

Collections

It's possible to store multiple values of the same type, mentioned previously in the form of an Array, these are normally declared with a fixed size and denoted using square brackets and can be used to store a sequence of values of that **type**.

To use an Array in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            string[] letters = { "A", "B", "C", "D", "E", "F" };
            foreach(string letter in letters)
            {
                  Console.WriteLine(letter);
            }
      }
}
```

It is also possible to loop through an Array by Index where the first element is zero.

To loop through an Array this way, enter **dotnetfiddle.net** the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            string[] letters = { "A", "B", "C", "D", "E", "F" };
            for(int index = 0; index < letters.Length; index++)
            {
                  Console.WriteLine(letters[index]);
            }
      }
}
```

Arrays are built into **C#** and can be utilised for many things but they have limitations that their size must be known in advance and often you won't know what to store beforehand, there is another way to store multiple items is with a collection.

There's another way of storing items of a **type**, this is known as a Collection and there are two main kinds which are **List** and **Dictionary**.

The first type of collection is **List** where it is a list of a particular item so if it's a **List** of **int** this would be **List<int>** where the **type** of the list is within angle brackets, they are more flexible than an Array as it can have as many items as you want and Collections have their own **namespace** which is **System.Collections.Generic**.

To use a "List" of "int" in **dotnetfiddle.net**, enter the following:

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

public class Demo
{
	public static void Main()
	{
		List<int> numbers = new List<int>();
		for(int index = 1; index <= 10; index++)
		{
			numbers.Add(index);
		}
		foreach(int number in numbers)
		{
			Console.WriteLine(number);
		}
	}
}
```

It's also possible to use a **List** of **string** in **dotnetfiddle.net**, enter the following:

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

public class Demo
{
	public static void Main()
	{
		List<string> letters = new List<string>()
		{ "A", "B", "C", "D", "E", "F" };
		foreach(string letter in letters)
		{
			Console.WriteLine(letter);
		}
	}
}
```

In the first example, there is a **List<int>** of numbers which is added to using the **Add** Method to add something to the **List** within a **for** loop. Then in the second **foreach** loop the items that were added are output. Then in the second example there is a **List<string>** of letters which have been prepopulated, like the Array and then the contents are output from a **foreach** loop.

The second type of collection is **Dictionary** which is like **List** but has two parts, there is the **Key** which will identify something that has been added, and **Value** which is the item that's been added and like **List** the **Dictionary** can have many types of **Value** but can also have types of **Key** but mainly **string** is the most commonly used **type** for a **Key** and you can get values of a **type** by their **Key**.

To use a "**Dictionary**" by **string** of **string** in **dotnetfiddle.net**, enter the following:

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

public class Demo
{
      public static void Main()
      {
            Dictionary<string, string> colours =
                  new Dictionary<string, string>();
            colours.Add("R", "Red");
            colours.Add("G", "Green");
            colours.Add("B", "Blue");
            foreach(string key in colours.Keys)
            {
                  Console.WriteLine(colours[key]);
            }
      }
}
```

In the example, there's a **Dictionary** of **string** where the **Key** is also a **string**, it has an **Add** Method with the **Key** and then the **Value** there there's loop which uses the **Keys** Property which is a **List** of the **Keys** and then output the item using the **Key** similar to an Array by using the square brackets.

Lambdas & LINQ

C# has a special type of Method that can be used as an alternative to creating another named Method you use but are created in-line with your code, they can be passed as a Parameter to other methods and these are known as Lambda expressions.

To use a Lambda expression in **C#**, in **dotnetfiddle.net** enter the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Func<int, int> addTen = x => x + 10;
            Console.WriteLine(addTen(4));
      }
}
```

In the example, there is a **Func<int, int>** which is a variable the Lambda is assigned to then to the right of the **=** is the Lambda expression and there is two parts, the first is the Parameter passed in and then there is a special operator **=>** pronounced as "produces" which is followed by the code to execute and can be done in one line and doesn't require a **return** statement like a Method that returns a value would normally need.

To use multiple Parameters with a Lambda in **dotnetfiddle.net** by entering the following:

```
using System;

public class Demo
{
      public static void Main()
      {
            Func<int, int, int> area = (x,y) => x * y;
            Console.WriteLine(area(4,5));
      }
}
```

Lambda expressions are a powerful and easy way to access functionality without needing a separate Method and can be reused in many places just like a Method but also passed into a Method like a Value can be.

The language of **C#** is powerful but to add to this power is a way of querying sets and lists of data using **LINQ** or **Language-Integrated Query**, which is a set of Extension Methods which are a special kind of **static** Method to extend existing types with new functionality and in **LINQ** this is used extensively and makes performing what would be complex tasks much easier and just requires the inclusion of another **using** for **System.Linq** and they also make use of Lambda expressions.

The following examples will add to the previous, so to start will need a **class** and then a collection of them as a **List** by entering in to **dotnetfiddle.net** the following:

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

namespace Workshop
{
      public class Person
      {
            public string Forename { get; set; }
            public string Surname { get; set; }
            public int Age { get; set; }
      }

      public class Demo
      {
            public static void Main()
            {
                  List<Person> people = new List<Person>();
                  people.Add(new Person { Forename = "John",
                  Surname = "Smith", Age = 30 });
                  people.Add(new Person { Forename = "Jane",
                  Surname = "Smith", Age = 42 });
                  people.Add(new Person { Forename = "John",
                  Surname = "Doe", Age = 24 });
                  people.Add(new Person { Forename = "Jane",
                  Surname = "Doe", Age = 58 });
            }
      }
}
```

In this example, there is a **class** of **type** of **Person** which has two Properties of **string** and another of **int** then there is a **List** of **Person** which is then added to or populated with some values and this example will be added to in subsequent examples.

LINQ allows you to search for items within a collection that match a set of criteria and is the main component of **Language-integrated Query**, which is to Query data, which can be done with **where**.

To use "**where**" with **LINQ** in **dotnetfiddle.net**, add into **Main** at the end the following:

```
IEnumerable<Person> overFourty = people.Where(x => x.Age > 40);
Console.WriteLine(overFourty.Count());
```

In this part of the example there is the use of **IEnumerable<Person>** which is the type of item that. The **where** uses a Lambda and the expression looks for any item that is in the **List** which has an **Age** value greater than **40** and then outputs this value which should be **2** using another **LINQ** method of **count**.

You may have too many items in the collection or not have enough properties and you can use the **select** Method in **LINQ** which can be a useful way of converting an existing Collection into other collections or values.

To use **select** with **LINQ** in **dotnetfiddle.net**, add below last example the following:

```
IEnumerable<string> foreNames = people.Select(x => x.Forename);
foreach(string foreName in foreNames)
{
      Console.WriteLine(foreName);
}
```

In this part of the example the **select** is used to get the Forename for each **Person** in the Collection and will return a collection of **string** Objects that are then looped though with a **foreach** statement to output each Forename.

You may just want to get one item from a collection and you can do this in **LINQ** with **First**, **Last** and **Single** to get an item or **FirstOrDefault**, **LastOrDefault** and **SingleOrDefault** where **OrDefault** is the default value for a **type** which is usually **null** in case the value being looked for in the collection is not present.

To use **first** & **last** with **LINQ** in **dotnetfiddle.net**, add below last example the following:

```
Person first = people.First();
Person last = people.Last();
Console.WriteLine(first.Surname);
Console.WriteLine(last.Surname);
```

It is also possible to get information about a collection using **LINQ** with **count**, **any** and **all** - **count** was used previously and will return the number of items in the collection but it can also take a Lambda" expression or Predicate like a **where** does so you could count only certain things that match the expression. With **any** this will check the collection and see if anything that matches the Predicate expression matches and will return **true** if it does, and **false** if not and **all** which is like **any** except that all values must match to return **true** otherwise it will return **false**.

To use **count** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.Count(x => x.Age < 50));
```

To use **any** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.Any(x => x.Surname == "Doe"));
```

To use **all** in **dotnetfiddle.net**, add below the last example the following:

```
Console.WriteLine(people.All(x => x.Forename == "John"));
```

The results of many of the **LINQ** queries shown have been of **IEnumerable** of a type such as **int** or **string**, but there may be times where you need a different **type** than **IEnumerable** and **LINQ** has Extension Methods that allow you to do this.

To get a **List** of the type returned by **LINQ** in **dotnetfiddle.net**, enter the following:

```
List<string> list = people.Select(x => x.Forename).ToList();
```

To get an Array of the type returned by **LINQ** in **dotnetfiddle.net**, enter the following:

```
string[] array = people.Select(x => x.Forename).ToArray();
```

LINQ is a very powerful addition to **C#** and there are many more ways you can use it to get various parts of a collection, think of it as a search engine for your code, whatever information is in there, you can get at it with **LINQ** expressions and can use them in combination with each other to make your applications much more powerful than they would otherwise be and makes it easy and straightforward to manipulate data.

Generics

C# contains many powerful features that are useful to allow the creation of software that can reuse or simplify many complex but common programming tasks, as projects become more complex there needs to be a better way to reuse code, to help with this **C#** includes a feature called Generics. Generics allows a **class** to take type parameters allow for more functionality in much the same way as Methods can be made more powerful because they take parameters. Generic Classes and Methods combine reusability and efficiency in a way that non-generic alternatives can't, they are most frequently used with collections in fact **List** and **Dictionary** used previously are an example of Generics. When you encounter the **<** and **>** such as in **List<int>** the **type** within them can be any type which is what makes them Generic – that is they can be any **type** of **List** or **Dictionary** this type is usually indicated by using **T** and referred to as being **type** of **T** where **T** is the **type**.

To use Generics in **C#** enter in to **dotnetfiddle.net** the following:

```
using System;

namespace Workshop
{
      public class Container<T>
      {
            private T _value;

            public Container(T val)
            {
                  this._value = val;
            }

            public T Get()
            {
                  return this._value;
            }
      }

      public class Demo
      {
            public static void Main()
            {
                  Container<string> name =
                        new Container<string>("John Smith");
                  Console.WriteLine(name.Get());
            }
      }
}
```

In the example, there is a Container **class** which takes **type** of **T** denoted by **<T>** which will be the **type**, then a value which is of **type** of **T** and a **Get** Method which returns **type** of **T**. Although this is a simple example the Container **class** could be used with any **type**.

To use another **type** in the previous example below the **Console.WriteLine(name.Get());** line in **dotnetfiddle.net** enter the following:

```
Container<int> a = new Container<int>(4);
Container<int> b = new Container<int>(5);
Console.WriteLine(a.Get() + b.Get());
```

You can also use Generics to combine functionality needed such as creating a **new** instance of a **class** whatever its type might be using with a Factory **class** with a Create Method to create a new instance of the **class**.

To implement a basic Factory class in **C#**, enter in to **dotnetfiddle.net** the following:

```
using System;

namespace Workshop
{
      public class Factory<T> where T : new()
      {
            public T Create()
            {
                  return new T();
            }
      }

      public class Person
      {
            public string Name { get; set; }
      }

      public class Demo
      {
            static Factory<Person> factory =
            new Factory<Person>();

            public static void Main()
            {
                  Person person = factory.Create();
                  person.Name = "John Smith";
                  Console.WriteLine(person.Name);
            }
      }
}
```