How does it work in C#? - Part 3 (C# LINQ in detail)

Language Integrated Query (LINQ) is a Microsoft .NET Framework component that works as a communicator between the objects and the data. This article partially taken from the "Expert C# 5.0" - book and it will explore the different extension methods used in the LINQ using C# and IL code to discuss..

Contents

* [LINQ basics](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdLinqBasics)
  + [Extension method - behind the scene](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdExtensionmethodUnderneath)
* [Extension Methods](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdExtensionMethods)
  + [Where and Select](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdWhereSelect)
  + [All](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdAll)
  + [Any](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdAny)
  + [Average](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdAverage)
  + [Concat](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdConcat)
  + [Contains](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdContains)
  + [Count](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdCount)
  + [DefaultIfEmpty](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdDefaultIfEmpty)
  + [Distinct](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdDistinct)
  + [ElementAt](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdElemenAt)
  + [Empty](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdEmpty)
  + [Except](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdExcept)
  + [First](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdFirst)
  + [FirstOrDefault](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdFirstOrDefault)
  + [Union](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdUnion)
  + [Intersect](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdIntersect)
  + [Last](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdLast)
  + [LastOrDefault](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdLastOrDefault)
  + [LongCount](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdLongCount)
  + [Max](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdMax)
  + [Min](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdMin)
  + [OfType](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdOfType)
  + [Range](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdRange)
  + [Repeat](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdRepeat)
  + [Reverse](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdReverse)
  + [Single](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdSingle)
  + [Skip](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdSkip)
  + [SkipWhile](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdSkipWhile)
  + [Sum](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdSum)
  + [ThenBy](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdThenBy)
  + [ToArray](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdToArray)
  + [ToDictionary](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdToDictionary)
  + [ToList](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdToList)
  + [Zip](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdZip)
* [Related Articles](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdRelatedArticles)
* [History](https://www.codeproject.com/Articles/383749/How-does-it-work-in#paraIdHistory)

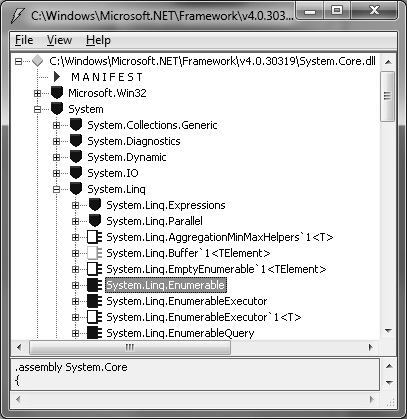
LINQ basics

In .NET any data structure which is derived from the IEnumerable<T> interface of the [System.Collections.Generic](http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx) namespace of the *mscorlib.dll* (this assembly is located in *C:\Windows\Microsoft.NET\Frameworkv4.0.30319* but depends on the installation of the VS) assembly is able to access all the extension methods defined in the Enumerable class of the [System.Linq](http://msdn.microsoft.com/en-us/library/system.linq.aspx) namespace of the *System.Core.dll* assembly in the .NET Framework (see more about [LINQ](http://en.wikipedia.org/wiki/Language_Integrated_Query)). This Enumerable class is a static non inheritable class defined in the System.Linq namespace of the *System.Core.dll* assembly(*C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Core\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Core.dll* but depends on the installation of VS). The definition of the Enumerbale class is as below:

Hide   Copy Code

.class public abstract auto ansi sealed beforefieldinit System.Linq.Enumerable extends [mscorlib]System.Object

Note: The figure shows the Enumerable class decompiled using the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program,



***Fig: Enumerable class in ILDasm.exe program.***

The static Enumerable class is a container of the different extension methods for the IEnumerable<T> interface, for example,

Hide   Copy Code

public static bool Contains<TSource>(

this IEnumerable<TSource> source, TSource value)

{ */\* code removed\*/*}

public static int Count<TSource>(

this IEnumerable<TSource> source)

{ */\* code removed\*/*}

public static IEnumerable<TSource> Distinct<TSource>(

this IEnumerable<TSource> source, IEqualityComparer<TSource> comparer)

{ */\* code removed\*/*}

*// and many more*

Please read more about extension methods [here](http://msdn.microsoft.com/en-us/library/bb383977.aspx) Or Chapter 04 of the [Expert C# 5.0 with .NET Framework 4.5 book](http://www.amazon.com/exec/obidos/ASIN/1430248602/ref=nosim/apre-20).

***Extension method - behind the scene -*** Now do bit of interesting work, in this research I use **ILDasm**(*C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools\ildasm.exe* but this is for my installed VS) tool which comes as part of the VS installation. I use the same C# code as OP provided except change the class name Exte to ExtensionMethodClass. I build the program and grab the exe of the program in this case it is ConsoleApplication24.exe (as I put the code in the console app which is called ConsoleApplication24.) into the ILDasm program and ILDasm generate following IL code for the ExtensionMethodClass class in here we define the extension method.

Hide   Copy Code

.class public abstract auto ansi sealed beforefieldinit ExtensionMethodClass

extends [mscorlib]System.Object

{

.custom instance void [System.Core]System.Runtime.CompilerServices.ExtensionAttribute::.ctor()

.method public hidebysig static string GetFirstThreeCh(string str) cil managed

{

.custom instance void [System.Core]System.Runtime.CompilerServices.ExtensionAttribute::.ctor()

.maxstack 3

.locals init (

[0] string str2,

[1] bool flag)

*//removed all the code for clearity*

}

}

If we look into the IL code above, we can see:

* ExtensionMethodClass class has been define as abstract and sealed.
* Inside this class GetFirstThreeCh method has been compiled as static public method which has string type parameter. **Note** that in here there is no this which we define in the C# code in the GetFirstThreeCh method.
* So it is clear more this IL code that extension method has been define as same as static method.

Let’s see the usage of this GetFirstThreeCh extension method from the Main method which will show us the calling convention of the extension method in underneath. I extracted the following IL code from the program class of the ConsoleApplication24.exe program using the ILdasm:

Hide   Copy Code

.method private hidebysig static void Main(string[] args) cil managed

{

.entrypoint

.maxstack 1

.locals init (

[0] string str)

L\_0000: nop

L\_0001: ldstr "my new String"

L\_0006: stloc.0

L\_0007: ldloc.0

L\_0008: call string ConsoleApplication24.ExtensionMethodClass::GetFirstThreeCh(string)

L\_000d: stloc.0

L\_000e: ret

}

From the above IL code in L\_0008 label we can see that GetFirstThreeCh method has been call as same as CLR call static method.To test this I created a small static class with a static method as below:

Hide   Copy Code

namespace ConsoleApplication24

{

class Program

{

static void Main(string[] args)

{

TestStaticMethod.TestMethod();

}

}

public class TestStaticMethod

{

public static void TestMethod()

{

*/\*Code has been removed\*/*

}

}

}

After building this new program I decompiled using ILDasm program to generate the IL code which is as below:

Hide   Copy Code

.method private hidebysig static void Main(string[] args) cil managed

{

.entrypoint

.maxstack 8

L\_0000: nop

L\_0001: call void ConsoleApplication24.TestStaticMethod::TestMethod()

L\_0006: nop

L\_0007: ret

}

So we can see that how does CLR handle the extension method in behind the scene.

And Iterators in [here](http://msdn.microsoft.com/en-us/library/dscyy5s0(v=vs.80).aspx). A complete list of extension methods defined in the Enumerable class is in [here](http://msdn.microsoft.com/en-us/library/bb345746).

Let’s see how these extension methods work internally.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Extension Methods

Where and Select

Where and Select are two important extension methods of the IEnumerable<TSource> interface defined in the Enumerable class of the *System.Core.dll* assembly.

Hide   Copy Code

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate)

So any data structure derived from IEnumerable<TSource> interface is be able to access these two extension methods over that data structure such as List<T> class. List<T> class implemented the IEnumerable<T> interface as the signature of the List<T> class shows as below,

Hide   Copy Code

public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable

For more information please see [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) and [Select](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.select.aspx). Let’s see an example in here where the Where and Select extension methods have been used.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One", "Two", "Three", "Four",

"Five", "Six", "Seven"

};

var numbersLengthThree =

numbers.Where(x => x.Length == 3).Select(x => x).ToList();

numbersLengthThree.ForEach(x => Console.WriteLine(x));

}

}

}

This code will create a list of string objects and store into a List<string> object, numbers. The above program will find out those items from the numbers whose total number of characters is equal to 3 and store the result into a new list, numbersLengthThree. Finally, display the number on the console. This program will produce the following output,

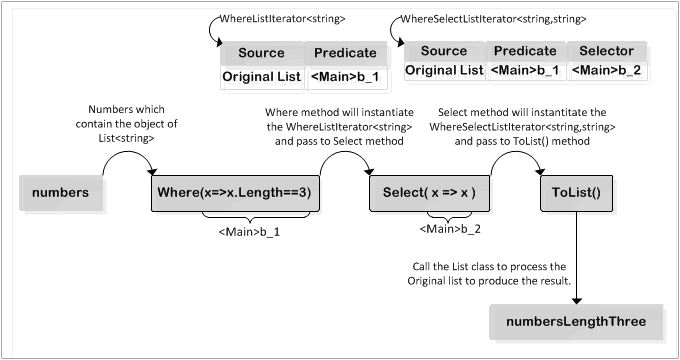
Hide   Copy Code

One

Two

Six

Let’s do bit of research to find out how does it work. The most important code from the above example is numbers.Where(x => x.Length == 3).Select(x => x).ToList(). Following diagram will explain the entire process of this execution.



***Fig: How does Where and Select extension methods work.***

From the above diagram we can see that the CLR passed the numbers list as input to the Where method along with the instance of the MulticastDelegate which holds the information about the <Main>b\_1 method (created from anonymous method (x=>x.Length == 3) ). From the Where method it will return instance of the WhereListIterator<string> iterator which will then be used to pass as input parameter to the Select clause along with another instance of the MulticastDelegate class which will hold the information about the method <Main>b\_2 (created from anonymous method (x=>x)). The Select method will instantiate the relevant Iterator based on the input. In this circumstance, it will be WhereSelectListIterator<string,string> iterator. This will be passed as input parameter to the ToList() method which will finally process the original list by iterating through to get the new list based on the filtering criteria. In depth of the execution will be as below,

**Step 1**: In the compile time, compiler will create a method <Main>b\_1 using anonymous method provided for the Where methods i.e. for x => x.Length == 3.

**Note**: I decompiled the executable file produced by the above program using the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) (*C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools\ILDasm.exe* depending on the VS installation) program and find out the generated method <Main>b\_1 for x => x.Length == 3 code is as below,

Hide   Copy Code

.method private hidebysig static bool <Main>b\_\_1(string x) cil managed

{

.maxstack 2

.locals init (

[0] bool CS$1$0000)

L\_0000: ldarg.0

L\_0001: callvirt instance int32 [mscorlib]System.String::get\_Length()

L\_0006: ldc.i4.3

L\_0007: ceq

L\_0009: stloc.0

L\_000a: br.s L\_000c

L\_000c: ldloc.0

L\_000d: ret

}

Or the equivalent C# code is as below,

Hide   Copy Code

private static bool <Main>b\_\_1(string x)

{

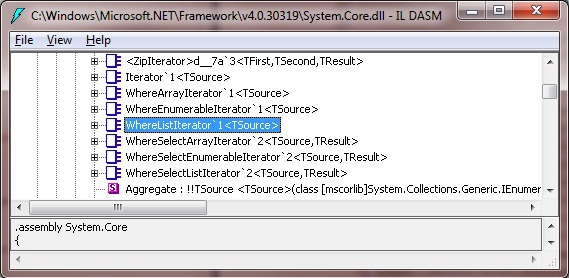
return (x.Length == 3);

}

The CLR will create an instance of the MulticastDelegate class using the method <Main>b\_1 and store this method information into the MulticastDelegate instance and continue the operation.

**Step 2**: The CLR will continue the execution and goes to the Where<TSource>(this IEnumerable<TSource> source, Func<tsource,> predicate) </tsource,>method of the Enumerable class with the original list numbers and instance of MulticastDelegate (created in **Step 1**) as input to the Where method. Based on the source type of the numbers object, Where method will return the appropriate iterator instance as output. For example, based on the above example code it will return WhereListIterator<TSource> iterator which will contain the original list as source and <Main> b\_1 as predicate inside it.

**Note**: Complete list of iterator’s class can be found in the *System.Core.dll* assembly as below,



***Fig: Iterator classes used in the Enumerable clss of the System.Core.dll assembly.***

The CLR will use WhereListIterator<TSource> to pass as parameter to the Select clause as input.

**Step 3**: In the compile time the compiler also created another method using anonymous method code (x=>x) which is <Main>b\_2,

NOTE: the contents of the <Main>b\_2 method extracted via the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program from the executable of the above example program,

Hide   Copy Code

.method private hidebysig static string <Main>b\_\_2(string x) cil managed

{

.custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGeneratedAttribute::.ctor()

.maxstack 1

.locals init (

[0] string CS$1$0000)

L\_0000: ldarg.0

L\_0001: stloc.0

L\_0002: br.s L\_0004

L\_0004: ldloc.0

L\_0005: ret

}

Or the equivalent C# code,

Hide   Copy Code

private static string <Main>b\_\_2(string x)

{

return x;

}

The CLR will use this as input parameter to the MulticastDelegateclass and create another instance of the MulticastDelegate type. It will then use the iterator instantiated from the **Step 2** and delegate instance in this step as input for the Select clause and move the execution to the Select method.

**Step 4**: In the Select method of the Enumerable class CLR will instantiate the relevant iterator based on the input of the Enumerable object and the selector delegate. For the above example, it will return an instance of the WhereSelectListIterator<tsource,tresult />iterator. This iterator will contain the original list, the predicate delegate which hold the anonymous method <Main>b\_1 and the selector delegate which will contain the anonymous method

b\_2 inside it.

**Step 5**: The CLR pass this WhereSelectListIterator<string,string> iterator instance to the ToList() method as the input parameter. In the ToList() method CLR will create an instance of List<TSource> type by passing the WhereSelectListIterator<string,string> iterator as input to it.

**Step 6**: In the constructor of the List<TSource> class CLR will copy the list provided as input into a new list and iterate through the enumerator of the iterator (WhereSelectListIterator<string,string> ) over this new list. This will make sure original list does not alter and add the result into a dynamic array \_items inside the List object. This list object will be returned as a result for the original list based on the filtering criteria. The approximate code for that is as below,

Hide   Copy Code

public List(IEnumerable<T> collection)

{

ICollection<T> is2 = collection as ICollection<T>;

if (is2 != null)

{

int count = is2.Count;

this.\_items = new T[count];

is2.CopyTo(this.\_items, 0);

this.\_size = count;

}

else

{

this.\_size = 0;

this.\_items = new T[4];

using (IEnumerator<T> enumerator = collection.GetEnumerator())

{

while (enumerator.MoveNext())

{

this.Add(enumerator.Current);

}

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

All

This extension method determines whether all elements of a sequence satisfy a condition, if every element of the source sequence passes the condition in the specified predicate, or if the sequence is empty return true; otherwise, false. Read [more](http://msdn.microsoft.com/en-us/library/bb548541.aspx)

The signature of this extension method is as below,

Hide   Copy Code

public static bool All<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

This extension method used to determine whether a sequence of items meet a condition i.e. Each of the items in the sequence will be evaluated against the predicate. In the following program, I created an instance of the List<string> numbers which contains One, Two, Three and so on as items. Following program will find out whether the items of this sequence have at least three characters.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One", "Two", "Three", "Four",

"Five", "Six", "Seven"

};

if (numbers.All<string>(x => x.Length >= 3))

Console.WriteLine("All numbers have at least three characters.");

}

}

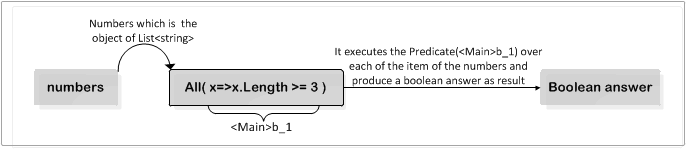
}

The above program will produce following output,

Hide   Copy Code

All numbers have at least three characters.

Because the extension method All will match the condition specified in the predicate whether it is valid for the item from the sequence or not. So it will be working as below,



***Fig: All extension method working details.***

From the above diagram we can see that the CLR will pass the numbers list as input to the extension method All along with the instance of the MulticastDelegate class created using the anonymous method (x=>x.Length >= 3) code. In the All method CLR will process the list to find out whether each of the items from the list meets the condition via the provided predicate.

The CLR will execute the extension method All as below,

**Step 1**: The compiler will construct a method <Main>b\_1 using the anonymous method (x => x.Length >= 3) . The CLR will pass this <Main>b\_1 method into the MulticastDelegate class to instantiate an instance of it. The CLR will pass the original list and the instance of the MulticastDelegate class created in this step as input to the extension method All of the Enumerable class.

**Step 2**: The All extension method will loop through the list and try to find out whether any element in the sequence does not meet the condition and return false otherwise a true value as a result of the operation. The approximate code for this logic is as be as below,

Hide   Copy Code

public static bool All<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

{

foreach (TSource local in source)

{

if (!predicate(local))

{

return false;

}

}

return true;

}

The C# method has been derived from the following IL code which has been generated by [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program for the Enumerable class of the *System.Core.dll* assembly,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

.method public hidebysig static bool All<TSource>(class [mscorlib]System.Collections.Generic.IEnumerable`1<!!TSource> source,

class [mscorlib]System.Func`2<!!TSource, bool> predicate) cil managed

{

.custom instance void System.Runtime.CompilerServices.ExtensionAttribute::.ctor()

.maxstack 2

.locals init (

[0] !!TSource local,

[1] bool flag,

[2] class [mscorlib]System.Collections.Generic.IEnumerator`1<!!TSource> enumerator)

L\_001c: ldarg.0

L\_001d: callvirt instance class [mscorlib]System.Collections.Generic.IEnumerator`1<!0>

[mscorlib]System.Collections.Generic.IEnumerable`1<!!TSource>::GetEnumerator()

L\_0022: stloc.2

L\_0023: br.s L\_0039

L\_0025: ldloc.2

L\_0026: callvirt instance !0 [mscorlib]System.Collections.Generic.IEnumerator`1<!!TSource>::get\_Current()

L\_002b: stloc.0

L\_002c: ldarg.1

L\_002d: ldloc.0

L\_002e: callvirt instance !1 [mscorlib]System.Func`2<!!TSource, bool>::Invoke(!0)

L\_0033: brtrue.s L\_0039

L\_0035: ldc.i4.0

L\_0036: stloc.1

L\_0037: leave.s L\_004f

L\_0039: ldloc.2

L\_003a: callvirt instance bool [mscorlib]System.Collections.IEnumerator::MoveNext()

L\_003f: brtrue.s L\_0025

L\_0041: leave.s L\_004d

L\_0043: ldloc.2

L\_0044: brfalse.s L\_004c

L\_0046: ldloc.2

L\_0047: callvirt instance void [mscorlib]System.IDisposable::Dispose()

L\_004c: endfinally

L\_004d: ldc.i4.1

L\_004e: ret

L\_004f: ldloc.1

L\_0050: ret

.try L\_0023 to L\_0043 finally handler L\_0043 to L\_004d

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Any

This extension method determines whether any element of a sequence exists or satisfies a condition provided as predicate. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.any.aspx).

The method signature for this extension method is as be as below,

Hide   Copy Code

public static bool Any<TSource>(this IEnumerable<TSource> source)

public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above two extension methods will do the following

* First version of the Any extension method will find out whether the sequence of items contains any element in it or not.
* Second version of the Any extension method will find out is there any element in the sequence which match the criteria provided in the predicate.

I wrote a small program to explain the working details of those two versions of extension methods from the Enumerable class.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One", "Two", "Three", "Four",

"Five", "Six", "Seven"

};

if (numbers.Any<string>())

Console.WriteLine("The sequence contains item.");

if (numbers.Any<string>(x => x.Length >= 3))

Console.WriteLine("The sequence contains at least a item which has three or more characters");

}

}

}

The above program will produce the following output,

Hide   Copy Code

The sequence contains item.

The sequence contains at least a item which has three or more characters

When the CLR find out the first version of the Any extension method it will execute following steps to perform the operation,

**Step 1**: The CLR will send the original sequence or the list in this case numbers to the Any<TSource>(this IEnumerable<TSource> source) extension method as input.

**Step 2**: This method will loop through the list numbers via the Enumerator object return from the list numbers and check whether the enumerator return a true value while calling the MoveNext() method of it and returns true otherwise false i.e. the sequence does not have any element in it.

The approximate code for the Any extension method is as below,

Hide   Copy Code

public static bool Any<TSource>(this IEnumerable<TSource> source)

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (enumerator.MoveNext())

{

return true;

}

}

return false;

}

For the second version of the Any extension method CLR will do the following steps to execute,

**Step 1**: The compiler will construct a method <Main>b\_1 using the anonymous method (x => x.Length >= 3) . The CLR will pass this <Main>b\_1 method to the MulticastDelegate class to instantiate an instance of it and pass this MulticastDelegate instance as predicate along with the original list to the Any extension method.

**Step 2**: Inside the Any extension method CLR will loop through the list to execute the predicate using each item as input to the predicate. If the predicate return true of first iteration it will return true otherwise it will continue until find a match or not. The approximate code for the,

Hide   Copy Code

public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

{

foreach (TSource local in source)

{

if (predicate(local))

{

return true;

}

}

return false;

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Average

The Average extension method calculates the average of a sequence of numeric values. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.average).

The method signature of these extension methods are below,

Hide   Copy Code

public static double Average(this IEnumerable<int> source)

public static decimal Average<TSource>(this IEnumerable<TSource> source, Func<TSource, decimal> selector)

An example of the Average extension method is as below,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7,8,9,10

};

Console.WriteLine("Average of the numbers :{0}", numbers.Average());

Console.WriteLine("Average of the original numbers x2 :{0}",

numbers.Average((x => x \* 2)));

}

}

}

The above program will produce the following output,

Hide   Copy Code

Average of the numbers :5.5

Average of the original numbers x2 :11

So when the CLR will find out the first version of the Average method as in the above program, it will do the following steps to perform the operation,

**Step 1**: The CLR will pass the original list in this case numbers as input to the Average method.

**Step 2**: The Average method will loop through the list and perform the average operation for the given list. The approximate code is as below,

Hide   Copy Code

public static double Average(this IEnumerable<int> source)

{

long num = 0L;

long num2 = 0L;

foreach (int num3 in source)

{

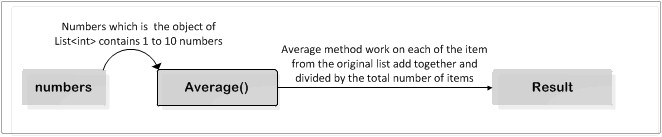
num += num3;

num2 += 1L;

}

return (((double) num) / ((double) num2));

}



***Fig: Average extension method working details.***

From the above diagram we can see that the CLR will pass the list numbers as input to the Average method. This method will process the original list by iterating through it and calculate the average of items stored into the numbers list and return the average of the items as result.

The second version of the Average extension method will do the following steps,

**Step 1**: The CLR will pass the original list and the instance of the MulticastDelegate type created using the <Main>b\_1 method (which is generated in the compile time by the compiler using anonymous method (x=>x\*2) ) as input to the Average extension method.

**Step 2**: The CLR will call the Select method (public static IEnumerable<TResult> Select<tsource,>(this IEnumerable<TSource> source, Func<tsource,> selector) </tsource,></tsource,>), it will instantiate WhereSelectListIterator<int, /> iterator from this method and send back to the Select method.

Step 3: The CLR will then call Average method which it will accept the iterator instance for example, WhereSelectListIterator<int, /> created in **Step 1** as input. This iterator will contain the original list and the instance of the MulticastDelegate as the Selector.

**Step 4**: In the Average method ForEach statement will iterate through the list

Hide   Copy Code

foreach (int num3 in source)

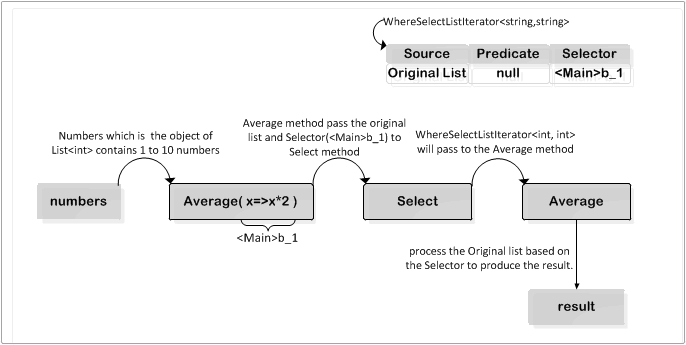
{

num += num3;

num2 += 1L;

}

and perform the average calculation. Following image shows,



***Fig: Average extension method working details.***

From the above diagram we can see that the CLR will pass the list numbers as input to the Average extension method (Average<int>(this IEnumerable<int> source, Func<int,> selector) </int,></int></int>) along with the instance of MulticaseDelegate class instantiated from anonymous method (x=>x\*2) code. The CLR will pass this to the Select method from where the appropiate iterator will be instantitated and return to the Average method. In this case it will be WhereSelectListIterator<int,int /> iterator will be passed to the Average method to process the original list. The WhereSelectListIterator<int,int /> iterator will hold the original list and the selector inside. From the average method CLR will iterate through the list and calculate the average based on the on the filtering criteria provided in the Selector.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Concat

This extension method concatenates two sequences. This method is implemented by using deferred execution (please read [here](http://msdn.microsoft.com/en-us/library/bb943859.aspx) for more about the deferred execution). The immediate return value is an instance one of the relevant iterator type that stores all the information that is required to perform the action. The query represented by this method is not executed until the object is enumerated either by calling its GetEnumerator method directly or by using foreach statement in C#. Read [more](http://msdn.microsoft.com/en-us/library/bb302894).

The Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>) method differs from the Union method because the Concat<TSource>(IEnumerable<TSource>, IEnumerable<TSource>) method returns all the original elements in the input sequences. The Union method returns only unique elements from the sequences. The method signature for this extension method is as below,

Hide   Copy Code

public static IEnumerable<TSource> Concat<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)

The following program will show the usage of the Concat method,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> listOne = new List<int>()

{

1,2,3,4,5

};

IList<int> listTwo = new List<int>()

{

6,7,8,9,10

};

var result = listOne.Concat(listTwo).ToList();

result.ForEach(x=> Console.WriteLine(x));

}

}

}

The above program will produce the following output,

Hide   Copy Code

1

2

3

4

5

6

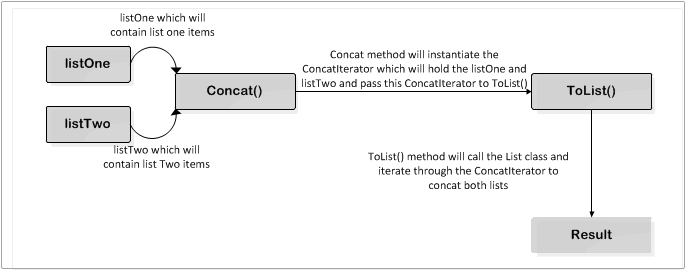
7

8

9

10

The Concat extension works as below:



***Fig: Concat extension method working details.***

From the above diagram we can see that the CLR passed the listOne and listTwo as input to the Concat method and from the Concat method it will return ConcatIterator<int /> instance as output to the caller of this extension method. Though this will be executed using deferred execution pattern, the ToList() method will start processing using the logic implemented in the ConcatIterator class of listOne and listTwo to produce the final list.

When the CLR finds the Concat extension method it will execute following steps to perform the operation,

**Step 1**: The CLR will pass the original lists in this case listOne and listTwo to the Concat<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second) method as input parameter.

**Step 2**: From the Concat method CLR will return an instance of the ConcatIterator<int> iterator which will hold the listOne and listTwo and return to the caller of the Concat method. The approximate code for the Concat method is as below,

Hide   Copy Code

public static IEnumerable<TSource> Concat<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)

{

return ConcatIterator<TSource>(first, second);

}

Step 3: Due to the deferred execution this iterator will be executed by the ToList() method which will loop through the lists such as listOne and listTwo via the Enumerator object returned from the ConcatIterator<int /> instance and insert each of the item from listOne and listTwo to a new list and return as result.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Contains

The Contains extension method determines whether a sequence contains a specified element. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.contains)

This method will work as below,

* It determines whether a sequence contains a specified element by using the default equality comparer.
* Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T>.

The method signature for this extension method is as below,

Hide   Copy Code

public static bool Contains<TSource>(this IEnumerable<TSource> source, TSource value)

public static bool Contains<TSource>(this IEnumerable<TSource> source, TSource value, IEqualityComparer<TSource> comparer)

This method will search the list whether it has a particular value in it or not. To explain this method I wrote a small program:

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> listOne = new List<int>()

{

1,2,3,4,5

};

var resultAsTrue = listOne.Contains(2);

var resultAsFalse = listOne.Contains(200);

Console.WriteLine("{0}\n{1}", resultAsTrue, resultAsFalse);

}

}

}

The above program will produce the following output,

Hide   Copy Code

True

False

So when the compiler finds the first version of the Contains method as in the above program, it will do the following steps to perform the operation,

* The CLR will search a particular item into the list in the Contains method. This search will have two direction such as if the input is a null value then it will loop through the list to match item with the null and return true if one of the item from the list is null otherwise false. Other than null value CLR will compare the value (provided to match as input) with each of the item from the list, depending on match it will return a boolean answer.

The approximate code of the Contains method is as below,

Hide   Copy Code

public bool Contains(T item)

{

if (item == null)

{

for (int j = 0; j < this.\_size; j++)

{

if (this.\_items[j] == null)

{

return true;

}

}

return false;

}

EqualityComparer<T> comparer = EqualityComparer<T>.Default;

for (int i = 0; i < this.\_size; i++)

{

if (comparer.Equals(this.\_items[i], item))

{

return true;

}

}

return false;

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Count

This count extension method returns the number of elements in a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.count).

The method signature for this extension method is as below,

Hide   Copy Code

public static int Count<TSource>(this IEnumerable<TSource> source)

public static int Count<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above two extension methods will do the followings,

* First version returns the number of elements in a sequence.
* Second version returns a number that represents how many elements in the specified sequence satisfy a condition.

The Count method will find out how many items in the list. For example in the following program I created a list of string objects. I use Count() method to find out how many items in the list and also how many items which is more than three characters.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> listOne = new List<string>()

{

"One","Two","Three"

};

var result = listOne.Count();

var fourOrMoreCharacters = listOne.Count(item => item.Length > 3);

Console.WriteLine("{0}\n{1}", result,fourOrMoreCharacters);

}

}

}

The above program will produce the following output,

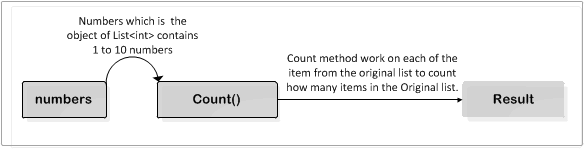
Hide   Copy Code

3

1

In this example I use two different version of Count method,

**Step 1**: When the CLR finds the first version of the Count method it will try to find out the Enumerator object of given list and iterate through items (using the iterator of the list) unless the MoveNext() method of the enumerator returns false.



***Fig: Count extension method working details.***

This method will return the number of iteration as the output of this program as the number of iteration will be the number of items in the list. The approximate code for the Count is as below,

Hide   Copy Code

public static int Count<TSource>(this IEnumerable<TSource> source)

{

int num = 0;

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

while (enumerator.MoveNext())

{num++;}

}

return num;

}

**Step 2**: The second version of the Count() method will accept the original list and a predicate to filter the count based on the condition. The predicate will be created in the compile time based on the anonymous method. The CLR will loop through the items of the list and execute the predicate over each of the item while looping through. If the predicate meet the condition over the item on iteration it will increase the item count. Finally it will return the item count as the total number of item which meets the condition. The approximate code for the Count method is as below,

Hide   Copy Code

public static int Count<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

{

foreach (TSource local in source)

{

if (predicate(local))

{

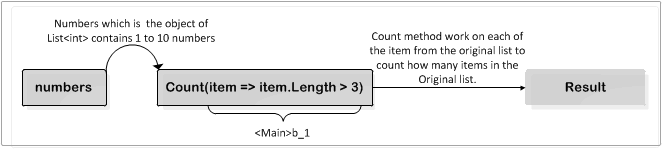
num++;

}

}

return num;

}



***Fig: Count extension method working details.***

From the above diagram we can see that the CLR passed the numbers list as input to the Count method along with the instance of the MulticastDelegate class created using <Main>b\_1 which will created using the anonymous method (item=>item.Length > 3) code. In the count method CLR will iterate through each of the item and execute the delegate object against the item, depending on the return value of the delegate the number of count will increase and return the result as the total count of the list depends on the condition provided by (item=>item.Length > 3) .

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

DefaultIfEmpty

This extension method returns the elements of an IEnumerable<T>, or a default valued singleton collection if the sequence is empty. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.defaultifempty)

The signature for this extension method is as below:

Hide   Copy Code

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(this IEnumerable<TSource> source)

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(this IEnumerable<TSource> source, TSource defaultValue)

The above two extension methods will do the followings,

* The first version returns the elements of the specified sequence or the type parameter's default value in a singleton collection if the sequence is empty.
* The second version returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty.

This method can be used on a list which does not have items in it and if we call the extension method over this list it will return default value of the item. Let’s see the following program which uses DefaultIfEmpty method:

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<Person> persons = new List<Person>();

IList<int> numbers = new List<int>();

IList<string> names = new List<string>();

var defaultPersons = persons.DefaultIfEmpty();

var defaultNumbers = numbers.DefaultIfEmpty().ToList();

var defaultNames = names.DefaultIfEmpty();

}

}

class Person

{

public string Name

{

get;

set;

}

public string Address

{

get;

set;

}

public int Age

{

get;

set;

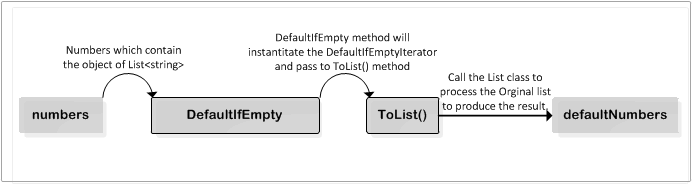
}

}

}

In the above program I declared three lists of person objects, numbers and names of type Person, int and string accordingly. These three lists do not have any item and Count Property of this list will return 0. When I call the DefaultIfEmpty extension method over any of this list then the CLR will do the following steps to process it,

* The CLR will copy the list to this DefaultIfEmpty method, from this method CLR will return the instance of the DefaultIfEmptyIterator<TSource> iterator which will hold the defaultvalue and source value. The defaultvalue property will contain the default value of the type of list and source will be the original list.
* The CLR will pass the DefaultIfEmptyItereator to the ToList() method which will call the the List class passing the object of the DefaultIfEmptyItereator as input. In this class CLR will iterate through original list and process the result.



***Fig: DefaultIfEmpty extension method working details.***

The approximate code for the DefaultIfEmptyIterator is as below,

Hide   Copy Code

private static IEnumerable<TSource> DefaultIfEmptyIterator<TSource>(IEnumerable<TSource> source, TSource defaultValue)

{

using (IEnumerator<TSource> iteratorVariable0 = source.GetEnumerator())

{

if (iteratorVariable0.MoveNext())

do

{

yield return iteratorVariable0.Current;

}

while (iteratorVariable0.MoveNext());

else

yield return defaultValue;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Distinct

This extension method returns distinct elements from a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.distinct).

The method signature of this extension method is as below,

Hide   Copy Code

public static IEnumerable<TSource> Distinct<TSource>(this IEnumerable<TSource> source)

public static IEnumerable<TSource> Distinct<TSource>(this IEnumerable<TSource> source, IEqualityComparer<TSource> comparer)

The above two extension methods will do the followings,

* The first version returns distinct elements from a sequence by using the default equality comparer to compare values.
* The second version returns distinct elements from a sequence by using a specified IEqualityComparer<T> to compare values.

The Distinct extension method will return the identical items from the list i.e. if we have a list which contains duplicate items using this method it will filtered the duplicated items and return a new list which will only contains the each of the item only once in the list. Let’s see the following program in where I use Distinct method on a list which contains a set of {1,1,1,2,2,2,3,3,3}.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,1,1,2,2,2,3,3,3

};

var distinctedNumbers = numbers.Distinct().ToList();

distinctedNumbers.ForEach(x=>Console.WriteLine(x));

}

}

}

This program will produce following output,

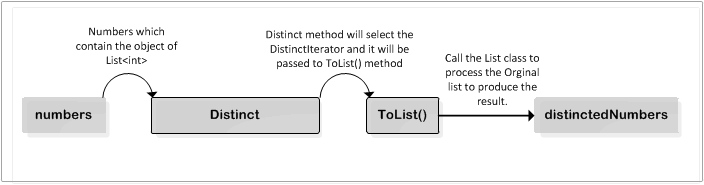
Hide   Copy Code

1

2

3

When the above program will run it will produce {1, 2, 3} as output. Following diagram shows that how does distinct method works,



***Fig: Distinct extension method working details.***

To execute the Distinct method CLR will do the followings,

**Step 1**: The CLR will copy the original list to the Distinct method as input which will call the Distinct<TSource>(this IEnumerable<TSource> source, IEqualityComparer<TSource> comparer) method internally it will return an instance of the DistinctIterator<TSource>( IEnumerable<TSource> source, IEqualityComparer<TSource> comparer) class but this iterator will not execute due to the deferred execution (to execute the DistinctIterator iterator we need to call the ToList() method over the list or need to do the ForEach).

**Step 2**: From the ToList() method CLR will call the List class by passing the DistinctIterator created in **Step 1** as input to it. The List class iterate through the instance of the DistinctIterator. The iteration logic implemented in the DistinctIterator will create a new instance of the Set<TSource> and iterates through the original list and adds the iterated item in the Set<TSource> instance it created earlier. Internally the Set<TSource> class will use Add and Find method to add the item from the given sequence into the internal slots array only when there is no duplicate item in the array slot. This will continue until CLR reaches the end of the list and will get a list with distinct items.

So the Distinct method will work as below over a list,

Hide   Copy Code

private static IEnumerable<TSource> DistinctIterator<TSource>(

IEnumerable<TSource> source, IEqualityComparer<TSource> comparer)

{

Set<TSource> iteratorVariable0 = new Set<TSource>(comparer);

foreach (TSource iteratorVariable1 in source)

{

if (iteratorVariable0.Add(iteratorVariable1))

{

yield return iteratorVariable1;

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

ElementAt

This extension method returns the element at a specified index in a sequence. Read [more](http://msdn.microsoft.com/en-us/library/bb299233).

The method signature for this extension method is as below,

Hide   Copy Code

public static TSource ElementAt<TSource>(this IEnumerable<TSource> source, int index)

public static TSource ElementAtOrDefault<TSource>(this IEnumerable<TSource> source, int index)

Following shows an example of ElementAt extension method of the IEnumerable<TSource> interface.

Hide   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One","Two","Three"

};

var elementAt = numbers.ElementAt(1);

Console.WriteLine(elementAt);

}

}

}

This program is creating a numbers list which is storing One, Two and Three. From this numbers list I am trying to access the element which is stored in the position (array position) 1 and stored into the elementAt variable to display on the console. This program will produce the following output,

Hide   Copy Code

Two

To execute the ElementAt method the CLR will do the following things,

**Step 1**: The CLR will call the get\_Item method from the System.Collections.Generic.List`1<T> class while executing the ElementAt<TSource>(this IEnumerable<TSource> source, int index) extension method. The following code has been extracted from the ElementAt method of the System.Linq.Enumerable class of the *mscorlib.dll* assembly using the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program:

Hide   Copy Code

.method public hidebysig static !!TSource ElementAt<TSource>(

class [mscorlib]System.Collections.Generic.IEnumerable`1<!!TSource> source,

int32 index) cil managed

{

IL\_0018: ldloc.0

IL\_0019: ldarg.1

IL\_001a: callvirt instance !0 class

[mscorlib]System.Collections.Generic.IList`1<!!TSource>::get\_Item(int32)

IL\_001f: ret

} *// end of method Enumerable::ElementAt*

From the above code we can see that the CLR will call the get\_Item(int32) method of the System.Collections.Generic.List`1<T> class with the index provided from the caller of the ElementAt method as input parameter in the label IL\_001a.

**Step 2**: get\_Item(int32) method of the System.Collections.Generic.List`1<T> class will load the \_items array of this class (label IL\_000f in the following IL code) and it will then load the argument (label IL\_0014 in the following IL code) to get the index which will later use to access item from the \_items array based on the index. If we see the IL code of the get\_Item(int32) method from the System.Collections.Generic.List`1<T> class from the *mscorlib.dll* using ILDasm:

Hide   Copy Code

.method public hidebysig newslot specialname virtual final

instance !T get\_Item(int32 index) cil managed

{

IL\_000e: ldarg.0

IL\_000f: ldfld !0[] class System.Collections.Generic.List`1<!T>::\_items

IL\_0014: ldarg.1

IL\_0015: ldelem !T

IL\_001a: ret

} *// end of method List`1::get\_Item*

From the above code ldfld IL instruction used in the IL\_000f will load the \_items field of the List<T> class and on IL\_0014 label it will load the argument 1 which is the index will use to access the item from the \_items array using the IL instruction ldelem used in the IL\_0015.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Empty

Empty extension method returns an empty IEnumerable<T> that has the specified type argument. Read [more](http://msdn.microsoft.com/en-us/library/bb341042).

The method signature for this extension is as below,

Hide   Copy Code

public static IEnumerable<TResult> Empty<TResult>()

Following example shows the usage of the Empty method,

Hide   Copy Code

using System;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

var emptyList = Enumerable.Empty<int>();

Console.WriteLine(emptyList.Count());

}

}

}

In the above code I created an empty list of int type using the Empty<int>()</int> method. This program will produce the following output,

Hide   Copy Code

0

When the CLR will execute the Empty extension method it will do the following things to execute the operation.

**Step 1**: The CLR will call the get\_Instance() method of the EmptyEnumerable`1<!!TResult> internal class from the System.Linq namespace of the *System.Core.dll* assembly while executing the Empty<TResult>() method. This class has an array field of given type (!!TResult) and a property Instance which will return the array field. The following IL code of the Empty method has been decompiled using the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program from the *System.Core.dll* assembly.

Hide   Copy Code

.method public hidebysig static class

[mscorlib]System.Collections.Generic.IEnumerable`1<!!TResult>

Empty<TResult>() cil managed

{

IL\_0000: call class [mscorlib]System.Collections.Generic.IEnumerable`1<!0>

class System.Linq.EmptyEnumerable`1<!!TResult>::get\_Instance()

IL\_0005: ret

} *// end of method Enumerable::Empty*

**Step 2**: The Instance property from the System.Linq.EmptyEnumerable`1<!!TResult> will call the get\_Instance() method. The get\_Instance() method will create an array with 0 items. The CLR will first push 0 onto the stack as int32 type using ldc.i4.0 IL instruction used in the label IL\_0007 in the following code. Using the newarr IL instruction CLR will create a new array with the 0 item and will push on the stack. In the label IL\_000d, CLR will use stsfld to replace the value of the field value i.e. instance field’s value using the value from the stack.

Hide   Copy Code

.method public hidebysig specialname static

class [mscorlib]System.Collections.Generic.IEnumerable`1<!TElement>

get\_Instance() cil managed

{

*// Code size 24 (0x18)*

.maxstack 8

IL\_0000: ldsfld !0[] class System.Linq.EmptyEnumerable`1<!TElement>::'instance'

IL\_0005: brtrue.s IL\_0012

IL\_0007: ldc.i4.0

IL\_0008: newarr !TElement

IL\_000d: stsfld !0[] class System.Linq.EmptyEnumerable`1<!TElement>::'instance'

IL\_0012: ldsfld !0[] class System.Linq.EmptyEnumerable`1<!TElement>::'instance'

IL\_0017: ret

} *// end of method EmptyEnumerable`1::get\_Instance*

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Except

The Except method can be used to remove a list of items from another. It produces the set difference of two sequences. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.except)

The method signature for this extension method is as below,

Hide   Copy Code

public static IEnumerable<TSource> Except<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)

public static IEnumerable<TSource> Except<TSource>(this IEnumerable<TSource> first,

IEnumerable<TSource> second, IEqualityComparer<TSource> comparer)

The above two extension methods will do the followings,

* The first version produces the set difference of two sequences by using the default equality comparer to compare values.
* the second version produces the set difference of two sequences by using the specified IEqualityComparer<T> to compare values.

The Except method can be used to remove a list of items from another. For example, if we have a list A with items {1,2,3,4,5,6,7} and B with {1,2,3} so the A except B will produce {4,5,6,7}. Following program used Except to show its usage,

Hide   Copy Code

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstNumbers = new List<int>()

{

1,2,3,4,5,6,7

};

IList<int> secondNumbers = new List<int>()

{

1,2,3

};

var result = firstNumbers.Except(secondNumbers).ToList();

result.ForEach(x => Console.WriteLine(x));

}

}

}

This program will produce the following output,

Hide   Copy Code

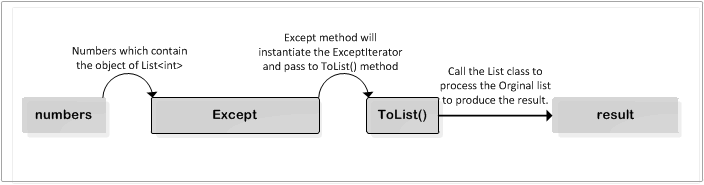
4

5

6

7

When the above program will run it will produce the {4,5,6,7}. Following diagram shows that how the Except extension method works:



***Fig: Except extension method working details.***

The CLR will execute the Except method as following,

**Step 1**: The CLR will copy the original list to the Except method as input which will call the

Hide   Copy Code

Except<TSource>(this IEnumerable<TSource> first,

this IEnumerable<TSource> second, IEqualityComparer<TSource> comparer)

method internally it will instantiate the ExceptIterator<TSource>( this IEnumerable<TSource> first, this IEnumerable<TSource> second, IEqualityComparer<TSource> comparer) iterator class. This iterator will not execute due to the deferred execution until the call the ToList() method over the list or use ForEach statement.

**Step 2**: When the CLR will execute the ToList() method it will call the List class by passing ExceptIterator instance created in **Step 1** as input. The List class will call the ExceptIterator method while it iterates through the list. The ExceptIterator method creates a new instance of the Set<TSource> type and iterates through the second list and adds the iterated item in the Set instance it created earlier. Internally the method will use Add and Find method to add the item into the internal slots array only when there is no duplicate item in the slot. This will continue until compiler reaches the end of the second list. In the second loop CLR will iterate through the first list and try to add iterate item in the Set object it created in this step. If the item of second list does not exist in the set object then it will return that item and continue until it finishes the first list.

The approximate code of the ExceptIterator will be as below,

Hide   Copy Code

private static IEnumerable<TSource> ExceptIterator<TSource>(

IEnumerable<TSource> first,

IEnumerable<TSource> second,

IEqualityComparer<TSource> comparer)

{

Set<TSource> iteratorVariable0 = new Set<TSource>(comparer);

foreach (TSource local in second)

{

iteratorVariable0.Add(local);

}

foreach (TSource iteratorVariable1 in first)

{

if (!iteratorVariable0.Add(iteratorVariable1))

{

continue;

}

yield return iteratorVariable1;

}

}

So the Distinct method will work as below over a list,

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

First

It returns the first element of a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.first).

The method signature is be as below,

Hide   Copy Code

public static TSource First<TSource>(this IEnumerable<TSource> source)

public static TSource First<TSource>(this IEnumerable<TSource> source,Func<TSource, bool> predicate)

The above two First extension methods will do the following,

* First version of this extension method will find out the first item from the sequence of items.
* Second version of this extension method will find out the first item of list which meets the predicate condition.

I wrote a small program to explain the working steps for those two versions of the First extension methods of Enumerable class.

Hide   Copy Code

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7

};

var firstItem = numbers.First();

var firstItemBasedOnConditions = numbers.First(item => item > 3);

Console.WriteLine("{0}\n{1}",

firstItem,

firstItemBasedOnConditions

);

}

}

}

This program will produce the following output:

Hide   Copy Code

1

4

When the CLR will execute the first version of the First extension method with following steps to perform the operation,

* The CLR sends the original list to the First <TSource>(this IEnumerable<TSource> source) method as input parameter.
* This method will return first item from the original list or iterate through the original list and return the first item from the iteration as a result.

The approximate code for this will be as below,

Hide   Copy Code

public static TSource First<TSource>(this IEnumerable<TSource> source)

{

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

if (list.Count > 0)

{

return list[0];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (enumerator.MoveNext())

{

return enumerator.Current;

}

}

}

}

For the second version of the First extension method CLR will do following steps,

* The compiler will construct a method <Main>b\_1 using the anonymous (item => item > 3) . The CLR will pass this <Main>b\_1 method to the MulticastDelegate class to construct an instance of it.
* First method will loop through the list and match with each element in the sequence based on the predicate. This return on first match otherwise it will continue until find a match or not.

The approximate code for the,

Hide   Copy Code

public static TSource First<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

{

foreach (TSource local in source)

{

if (predicate(local))

{

return local;

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

FirstOrDefault

It returns the first element of a sequence, or a default value if no element is found. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.firstordefault).

The method signature for the FirstOrDefault extension is as below:

Hide   Copy Code

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source)

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above two extension method will do the followings,

* It returns the first element of a sequence, or a default value if the sequence contains no elements.
* It returns the first element of the sequence that satisfies a condition or a default value if no such element is found.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstNumbers = new List<int>();

IList<int> secondNumbers = new List<int>()

{

1,2,3,4,5,6,7

};

var firstItemOfFirstList = firstNumbers.FirstOrDefault();

var firstItemIfFirstListBasedOnConditions =

firstNumbers.FirstOrDefault(item => item > 3);

var firstItemOfSecondList = secondNumbers.FirstOrDefault();

var firstItemOfSecondListBasedOnConditions =

secondNumbers.FirstOrDefault(item => item > 3);

Console.WriteLine("{0}\n{1}\n{2}\n{3}",

firstItemOfFirstList,

firstItemIfFirstListBasedOnConditions,

firstItemOfSecondList,

firstItemOfSecondListBasedOnConditions

);

}

}

}

This program will produce the following output,

Hide   Copy Code

0

0

1

4

Approximate code for the first version of the FirstOrDefault extension method:

Hide   Copy Code

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source)

{

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

if (list.Count > 0)

{

return list[0];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (enumerator.MoveNext())

{

return enumerator.Current;

}

}

}

return default(TSource);

}

Approximate for the second version of the FirstOrDefault extension method:

Hide   Copy Code

public static TSource FirstOrDefault<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

{

foreach (TSource local in source)

{

if (predicate(local))

{

return local;

}

}

return default(TSource);

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Union

The Union method will union (denoted as ?) of a collection of sets is the set of all distinct elements in the collection. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.union). For example, if we have two sets, A={1,2,3,4,5,6,7} and B={5,6,7,8,9} the Union of these sets will be A u B ={1,2,3,4,5,6,7,8,9}.

In .NET, the Union method will do the exactly as above diagram, it will join two list and create a new list.

Hide   Copy Code

public static IEnumerable<TSource> Union<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)

public static IEnumerable<TSource> Union<TSource>(this IEnumerable<TSource> first,

IEnumerable<TSource> second, IEqualityComparer<TSource> comparer)

The following program shows the usage of the Union operation:

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstList = new List<int>()

{

1,2,3,4

};

IList<int> secondList = new List<int>()

{

7,9,3,4,5,6,7

};

var result = firstList.Union(secondList);

result.ToList().ForEach(x => Console.WriteLine(x));

}

}

}

This program will produce the following output,

Hide   Copy Code

1

2

3

4

7

9

5

6

When the CLR will execute the Union method as below,

**Step 1**: The Union method will instantiate a UnionIterator<TSource> which will hold the firstList and the secondList and null for the IEqualityComparer as the above program does not provided any.

**Step 2**: Due to the deferred execution this UnionIterator<TSource> will be executed when the CLR will start executing the ToList() method. Inside the UnionIterator<TSource> a new instance of the Set<TSource> class will be instantiated which will be used to find out the distinct item from the both lists.

The approximate code for the UnitonIterator will be as below,

Hide   Copy Code

private static IEnumerable<TSource> UnionIterator<TSource>(

IEnumerable<TSource> first,

IEnumerable<TSource> second,

IEqualityComparer<TSource> comparer)

{

Set<TSource> iteratorVariable0 = new Set<TSource>(comparer);

foreach (TSource iteratorVariable1 in first)

{

if (iteratorVariable0.Add(iteratorVariable1))

{

yield return iteratorVariable1;

}

}

foreach (TSource iteratorVariable2 in second)

{

if (!iteratorVariable0.Add(iteratorVariable2))

{

continue;

}

yield return iteratorVariable2;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Intersect

It produces the set intersection of two sequences. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.intersect). The method signature for this extension method is as below:

Hide   Copy Code

public static IEnumerable<TSource> Intersect<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)

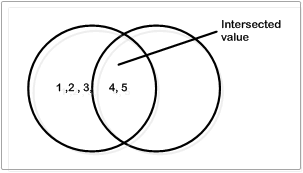
public static IEnumerable<TSource> Intersect<TSource>(this IEnumerable<TSource> first,

Enumerable<TSource> second,IEqualityComparer<TSource> comparer)

The above extension methods will do the followings,

* It produces the set intersection of two sequences by using the default equality comparer to compare values.
* It produces the set intersection of two sequences by using the specified IEqualityComparer<T> to compare values.

It produces the set intersection of two sequences. The intersect operation will produce those element which are common in the both list. For example, if we have a list A with items {1, 2, 3, 4, 5} and B with {4, 5} then the intersection of these two list A n B will produce {4, 5}



***Fig: Intersect operation.***

Let’s see the following program which is creating two list named listA with values 1, 2, 3, 4, 5, and listB with values 4, 5. I am doing an intersect operation between this two list using the Enumerable extension method Intersect.

Hide   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> listA = new List<int>() { 1, 2, 3, 4, 5 };

IList<int> listB = new List<int>() { 4, 5 };

var intersectResult = listA.Intersect(listB);

intersectResult.ToList().ForEach(x => Console.Write("{0}\t",x));

Console.WriteLine();

}

}

}

This program will produce the following output:

Hide   Copy Code

4 5

To execute the Intersect extension method the CLR will do,

**Step 1**: The CLR will instantiate an instance of the IntersectIterator<TSource> and return to the caller. Due to the deferred execution this iterator will not execute until ToList() method is being called.

**Step 2**: From the IntersectIterator<TSource>, the CLR will create an instance of the Set<TSource> object which used to hold all the items from the second list. The CLR will then iterate through the first list and try to remove each of the items of the first list from the Set<TSource> object. If it can remove then it will return the item of the firstlist otherwise it will continue to iterate the first list until it can remove from the set. Following code shows the approximate code for the, IntersectIterator method,

Hide   Copy Code

private static IEnumerable<TSource> IntersectIterator<TSource>(

IEnumerable<TSource> first,

IEnumerable<TSource> second,

IEqualityComparer<TSource> comparer)

{

Set<TSource> iteratorVariable0 = new Set<TSource>(comparer);

foreach (TSource local in second)

{

iteratorVariable0.Add(local);

}

foreach (TSource iteratorVariable1 in first)

{

if (!iteratorVariable0.Remove(iteratorVariable1))

{

continue;

}

yield return iteratorVariable1;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Last

It returns the last element of a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.last)

The method signature for this extension method is as below,

Hide   Copy Code

public static TSource Last<TSource>(this IEnumerable<TSource> source)

public static TSource Last<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above two First extension methods will do the following

* First version of this extension method will find out the first item from the sequence of items.
* Second version of this extension method will find out the first item of list which meets the predicate condition.

I wrote a small program to explain the working steps for those two versions of Last extension methods of Enumerable class.

Hide   Copy Code

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7

};

var lastItem = numbers.Last();

var lastItemBasedOnConditions = numbers.Last(item => item > 3);

}

}

}

When the CLR will execute the first version of the Last extension method as below,

* The original list will be passed to the Last <TSource>(this IEnumerable<TSource> source) method as input parameter.
* This method will loop through the list via the Enumerator object returns from the list and check whether the enumerator return a true value while calling the MoveNext() method of it and returns true otherwise false, i.e., the sequence does not have any element in it.

The approximate code for this will be as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

public static TSource Last<TSource>(this IEnumerable<TSource> source)

{

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

int count = list.Count;

if (count > 0)

{

return list[count - 1];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (enumerator.MoveNext())

{

TSource current;

do

{

current = enumerator.Current;

}

while (enumerator.MoveNext());

return current;

}

}

}

throw Error.NoElements();

}

For the second version of the Any extension method CLR will do following steps,

* The compiler will construct a method <Main>b\_1 using the anonymous method (item => item > 3) in the compile time. The CLR will pass this <Main>b\_1 method to the MulticastDelegate class to construct an instance of it and pass this <Main>b\_1 to the Last extension method.
* The CLR will loop through the list and match with each element in the sequence based on condition provided in the predicate. This return on first match otherwise it will continue until find a match or not.

The approximate code for the,

Hide   Copy Code

public static TSource Last<TSource>(

this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

{

TSource local = default(TSource);

bool flag = false;

foreach (TSource local2 in source)

{

if (predicate(local2))

{

local = local2;

flag = true;

}

}

return local;

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

LastOrDefault

It returns the last element of a sequence, or a default value if no element is found. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.lastordefault)

The method signature for this extension method is as below,

Hide   Copy Code

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source)

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above extension method will for the followings,

* It returns the last element of a sequence, or a default value if the sequence contains no elements.
* It returns the last element of a sequence that satisfies a condition or a default value if no such element is found.

I wrote an example of discuss the LastOrDefault extension method,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstNumbers = new List<int>();

IList<int> secondNumbers = new List<int>()

{

1,2,3,4,5,6,7

};

var lastItemOfFirstList = firstNumbers.LastOrDefault();

var lastItemIfFirstListBasedOnConditions =

firstNumbers.LastOrDefault(item => item > 3);

var lastItemOfSecondList = secondNumbers.LastOrDefault();

var lastItemOfSecondListBasedOnConditions =

secondNumbers.LastOrDefault(item => item > 3);

Console.WriteLine("{0}\n{1}\n{2}\n{3}",

lastItemOfFirstList,

lastItemIfFirstListBasedOnConditions,

lastItemOfSecondList,

lastItemOfSecondListBasedOnConditions

);

}

}

}

This program will produce the following output:

Hide   Copy Code

0

0

7

7

The approximate code of the first version of the LastOfDefault extension method is as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source)

{

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

int count = list.Count;

if (count > 0)

{

return list[count - 1];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (enumerator.MoveNext())

{

TSource current;

do

{

current = enumerator.Current;

}

while (enumerator.MoveNext());

return current;

}

}

}

return default(TSource);

}

The approximate code of the the second version of the LastOfDefault extension method is as below,

Hide   Copy Code

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source,

Func<TSource, bool> predicate)

{

TSource local = default(TSource);

foreach (TSource local2 in source)

{

if (predicate(local2))

{

local = local2;

}

}

return local;

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

LongCount

It returns an Int64 that represents the number of elements in a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.longcount)

The method signature for this extension method is as below,

Hide   Copy Code

public static long LongCount<TSource>(this IEnumerable<TSource> source)

public static long LongCount<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

The above extension method will do as below,

* Returns an Int64 that represents the total number of elements in a sequence.
* Returns an Int64 that represents how many elements in a sequence satisfy a condition.

Let’s do an example,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstList = new List<int>()

{

1,2,3,4

};

Console.WriteLine(firstList.LongCount());

}

}

}

The program will produce following output,

Hide   Copy Code

4

The approximate code of the the first version of the LongCount extension method is as below,

Hide   Copy Code

public static long LongCount<TSource>(this IEnumerable<TSource> source)

{

long num = 0L;

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

while (enumerator.MoveNext())

{

num += 1L;

}

}

return num;

}

The approximate code of the the second version of the LongCount extension method is as below,

Hide   Copy Code

public static long LongCount<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

{

long num = 0L;

foreach (TSource local in source)

{

if (predicate(local))

{

num += 1L;

}

}

return num;

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Max

In .NET, five overloaded Max extension methods have been defined in the Enumerable class. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.max).

Hide   Copy Code

public static int Max(this IEnumerable<int> source)

public static decimal Max<TSource>(this IEnumerable<TSource> source, Func<TSource, decimal> selector)

An example of the Max extension method is as below:

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7,8,9,10

};

Console.WriteLine("Max of the numbers :{0}", numbers.Max());

Console.WriteLine("Max of the original numbers x2 :{0}", numbers.Max(x => x \* 2));

}

}

}

The above program will produce following output,

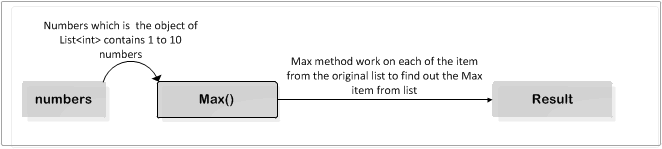
Hide   Copy Code

Max of the numbers :10

Max of the original numbers x2 :20

So when the CLR finds the first version of the Max method in the above program, it will do the following steps to perform the operation,

**Step 1**: The CLR will pass the original list as input to the Max method.



***Fig: Max extension method working details.***

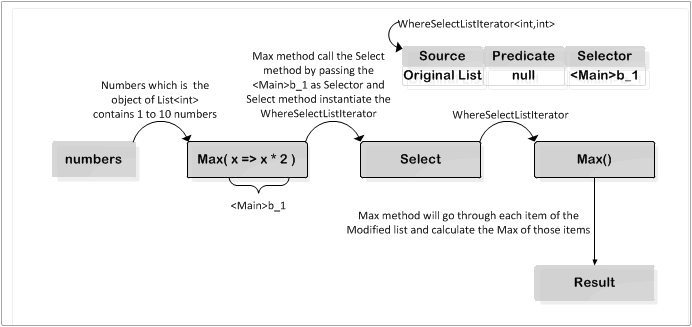
**Step 2**: The Max method will loop through the list and perform the Max operation.

The second version of the Max extension method will do the following steps,

**Step 1**: The compiler will construct a method <Main>b\_1 using the anonymous method (x => x \* 2). The CLR will pass this <Main>b\_1 method to the MulticastDelegate class to construct an instance of it and call the Select method of the list which will take original list and the instance of the MulticastDelegate class as input. It will then return the relevant iterator instance for example, for the above example it will be WhereSelectListIterator<tsource,tresult> for the list as output.

**Step 2**: It will then call Max method which will accept only the Iterator. This Iterator will contain the original list and the instance of the MulticastDelegate. In the Max method ForEach method will iterate through the list and perform the max calculation.

Following image shows:



***Fig: Max extension method working details.***

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Min

This extension method will find out the minimum of the list. Read [more](http://msdn.microsoft.com/en-us/library/bb361228). The signature of the extension methods Min are below,

Hide   Copy Code

public static int Min(this IEnumerable<int> source)

public static int Min<TSource>(this IEnumerable<TSource> source, Func<TSource, int> selector)

An example of the Min Extension method is as below,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7,8,9,10

};

Console.WriteLine("Min of the numbers :{0}", numbers.Min());

Console.WriteLine("Min of the original numbers x2 :{0}", numbers.Min(x => x \* 2));

}

}

}

The program will produce the following output,

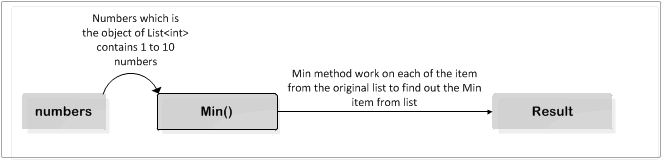
Hide   Copy Code

Min of the numbers :1

Min of the original numbers x2 :2

So when the CLR finds the first version of the Min extension method as in the above program, it will do the following steps to perform the operation,

* The CLR will pass the original list as input to the Min extension method.
* The Min method will loop through the list and perform the minimum calculation operation.



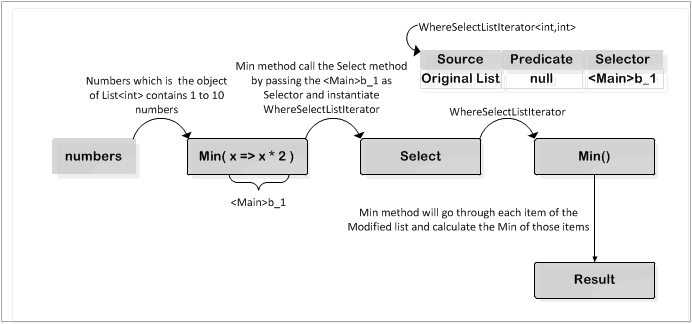
***Fig: Min extension method working details.***

The second version of the Min extension method will do the following steps,

**Step 1**: The CLR will call the Select method of the list which will take original list and the instance of the MulticastDelegate which will be instantiated using the <Main>b\_1 method. The <Main>b\_1 method will be created in the compile time based on the anonymous method (x=>x\*2). It will return the appropriate iterator which will be the WhereSelectListIterator<tsource,tresult /> and pass as input to the internal Min method.

**Step 2**: Inside the Min method CLR will perform the minimum calculation operation and return the result.

Following image shows,



***Fig: Min extension method working details.***

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

OfType

This extension method filters the elements of an IEnumerable based on a specified type using the deferred execution. The immediate return value is an object of the iterator class which stores all the information that is required to perform the action. Read [more](http://msdn.microsoft.com/en-us/library/bb360913.aspx).

The signature of this extension method is as below,

Hide   Copy Code

public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source)

Example of OfType<TResult>:

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<object> numbers = new List<object>()

{

"One",

"Two",

1,

2,

"Three",

new Person

{

Name="A Person"

}

};

var filteredNumbers = numbers.OfType<string>();

filteredNumbers.ToList().ForEach(x => Console.Write("{0}\t", x));

Console.WriteLine();

}

}

public class Person

{

public string Name { get; set; }

}

}

The above program will filter string values from the numbers list as I set string in OfType method. So the program will produce following output,

Hide   Copy Code

One Two Three

**Step 1**: The CLR will pass the sequence for example numbers in the above example to the OfType method as input. Inside the OfType method the CLR will instantiate the OfTypeIterator which will hold the original sequence inside it. The approximate code for the OfType method is as below,

Hide   Copy Code

public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source)

{

return OfTypeIterator<TResult>(source);

}

**Step 2**: The CLR will pass the instance of the OfTypeIterator<TResult> class to the ToList() method which will pass this iterator to the List class and process the operation based on the iteration logic implemented in the OfTypeIterator and produce codethe ranged sequence as output.

The approximate code for the RangeIteraor is as below,

Hide   Copy Code

private static IEnumerable<TResult> OfTypeIterator<TResult>(IEnumerable source)

{

IEnumerator enumerator = source.GetEnumerator();

while (enumerator.MoveNext())

{

object current = enumerator.Current;

if (current is TResult)

{

yield return (TResult) current;

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Range

It generates a sequence of integral numbers within a specified range, implemented by using deferred execution. The immediate return value is an instance of the relevant iterator instance that stores all the information that is required to perform the action. The method signature for this extension method is as below,

Hide   Copy Code

public static IEnumerable<int> Range(int start, int count)

Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.range.aspx).

This method will create a list of int item based on the start number to till the number of times defined in the count.

Hide   Copy Code

using System;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

Enumerable.Range(1, 10).ToList().ForEach(x => Console.Write("{0}\t", x));

}

}

}

The program will produce the following output,

Hide   Copy Code

1 2 3 4 5 6 7 8 9 10

The CLR will do the followings,

**Step 1**: The CLR will pass the start element and no of times or the length of the generated sequence to the Range method as input. Inside the Range method the CLR will return the RangeIterator<int> which will hold all the related information such as start element and length of the sequence inside it. The approximate code for the Range method is as below,

Hide   Copy Code

public static IEnumerable<int> Range(int start, int count)

{

long num = (start + count) - 1L;

if ((count < 0) || (num > 0x7fffffffL))

{

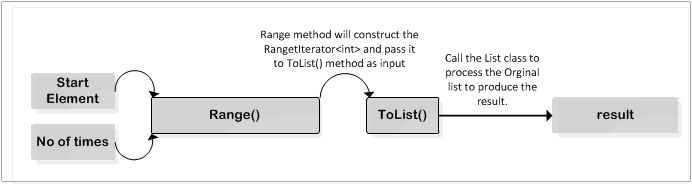
throw Error.ArgumentOutOfRange("count");

}

return RangeIterator(start, count);

}

RangeIterator<int> will not be executed (due to the deferred execution) until the CLR call the ToList() method.



***Fig: Range extension method***

**Step 2**: The CLR will pass this RangeIterator<int> to the ToList() method which will pass this iterator instance to the List class and process the operation based on the iteration logic implemented in the RangeIterator<int> class and produce the ranged sequence as output. The approximate code for the RangeIteraor<int> is as below,

Hide   Copy Code

private static IEnumerable<int> RangeIterator(int start, int count)

{

int iteratorVariable0 = 0;

while (true)

{

if (iteratorVariable0 >= count)

{

yield break;

}

yield return (start + iteratorVariable0);

iteratorVariable0++;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Repeat

It generates a sequence that contains one repeated value is implemented by using deferred execution. The immediate return value is an object of the relevant iterator type that stores all the information that is required to perform the action. Read [more](http://msdn.microsoft.com/en-us/library/bb348899.aspx)

The method signature of this extension method is as below,

Hide   Copy Code

public static IEnumerable<TResult> Repeat<TResult>(TResult element, int count)

It will generate a sequence of a number defined by the TResult type of a number of times measured by count.

Hide   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

Enumerable.Repeat(1, 5).ToList().ForEach(x=>Console.Write("{0}\t",x));

}

}

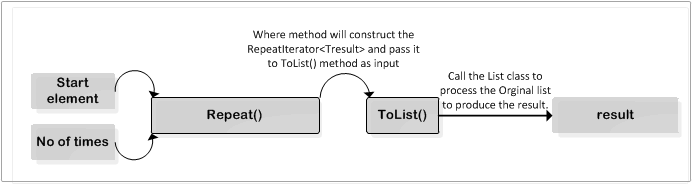
}

The Repeat method of the Enumerable class will generate a sequence of 1 at 5 times inside. It will produce the following output,

Hide   Copy Code

1 1 1 1 1

**Step 1**: The CLR will pass the element to repeat and no of times to repeat number to the Repeat method as input. Inside the Repeat method it will construct the RepeatIterator<TResult> iterator which will hold all the related information to generate the sequence.



***Fig: Repeat extension method.***

**Step 2**: The CLR will pass this RepeatIterator<TResult> instance to the ToList() method which will pass this iterator to the List class and process the operation based on the iteration logic implemented in the RepeatIterator<TResult> and produce the repeated sequence as output.

The approximate code for the Repeat method is as below,

Hide   Copy Code

private static IEnumerable<TResult> RepeatIterator<TResult>(TResult element, int count)

{

int iteratorVariable0 = 0;

while (true)

{

if (iteratorVariable0 >= count)

{

yield break;

}

yield return element;

iteratorVariable0++;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Reverse

It inverts the order of the elements in a sequence is implemented by using deferred execution. The immediate return value is an object of the iterator type that stores all the information that is required to perform the action. Read [more](http://msdn.microsoft.com/en-us/library/bb358497.aspx)

Unlike OrderBy, this sorting method does not consider the actual values themselves in determining the order. Rather, it just returns the elements in the reverse order from which they are produced by the underlying source.

Hide   Copy Code

public static IEnumerable<TSource> Reverse<TSource>(this IEnumerable<TSource> source)

It will reverse the original list, following example shows the usage of the Reverse extension method,

Hide   Copy Code

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };

var reverseNumbers = numbers.Reverse();

var result = reverseNumbers.ToList();

result.ForEach(x => Console.Write("{0}\t", x));

Console.WriteLine();

}

}

}

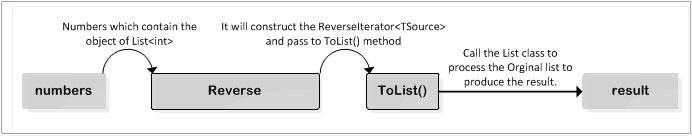
This program will produce the following output,

Hide   Copy Code

5 4 3 2 1

While the CLR will execute the above program, to process the Reverse method it will do the following steps,

**Step 1**: The CLR will pass the original sequence in this case the numbers object as input to the Reverse method. Inside the Reverse method it will construct the ReverseIterator<TSource> iterator which will hold all the information related to the original sequence.



***Fig: Reverse extension method.***

**Step 2**: The CLR will pass the ReverseIterator<TSource> instance to the ToList() method which will pass this iterator to the List class and process the operation based on the iteration logic implemented in the ReverseIterator<TSource> and produce the reversed sequence as output.

Approximate code for the ReverseIterator<TSource>:

Hide   Copy Code

private static IEnumerable<TSource> ReverseIterator<TSource>(

IEnumerable<TSource> source)

{

Buffer<TSource> iteratorVariable0 = new Buffer<TSource>(source);

int index = iteratorVariable0.count - 1;

while (true)

{

if (index < 0)

{

yield break;

}

yield return iteratorVariable0.items[index];

index--;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Single

This extension method returns a single, specific element of a sequence. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.single.aspx).

The method signature for the extension method is as below,

Hide   Copy Code

public static TSource Single<TSource>(this IEnumerable<TSource> source)

public static TSource Single<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source)

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source,Func<TSource, bool> predicate)

The above extension method will do the followings,

* It returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence.
* It returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists.

I wrote a small program to test the Single extension method. In this program I will use the Single method over the IList<string> object numbers which contains only one item One. Following example will return One as output because this is the exactly one item in the list and the Single method works only those List object which contains exactly 1 item inside,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>

{

"One"

};

var result = numbers.Single();

Console.WriteLine("{0}", result);

}

}

}

The above program will produce following output,

Hide   Copy Code

One

If I modify the above code and add one more item in the numbers list and execute the program then we will get following error message,

Hide   Copy Code

Unhandled Exception: System.InvalidOperationException: Sequence contains more th

an one element

at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source)

at Chapter\_5.Program.Main(String[] args) in J:\Book\How Does it Work in C#\Bo

ok-Projects\HDIWIC\Chapter-5\Program.cs:line 16

Let’s now find out how does it works behind the scence,

**Step 1**: The CLR will make a new List<string> object using a copy of the original list and check whether the new list is null or not. If it is not null then it will check the number of items in the list. If the number of items in the list is 0 then the CLR will throw an exception otherwise it 1 then return the first and only item from the list. The approximate code will be as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

public static TSource Single<TSource>(this IEnumerable<TSource> source)

{

if (source == null)

{

throw Error.ArgumentNull("source");

}

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

switch (list.Count)

{

case 0:

throw Error.NoElements();

case 1:

return list[0];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (!enumerator.MoveNext())

{

throw Error.NoElements();

}

TSource current = enumerator.Current;

if (!enumerator.MoveNext())

{

return current;

}

}

}

throw Error.MoreThanOneElement();

}

Let’s try the Single extension method with a predicate function. I wrote a small program as below,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>

{

"One","Four"

};

var result = numbers.Single(x => x.Length > 3);

Console.WriteLine("{0}", result);

}

}

}

This program will produce the following output,

Hide   Copy Code

Four

If I change the numbers list by adding one more item whose length is more than three character the program will fail by throwing following exception,

Hide   Copy Code

Unhandled Exception: System.InvalidOperationException: Sequence contains more th

an one matching element

at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source, Func`2 predic

ate)

at Chapter\_5.Program.Main(String[] args) in J:\Book\How Does it Work in C#\Bo

ok-Projects\HDIWIC\Chapter-5\Program.cs:line 16

I wrote an example of the SingleOrDefault extension method as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> listStringWithoutItem = new List<string>();

IList<string> listStringWithItem = new List<string>() { "One" };

IList<int> listInt = new List<int>();

IList<char> listChar = new List<char>();

IList<long> listLong = new List<long>();

IList<double> listDouble = new List<double>();

var resultStringWithoutItem = listStringWithoutItem.SingleOrDefault();

var resultStringWithItem = listStringWithItem.SingleOrDefault();

var resultInt = listInt.SingleOrDefault();

var resultChar = listChar.SingleOrDefault();

var resultLong = listLong.SingleOrDefault();

var resultDouble = listDouble.SingleOrDefault();

Console.WriteLine("string : {0}", resultStringWithoutItem);

Console.WriteLine("string : {0}", resultStringWithItem);

Console.WriteLine("int : {0}", resultInt);

Console.WriteLine("char : {0}", resultChar);

Console.WriteLine("long : {0}", resultLong);

Console.WriteLine("double : {0}", resultDouble);

}

}

}

The above will produce following output:

Hide   Copy Code

string :

string : One

int : 0

char :

long : 0

double : 0

The CLR will execute the SingleOrDefault extension with following steps,

**Step 1**: The CLR will check whether the list is null or not. If not then make a copy of the original into a temporary list. It will check the number of items in the list if it is 0 then the CLR will return the default value of the provided type for example, string for the listStringWithoutItem.SingleOrDefault<string>() or inferred type from the list for example, listStringWithoutItem is a type of IList<string> so the inferred type will be string . Complete list default value Table is [here](http://msdn.microsoft.com/en-us/library/83fhsxwc.aspx)

The approximate code for the SingleOrDefault extension method is as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source)

{

IList<TSource> list = source as IList<TSource>;

if (list != null)

{

switch (list.Count)

{

case 0:

return default(TSource);

case 1:

return list[0];

}

}

else

{

using (IEnumerator<TSource> enumerator = source.GetEnumerator())

{

if (!enumerator.MoveNext())

{

return default(TSource);

}

TSource current = enumerator.Current;

if (!enumerator.MoveNext())

{

return current;

}

}

}

throw Error.MoreThanOneElement();

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Skip

It bypasses a specified number of elements in a sequence and then returns the remaining elements is implemented by using deferred execution. The immediate return value is an object of the relevant type that stores all the information that is required to perform the action. Read [more](http://msdn.microsoft.com/en-us/library/bb358985.aspx)

Skip method will iterate through the list and skip the specified number of items from the beginning of the list. The specified number will accept as parameter of the method.

Hide   Copy Code

public static IEnumerable<TSource> Skip<TSource>(this IEnumerable<TSource> source, int count)

The following program will create list of string type which will hold One, Two, Three, Four and Five as items of this list, On the list I applied the skip operation by providing 2 as the number of items to skip,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One","Two","Three", "Four","Five"

};

var result = numbers.Skip(2);

result.ToList().ForEach(number => Console.WriteLine(number));

}

}

}

The program will produce following output,

Hide   Copy Code

Three

Four

Five

So the CLR will find the above code specially numbers.Skip(1) it will,

**Step 1**: The CLR will go to the Skip method of Enumerable class from the System.Linq namespace. This method will return the instance of the SkipIterator<TSource> which will hold the original list and the count which will define how many item to skip.

**Step 2**: As because of the deferred execution pattern, this SkipIterator<TSource> will execute while for iterate it via the ToList() method. So inside the SkipIterator<tsource> method it will run a loop until the number of item to skip becomes 0. During this iteration it will move the current position of the inner Enumerator object. While the number of item becomes 0 then it will loop through the list again to return the remaining item from the list.

The approximate code of the SkipIterator is as below,

Hide   Copy Code

private static IEnumerable<TSource> SkipIterator<TSource>(

IEnumerable<TSource> source, int count)

{

using (IEnumerator<TSource> iteratorVariable0 = source.GetEnumerator())

{

while ((count > 0) && iteratorVariable0.MoveNext())

{

count--;

}

if (count <= 0)

{

while (iteratorVariable0.MoveNext())

{

yield return iteratorVariable0.Current;

}

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

SkipWhile

This extension method bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.skipwhile.aspx).

The method signature of this extension method is as below,

Hide   Copy Code

public static IEnumerable<TSource> SkipWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

public static IEnumerable<TSource> SkipWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate)

Let’s see an example of the SKipWhile extension method which will help to understand of this extension method,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<string> numbers = new List<string>()

{

"One","Two","Three", "Four","Five"

};

var result = numbers.SkipWhile(number => number.Length == 3);

result.ToList().ForEach(number => Console.WriteLine(number));

}

}

}

The above program will produce the following output,

Hide   Copy Code

Three

Four

Five

The output shows the result list excluded those items whose Length is equal to 3. So while the CLR find the SkipWhile method it will do the followings,

**Step 1**: The compiler will construct a method <Main>b\_1 using the anonymous method (number => number.Length == 3) in the compile time. The CLR will pass this <Main>b\_1 method to the MulticastDelegate class to instantiate an instance of it. So the CLR will pass the original list and predicate in this <Main>b\_1 as input to the SkipWhile method and it will return SkipWhileIterator which will hold the original list and <Main>b\_1 as predicate.

**Step 2**: Inside SkipWhileIterator, CLR will loop through the original list one by one and execute the predicate over the item. If the predicate return false then it will return that item as a result item of the SkipWhile method or otherwise if it return true then it will keep continue through the list until it finishes.

The approximate code for the SkipWhileIterator will be as below,

Hide   Copy Code

private static IEnumerable<TSource> SkipWhileIterator<TSource>(

IEnumerable<TSource> source, Func<TSource, bool> predicate)

{

bool iteratorVariable0 = false;

foreach (TSource iteratorVariable1 in source)

{

if (!iteratorVariable0 && !predicate(iteratorVariable1))

{

iteratorVariable0 = true;

}

if (iteratorVariable0)

{

yield return iteratorVariable1;

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Sum

To sum all the items inside a list we can use this Sum extension method. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.sum.aspx) The signature of those methods is below,

Hide   Copy Code

public static int Sum(this IEnumerable<int> source)

public static int Sum<TSource>(this IEnumerable<TSource> source, Func<TSource, int> selector)

An example of the Min Extension method is as below,

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7,8,9,10

};

Console.WriteLine("Sum of the numbers :{0}", numbers.Sum());

Console.WriteLine("Sum of the original numbers x2 :{0}",

numbers.Sum(x => x \* 2));

}

}

}

The program will produce the following output,

Hide   Copy Code

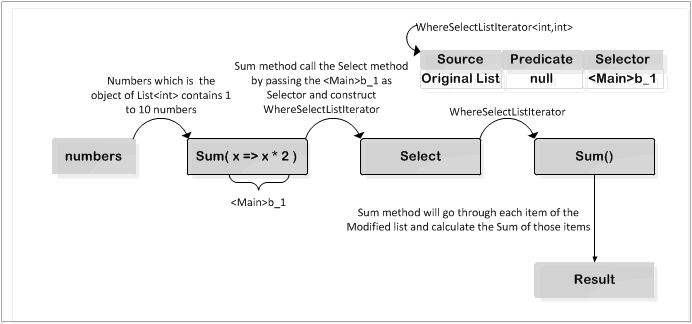
Sum of the numbers :55

Sum of the original numbers x2 :110

To execute the first version of the Sum extension method used in the above program, CLR will do the following steps to perform the operation,

**Step 1**: The CLR will pass the original list as input to the Sum extension method.

**Step 2**: Inside the Sum method it will loop through the list and perform the summation of each of the item and produced the result and return as output of the Sum.



***Fig: Sum Extension method***

The approximate of the Sum method is as below,

Hide   Copy Code

public static int Sum(this IEnumerable<int> source)

{

if (source == null)

{

throw Error.ArgumentNull("source");

}

int num = 0;

foreach (int num2 in source)

{

num += num2;

}

return num;

}

To execute the second version of the Sum extension method CLR will execute following operations,

**Step 1**: The compiler will create a method <Main>b\_1 using the anonymous method (x => x \* 2) code.

Note: When I decompiled the produced executable from the above program using [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx), there will be a method:

Hide   Copy Code

.method private hidebysig static int32 <Main>b\_\_1(int32 x) cil managed

{

.maxstack 2

.locals init (

[0] int32 CS$1$0000)

L\_0000: ldarg.0

L\_0001: ldc.i4.2

L\_0002: mul

L\_0003: stloc.0

L\_0004: br.s L\_0006

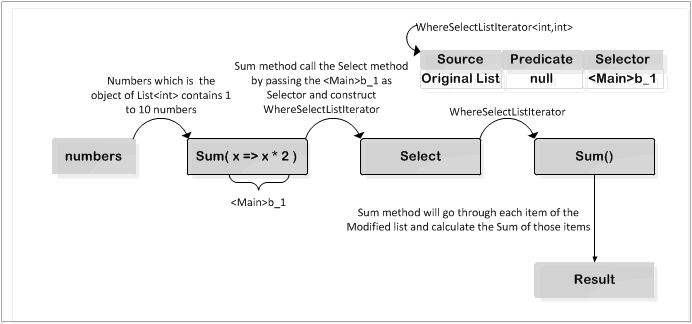
L\_0006: ldloc.0

L\_0007: ret

}

The CLR will create an instance of MulticastDelegate instance using the <Main>b\_1 method.

**Step 2**: The CLR will pass the original list and MulticastDelegate instance created in the **Step 1** as input to the Sum method which will call the Select method with original list and the MulticastDelegate object as input. The CLR will instantiate the relevant iterator and return back to the Sum method. The CLR will then call the overloaded Sum() method.



***Fig: Sum extension method working details.***

**Step 3**: The CLR will iterate through the item from the original list based on the iterator and execute the given selector ie the delegate instance created in the **Step 1** and sum all the modified item to complete the summation operation,

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

ThenBy

ThenBy extension method performs a subsequent ordering of the elements in a sequence in ascending order. This extension method is implemented by using deferred execution. The immediate return value is an object of the relevant type that stores all the information that is required to perform the action. Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.thenby.aspx).

The signature of the ThenBy extension methods is as below,

Hide   Copy Code

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(

this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector, IComparer<TKey> comparer)

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(

this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector)

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(

this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector, IComparer<TKey> comparer)

I wrote this following program to explain the ThenyBy extension method,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<Person> persons = new List<Person>()

{

new Person(){ Name="Person F", Address= "Address of F", Id= 111116},

new Person(){ Name="Person G", Address= "Address of G", Id= 111117},

new Person(){ Name="Person C", Address= "Address of C", Id= 111113},

new Person(){ Name="Person B", Address= "Address of B", Id= 111112},

new Person(){ Name="Person D", Address= "Address of D", Id= 111114},

new Person(){ Name="Person A", Address= "Address of A", Id= 111111},

new Person(){ Name="Person E", Address= "Address of E", Id= 111115}

};

var result = persons.OrderBy(person => person.Id).ThenBy(person => person);

foreach (Person person in result)

{

Console.WriteLine("{0,-15} {1,-20}{2,-20}",

person.Name,

person.Address,

person.Id);

}

}

}

public class Person

{

public string Name

{

get;

set;

}

public string Address

{

get;

set;

}

public double Id

{

get;

set;

}

}

}

The above program will produce the following output,

Hide   Copy Code

Person A Address of A 111111

Person B Address of B 111112

Person C Address of C 111113

Person D Address of D 111114

Person E Address of E 111115

Person F Address of F 111116

Person G Address of G 111117

It will work as,

**Step 1**: Code: Approximate code for the ThenBy method.

Hide   Copy Code

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(

this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector)

{

return source.CreateOrderedEnumerable<TKey>(keySelector, null, false);

}

**Step 2**:

Hide   Copy Code

IOrderedEnumerable<TElement> IOrderedEnumerable<TElement>.CreateOrderedEnumerable<TKey>(

Func<TElement, TKey> keySelector, IComparer<TKey> comparer, bool descending)

{

return new OrderedEnumerable<TElement, TKey>(this.source, keySelector,

comparer, descending) { parent = (OrderedEnumerable<TElement>) this };

}

**Step 2**: The implementation of the Order

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

internal class OrderedEnumerable<TElement, TKey> : OrderedEnumerable<TElement>

{

internal IComparer<TKey> comparer;

internal bool descending;

internal Func<TElement, TKey> keySelector;

internal OrderedEnumerable<TElement> parent;

internal OrderedEnumerable(IEnumerable<TElement> source,

Func<TElement, TKey> keySelector,

IComparer<TKey> comparer,

bool descending)

{

base.source = source;

this.parent = null;

this.keySelector = keySelector;

this.comparer = (comparer != null) ? comparer : ((IComparer<TKey>) Comparer<TKey>.Default);

this.descending = descending;

}

internal override EnumerableSorter<TElement> GetEnumerableSorter(

EnumerableSorter<TElement> next)

{

EnumerableSorter<TElement> enumerableSorter = new

EnumerableSorter<TElement, TKey>(

this.keySelector,

this.comparer,

this.descending, next);

if (this.parent != null)

{

enumerableSorter = this.parent.GetEnumerableSorter(enumerableSorter);

}

return enumerableSorter;

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

ToArray

It will create an array from the list. Following program will show the usage of the ToArray() method. The method signature is as below,

Hide   Copy Code

public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)

ToList<TSource> has similar behavior but returns a List<T> instead of an array.

Read [more](http://msdn.microsoft.com/en-us/library/bb298736.aspx).

Let’s see an example of the ToArray()

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstList = new List<int>()

{

1,2,3,4

};

var result = firstList.ToArray();

result.ToList().ForEach(x => Console.WriteLine(x));

}

}

}

This program will produce the following output:

Hide   Copy Code

1

2

3

4

The CLR will execute the above as below,

**Step 1**: The CLR pass the original list as input to the ToArray<TSource>(this IEnumerable<TSource> source) method as input, inside the ToArray method it will create an instance of the Buffer<TSource> type by passing the original list object as input.

**Step 2**: The CLR will copy the each of the item from the original list to an internal array named items. The approximate code for the ToArray() method will be as below,

Hide   Copy Code

public static TSource[] ToArray<TSource>(this IEnumerable<TSource> source)

{

Buffer<TSource> buffer = new Buffer<TSource>(source);

return buffer.ToArray();

}

The internal of the Buffer<TSource> structure will be as below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

internal struct Buffer<TElement>

{

internal TElement[] items;

internal int count;

internal Buffer(IEnumerable<TElement> source)

{

TElement[] array = null;

int length = 0;

ICollection<TElement> i.= source as ICollection<TElement>;

if (i.!= null)

{

length = i..Count;

if (length > 0)

{

array = new TElement[length];

i..CopyTo(array, 0);

}

}

else

{

foreach (TElement local in source)

{

if (array == null)

{

array = new TElement[4];

}

else if (array.Length == length)

{

TElement[] destinationArray = new TElement[length \* 2];

Array.Copy(array, 0, destinationArray, 0, length);

array = destinationArray;

}

array[length] = local;

length++;

}

}

this.items = array;

this.count = length;

}

}

**Step 3**: And finally when the CLR calls the ToArray method, it will return:

Hide   Copy Code

internal TElement[] ToArray()

{

if (this.count == 0)

{

return new TElement[0];

}

if (this.items.Length == this.count)

{

return this.items;

}

TElement[] destinationArray = new TElement[this.count];

Array.Copy(this.items, 0, destinationArray, 0, this.count);

return destinationArray;

}

A copy of the items array as output of the ToArray method.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

ToDictionary

It creates a Dictionary<tkey,> from an IEnumerable<T>. If we want to create a dictionary object based on the data in a list this method will do everything by self but we need to specify a field from the list data as a key.

The signature of the ToDictionary extension is as below:

Hide   Copy Code

public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(

this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)

public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(

this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector, Func<TSource, TElement> elementSelector)

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector, Func<TSource, TElement> elementSelector, IEqualityComparer<TKey> comparer)

Read [more](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.todictionary.aspx).

Let’s see an example which will help to understand this ToDictionary method easily,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png  Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<Person> persons = new List<Person>()

{

new Person(){ Name="Person A", Address= "Address of A", Id= 111111},

new Person(){ Name="Person B", Address= "Address of B", Id= 111112},

new Person(){ Name="Person C", Address= "Address of C", Id= 111113},

new Person(){ Name="Person D", Address= "Address of D", Id= 111114},

};

var result = persons.ToDictionary(person => person.Id);

foreach (KeyValuePair<double, Person> person in result)

{

Console.WriteLine("{0,-15} {1,-20}{2,-20}{3,-20}",

person.Key,

person.Value.Name,

person.Value.Address,

person.Value.Id);

}

}

}

public class Person

{

public string Name

{

get;

set;

}

public string Address

{

get;

set;

}

public double Id

{

get;

set;

}

}

}

The above program will produce the following output,

Hide   Copy Code

111111 Person A Address of A 111111

111112 Person B Address of B 111112

111113 Person C Address of C 111113

111114 Person D Address of D 111114

I created a list of Person object and stored into a List object persons. Then I turned this persons list into a Dictionary using ToDictionary extension method. As we can see ToDictionary is taking an anonymous method as input. This anonymous method is actually a key selector which will select the key of the object from the list and set as key into the dictionary object. So from Person object, Id will be selected as key for the result dictionary and value will be person object itself, interesting enough the Id property will be used as Key for the Dictionary and also it will be stored into the person as it was initialized.

When the CLR do the following to execute the ToDictionary method,

**Step 1**: If we open the System.Linq.Enumerable namespace from the *C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Core.dll* assembly using [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program then we can see that the ToDictionary method interanlly call interanl ToDictionary method which has the following signature,

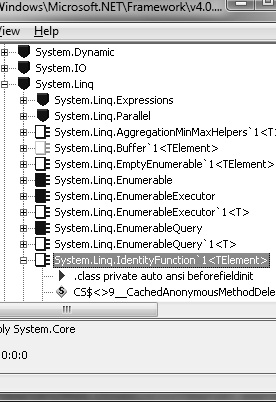
Hide   Copy Code

public static Dictionary<tkey,> ToDictionary<tsource,>( this IEnumerable<tsource> source,

Func<tsource,> keySelector, Func<tsource,> elementSelector, IEqualityComparer<tkey> comparer)

Before the CLR call the above internal ToDictionary method from the ToDictionary extension method, it will create an element selector function. In this case though I haven't provided any element selector the CLR will use the default element selector which is IdentityFunction<TSource>.Instance.

Note:IdentityFunction<TSource>.Instance is an internal class which will be used as an element selector for the ToDictionary method. Following diagram shows this class from the *System.Core.dll* assembly,



***Fig: IdentityFunction***

Hide   Copy Code

public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(

this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector)

{

return source.ToDictionary<TSource, TKey, TSource>(

keySelector, IdentityFunction<TSource>.Instance, null);

}

**Step 2**: When the CLR goes to the above method, this method call overloaded ToDictionary as below,

Hide   Copy Code

public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(

this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

Func<TSource, TElement> elementSelector,

IEqualityComparer<TKey> comparer)

{

Dictionary<TKey, TElement> dictionary = new Dictionary<TKey, TElement>(comparer);

foreach (TSource local in source)

{

dictionary.Add(keySelector(local), elementSelector(local));

}

return dictionary;

}

Which will instantiate an instance of the Dictionary<TKey, TElement> class. It will then iterate through the original list, each of the iterate value will pass to the KeySelector and ElementSelector function to extract the Key and Value from the iterate value.

As I provided (person => person.Id) as the KeySelector, the compiler will generate an anonymous method <Main>b\_5 and pass to the KeySelector which will return the Id from the person object and compiler will provide the ElementSelector ( x=>x as in the IdentityFunction<telement> in where x=>x will converted as <get\_instance>b\_\_0</get\_instance>) which will return the value itself, i.e., the person object with Name, Address and Id value inside.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

ToList

It creates a List<T> from an IEnumerable<T>. Read [more](http://msdn.microsoft.com/en-us/library/bb342261.aspx)

The method signature for this extension method is as below,

Hide   Copy Code

public static List<TSource> ToList<TSource>(this IEnumerable<TSource> source)

To explain the working details of the ToList() extension method let’s see an example which is ToList() to produce the result,

I wrote a small program to show the usage of the ToList( this IEnumerable<TSource> collection) extension methods,

Hide   Copy Code

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> numbers = new List<int>()

{

1,2,3,4,5,6,7,8,9,10

};

var result = numbers.Where(x => x > 3).ToList();

result.ForEach(x => Console.Write("{0}\t", x));

Console.WriteLine();

}

}

}

The above program will produce following output as result,

Hide   Copy Code

4 5 6 7 8 9 10

So the CLR will execute the ToList() as below,

**Step 1**: The ToList() extension method will accept an IEnumerable<tsrouce> object as input. It will pass this IEnumerable object as input to the List<TSource> type. The approximate code of that will be as below,

Hide   Copy Code

public static List<TSource> ToList<TSource>(this IEnumerable<TSource> source)

{

return new List<TSource>(source);

}

**Step 2**: The List<ienumerable<tsource>> type will accept an IEnumerable<TSource> collection as input of the constructor. Inside the constructor the CLR will initialize the \_items array with the type as TSource and define the initial size of the array with 4. It will then iterate through the enumerator of the input list object. The approximate code for the List constructor is be as below, </ienumerable<tsource>>

Hide   Copy Code

public List(IEnumerable<T> collection)

{

ICollection<T> i. = collection as ICollection<T>;

if (i. != null)

{

int count = i..Count;

this.\_items = new T[count];

i..CopyTo(this.\_items, 0);

this.\_size = count;

}

else

{

this.\_size = 0;

this.\_items = new T[4];

using (IEnumerator<T> enumerator = collection.GetEnumerator())

{

while (enumerator.MoveNext())

{

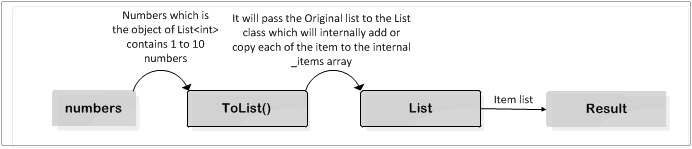
this.Add(enumerator.Current);

}

}

}

}



***Fig: ToList() extension method working details.***

**Step 3**: In the iteration phase each of the item the CLR will retrieve pass to the Add( TSource item) method to add into the \_items array initialized (in the **Step 2**). The approximate code for the Add method is as below,

Hide   Copy Code

public void Add(T item)

{

if (this.\_size == this.\_items.Length)

{

this.EnsureCapacity(this.\_size + 1);

}

this.\_items[this.\_size++] = item;

this.\_version++;

}

In the Add method the most import code is the line this.EnsureCapacity(this.\_size + 1). The size of this \_items array is dynamic and it will be ensured by the EnsureCapacity method.

**Step 4**: So after finishing the iteration the CLR will return the list object as the output of the ToList() method which will contain elements returned from the given IEnumerable<TSource> object inside the \_items array.

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Zip

It applies a specified function to the corresponding elements of two sequences, producing a sequence of the results. The method steps through the two input sequences, applying function resultSelector to corresponding elements of the two sequences. The method returns a sequence of the values that are returned by resultSelector. If the input sequences do not have the same number of elements, the method combines elements until it reaches the end of one of the sequences. For example, if one sequence has three elements and the other one has four, the result sequence has only three elements. The Zip extension method will combine two list items by item based on the provided combination logic. Based on the following method signature we can see it’s an extension of IEnumerable<TFirst> type and accept IEnumerable<TSecond> second, Func<TFirst, TSecond, TResult> resultSelector items as input.

The signature of this extension method is as below,

Hide   Copy Code

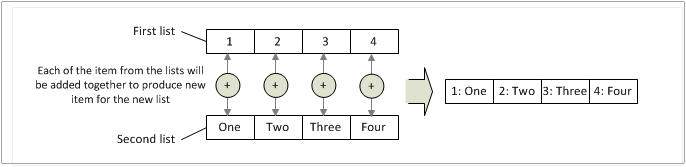
public static IEnumerable<TResult> Zip<TFirst, TSecond, TResult>(

this IEnumerable<TFirst> first, IEnumerable<TSecond> second,

Func<TFirst, TSecond, TResult> resultSelector)

Read [more](http://msdn.microsoft.com/en-us/library/dd267698.aspx)

So the items of first and second list will be combined item by item together to produce a new list based on the combined logic provided into the resultSelector Func. So we can the Zip method will combine each of the item in the list as below,



***Fig: An example of Zip extension method.***

Based on the above diagram I wrote a small program which will combine the firstList which contains {1, 2, 3, 4} item with the secondList which contains {“One”,”Two”,”Three”,”Four”} with the combined logic, item from the first List + “:\t” + item from the secondList.

Hide   Copy Code

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Chapter\_5

{

class Program

{

static void Main(string[] args)

{

IList<int> firstList = new List<int>()

{

1,2,3,4

};

IList<string> secondList = new List<string>()

{

"One","Two","Three","Four"

};

var result = firstList.Zip(secondList, (x, y) => x + ":\t" + y);

result.ToList().ForEach(x => Console.WriteLine(x));

}

}

}

This program will produce following output as a result,

Hide   Copy Code

1: One

2: Two

3: Three

4: Four

So when the Compiler finds the Zip method it will,

**Step 1**: The compiler will construct a method <Main>b\_2 using the anonymous method (x, y) => x + ":\t" + y. The CLR will pass this <Main>b\_2 method to the MulticastDelegate class to instantiate an instance of it.

**Note**: If we decompile the executable produced by the program and drop into the [ILDasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.80).aspx) program we can see that the compiler generated following method block for the (x, y) => x + ":\t" + y code,

Hide   Copy Code

.method private hidebysig static string '<Main>b\_\_2'(int32 x,

string y) cil managed

{

*// Code size 22 (0x16)*

.maxstack 3

.locals init ([0] string CS$1$0000)

IL\_0000: ldarg.0

IL\_0001: box [mscorlib]System.Int32

IL\_0006: ldstr ":\t"

IL\_000b: ldarg.1

IL\_000c: call string [mscorlib]System.String::Concat(object,

object,

object)

IL\_0011: stloc.0

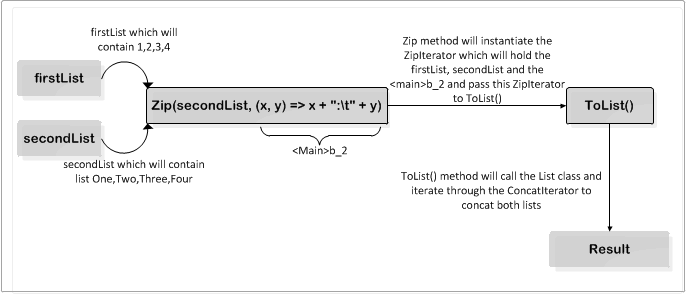
IL\_0012: br.s IL\_0014

IL\_0014: ldloc.0

IL\_0015: ret

} *// end of method Program::'<Main>b\_\_2'*

**Step 2**: The CLR will pass the instance of the MulticastDelegate created in the step1 to the Zip method which will return the ZipIterator<tfirst, /> instance after doing few basic null checks. The ZipIterator<tfirst, /> instance will hold the first and second list and resultSelector (instance of the MulticastDelegate created in step1) inside it.



***Fig: The Zip method working details***

**Step 3**: As this Zip extension method will execute using deferred execution pattern, whenever the CLR execute the ToList() method it will iterate through the ZipIterator enumerator. Inside the ZipIteraor enumerator CLR will iterate through each of the List and get the Current item from the each list and it will pass that Current item as input to the resultSelector Func as the input. The resultSelector will then combine each of the provided items into one single item (for example, 1 from the firstList and One from the secondList will be combined as 1: One) and return. This will continue until the both list has finished. In this iteration process if one of the list has less item than the other then it will only return same amount of item from the both list. For example, if list A has {A1,B1,C1,D1} items and B has{A2,B2,C2} then the result will be based on combination logic (+) processed result {A1A2, B1B2,C1C2}. The D1 from the A list will be deducted.

The approximate code for the Zip extension is as below,

Hide   Copy Code

public static IEnumerable<TResult> Zip<TFirst, TSecond, TResult>(

this IEnumerable<TFirst> first,

IEnumerable<TSecond> second,

Func<TFirst, TSecond, TResult> resultSelector)

{

return ZipIterator<TFirst, TSecond, TResult>(first, second, resultSelector);

}

private static IEnumerable<TResult> ZipIterator<TFirst, TSecond, TResult>(

IEnumerable<TFirst> first,

IEnumerable<TSecond> second,

Func<TFirst, TSecond, TResult> resultSelector)

{

using (IEnumerator<TFirst> iteratorVariable0 = first.GetEnumerator())

{

using (IEnumerator<TSecond> iteratorVariable1 = second.GetEnumerator())

{

while (iteratorVariable0.MoveNext() && iteratorVariable1.MoveNext())

{

yield return resultSelector(

iteratorVariable0.Current,

iteratorVariable1.Current);

}

}

}

}

[Go to the contents.](https://www.codeproject.com/Articles/383749/How-does-it-work-in#tocContents)

Related articles

* [How does it work in C#? - Part 1](http://www.codeproject.com/Articles/266593/How-does-it-work-in-Csharp)
* [How does it work in C#? - Part 2](http://www.codeproject.com/Articles/267701/How-does-it-work-in-Csharp-Part-2)

History

* Added Extension method working details in behind the scene.
* PDF file has been updated.
* Table of contents updated.
* External link has been added for the Min extension.
* Table of contents updated.
* PDF file has been removed.

<a href="https://pubads.g.doubleclick.net/gampad/jump?iu=/6839/lqm.codeproject.site/Languages/Csharp/General&sz=300x250&c=478856"><img src="https://pubads.g.doubleclick.net/gampad/jump?iu=/6839/lqm.codeproject.site/Languages/Csharp/General&sz=300x250&c=478856" width="300px" height="250px" target="\_blank"/></a>

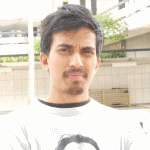
License

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)

Share

* [twitter](http://twitter.com/home?status=How+does+it+work+in+C%23%3f+-+Part+3+(C%23+LINQ+in+detail)+-+CodeProject+-+https%3a%2f%2fwww.codeproject.com%2fArticles%2f383749%2fHow-does-it-work-in)
* [facebook](https://www.facebook.com/sharer/sharer.php?u=https%3a%2f%2fwww.codeproject.com%2fArticles%2f383749%2fHow-does-it-work-in)
* [linkedin](http://www.linkedin.com/shareArticle?mini=true&url=https%3a%2f%2fwww.codeproject.com%2fArticles%2f383749%2fHow-does-it-work-in&title=How+does+it+work+in+C%23%3f+-+Part+3+(C%23+LINQ+in+detail)+-+CodeProject&summary=&source=codeproject.com)
* [reddit](http://www.reddit.com/submit?url=https%3a%2f%2fwww.codeproject.com%2fArticles%2f383749%2fHow-does-it-work-in&title=How+does+it+work+in+C%23%3f+-+Part+3+(C%23+LINQ+in+detail)+-+CodeProject)
* [google+](https://plus.google.com/share?url=https%3a%2f%2fwww.codeproject.com%2fArticles%2f383749%2fHow-does-it-work-in)

About the Author



[**Mohammad A Rahman**](https://www.codeproject.com/Members/Mohammad-A-Rahman)

|  |  |
| --- | --- |
|  | Software Developer |
| Australia Australia |

Designer and Architect.  
Author of the

* [**C# Deconstructed: Discover how C# works on the .NET Framework**](http://www.apress.com/us/book/9781430266709)
* [**Expert C# 5.0: with the .NET 4.5 Framework**](http://www.apress.com/us/book/9781430248606)