You have questions about the **List** collection in the .NET Framework, which is located in the System.Collections.Generic namespace. You want to see examples of using List and also explore some of the many useful methods it provides, making it an ideal type for dynamically adding data. This document has lots of tips and resources on the List constructed type, with examples using the C# programming language.

***Lists*** *are dynamic arrays in the C# language.  
They can grow as needed when you add elements.  
They are considered generics and constructed types.  
You need to use < and > in the List declaration.*

**Adding values**

To start, we see how to declare a new List of int values and add integers to it. This example shows how you can create a new List of unspecified size, and add four prime numbers to it. Importantly, the angle brackets are part of the declaration type, not conditional operators that mean less or more than. They are treated differently in the language.

**~~~ Program that adds elements to List (C#) ~~~**

using System.Collections.Generic;

class Program

{

static void Main()

{

**List**<int> list = new List<int>();

list.**Add**(2);

list.**Add**(3);

list.**Add**(5);

list.**Add**(7);

}

}

**Adding objects.** The above example shows how you can add a primitive type such as integer to a List collection, but the List collection can receive reference types and object instances. There is more information on adding objects with the Add method on this site.

[**See List Add Method.**](http://dotnetperls.com/list-add)

**Loops**

Here we see how you can loop through your List with for and foreach loops. This is a very common operation when using List. The syntax is the same as that for an array, except your use Count, not Length for the upper bound. You can also loop backwards through your List by reversing the for loop iteration variables. Start with list.Count - 1, and proceed decrementing to >= 0.

**~~~ Program that loops through List (C#) ~~~**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

List<int> list = new List<int>();

list.Add(2);

list.Add(3);

list.Add(7);

**foreach** (int prime in list) // Loop through List with foreach

{

Console.WriteLine(prime);

}

**for** (int i = 0; i < list.Count; i++) // Loop through List with for

{

Console.WriteLine(list[i]);

}

}

}

**~~~ Output of the program ~~~**

(Repeated twice)

2

3

7

**Counting elements**

To get the number of elements in your List, access the Count property. This is fast to access, if you avoid the Count() extension method. Count is equal to Length on arrays. See the section "Clearing List" for an example on using the Count property.

**Clearing List—setting to null**

Here we see how to use the Clear method, along with the Count property, to erase all the elements in your List. Before Clear is called, this List has 3 elements; after Clear is called, it has 0 elements. Alternatively, you can assign the List to null instead of calling Clear, with similar performance. However, after assigning to null, you must call the constructor again.

**=== Program that counts List (C#) ===**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

List<bool> list = new List<bool>();

list.Add(true);

list.Add(false);

list.Add(true);

Console.WriteLine(list.Count); // 3

list.**Clear**();

Console.WriteLine(list.Count); // 0

}

}

**=== Output of the program ===**

3

0

**Copying array to List**

Here we see an easy way to create a new List with the elements in an array that already exists. You can use the List constructor and pass it the array as the parameter. List receives this parameter, and fills its values from it.

**--- Program that copies array to List (C#) ---**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

int[] arr = new int[3]; // New array with 3 elements

arr[0] = 2;

arr[1] = 3;

arr[2] = 5;

List<int> list = **new List<int>(arr)**; // Copy to List

Console.WriteLine(list.Count); // 3 elements in List

}

}

**--- Output of the program ---**

(Indicates number of elements.)

3

**Notes on the example.** It is useful to use the List constructor code here to create a new List from Dictionary keys. This will give you a List of the Dictionary keys. The array element type must match the type of the List elements, or the compiler will refuse to compile your code.

**Finding elements**

Here we an example of how you can test each element in your List for a certain value. This shows the foreach loop, which tests to see if 3 is in the List of prime numbers. Note that more advanced List methods are available to find matches in the List, but they often aren't any better than this loop. They can sometimes result in shorter code.

[**See List Find Method.**](http://dotnetperls.com/list-find)

**~~~ Program that uses foreach on List (C#) ~~~**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// New list for example

List<int> primes = new List<int>(new int[] { 2, 3, 5 });

// See if List contains 3

**foreach** (int number in primes)

{

if (number == 3) // Will match once

{

Console.WriteLine("Contains 3");

}

}

}

}

**~~~ Output of the program ~~~**

Contains 3

**Using capacity**

You can use the Capacity property on List, or pass an integer into the constructor, to improve allocation performance when using List. The author's research shows that capacity can improve performance by nearly two times for adding elements. Note however that this is not usually a performance bottleneck in programs that access data.

[**See Capacity Property.**](http://dotnetperls.com/capacity)

**TrimExcess method.** There is the TrimExcess method on List as well, but its usage is very limited and I have never needed to use it. It reduces the memory used. Note: "The TrimExcess method does nothing if the list is at more than 90 percent of capacity".

[**Visit msdn.microsoft.com.**](http://msdn.microsoft.com/en-us/library/ms132207.aspx)

**Using BinarySearch**

You can use the binary search algorithm on List with the instance BinarySearch method. Binary search uses guesses to find the correct element much faster than linear searching. It is often much slower than Dictionary.

[**See BinarySearch List.**](http://dotnetperls.com/binarysearch-list)

**Using AddRange and InsertRange**

You can use AddRange and InsertRange to add or insert collections of elements into your existing List. This can make your code simpler. See an example of these methods on this site.

[**See List AddRange Use.**](http://dotnetperls.com/list-addrange)

**Using ForEach method**

Sometimes you may not want to write a regular foreach loop, which makes ForEach useful. This accepts an Action, which is a void delegate method. Be very cautious when you use Predicates and Actions, because they can decrease the readability of your code.

**Another useful method.** There is a TrueForAll method that accepts a Predicate. If the Predicate returns true for each element in your List, the TrueForAll method will return true also. Else, it will return false.

**Using Join—string List**

Here we see how you can use string.Join on a List of strings. This is useful when you need to turn several strings into one comma-delimited string. It requires the ToArray instance method on List. The biggest advantage of Join here is that no trailing comma is present on the resulting string, which would be present in a loop where each string is appended.

**=== Program that joins List (C#) ===**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// List of cities we need to join

List<string> cities = new List<string>();

cities.Add("New York");

cities.Add("Mumbai");

cities.Add("Berlin");

cities.Add("Istanbul");

// Join strings into one CSV line

string line = string.**Join**(",", cities.**ToArray()**);

Console.WriteLine(line);

}

}

**=== Output of the program ===**

New York,Mumbai,Berlin,Istanbul

**Getting List from Keys in Dictionary**

Here we see how you can use the List constructor to get a List of keys in your Dictionary collection. This gives you a simple way to iterate over Dictionary keys, or store them elsewhere. The Keys instance property accessor on Dictionary returns an enumerable collection of keys, which can be passed to the List constructor as a parameter.

**::: Program that converts Keys (C#) :::**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Populate example Dictionary

var dict = new Dictionary<int, bool>();

dict.Add(3, true);

dict.Add(5, false);

// Get a List of all the Keys

List<int> keys = new **List<int>(dict.Keys)**;

foreach (int key in keys)

{

Console.WriteLine(key);

}

}

}

**::: Output of the program :::**

3, 5

**Inserting elements**

Here we see how you can insert an element into your List at any position. The string "dalmation" is inserted into index 1, which makes it become the second element in the List. Note that if you have to Insert elements extensively, you should consider the Queue and LinkedList collections for better performance. Additionally, a Queue may provide clearer usage of the collection in your code.

**~~~ Program that inserts into List (C#) ~~~**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

List<string> dogs = new List<string>(); // Example List

dogs.Add("spaniel"); // Contains: spaniel

dogs.Add("beagle"); // Contains: spaniel, beagle

dogs.**Insert**(1, "dalmation"); // Contains: spaniel, dalmation, beagle

foreach (string dog in dogs) // Display for verification

{

Console.WriteLine(dog);

}

}

}

**~~~ Output of the program ~~~**

spaniel

dalmation

beagle

**Removing elements**

The removal methods on List are covered in depth in another article on this site. It contains examples for Remove, RemoveAt, RemoveAll, and RemoveRange, along with the author's notes.

[**See List Remove Methods.**](http://dotnetperls.com/list-remove)

**Sorting and reversing**

You can use the powerful Sort and Reverse methods in your List collection. These allow you to order your List in ascending or descending order. Additionally, you can use Reverse even when your List is not presorted. There is more information on these topics, as well as sorting your List with LINQ on a property on this site.

[**See Sort List Method, Sorting and Reversing Lists.**](http://dotnetperls.com/sort-list)

**Converting List to array**

You can convert your List to an array of the same type using the instance method ToArray. There are examples of this conversion, and the opposite, on this site.

[**See Convert List to Array.**](http://dotnetperls.com/convert-list-array)

[**See List CopyTo Method.**](http://dotnetperls.com/list-copyto)

**Getting range of elements**

Here we see how you can get a range of elements in your List collection using the GetRange instance method. This is similar to the Take and Skip methods from LINQ, but has different syntax.

**--- Program that gets ranges from List (C#) ---**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

List<string> rivers = new List<string>(new string[]

{

"nile",

"amazon", // River 2

"yangtze", // River 3

"mississippi",

"yellow"

});

// Get rivers 2 through 3

List<string> range = rivers.**GetRange**(1, 2);

foreach (string river in range)

{

Console.WriteLine(river);

}

}

}

**--- Output of the program ---**

amazon

yangtze

**Testing Lists for equality**

Sometimes you may need to test two Lists for equality, even when their elements are unordered. You can do this by sorting both of them and then comparing, or by using a custom List equality method. This site contains an example of a method that tests lists for equality in an unordered way.

[**See List Element Equality.**](http://dotnetperls.com/list-equals)

**Using List with structs**

When using List, you can improve performance and reduce memory usage with structs instead of classes. A List of structs is allocated in contiguous memory, unlike a List of classes. This is an advanced optimization. Note that in many cases using structs will actually decrease the performance when they are used as parameters in methods such as those on the List type.

**Using var keyword**

Here we see how you can use List collections with the var keyword. This can greatly shorten your lines of code, which sometimes improves readability. The var keyword has no effect on performance, only readability for programmers.

**~~~ Program that uses var with List (C#) ~~~**

using System.Collections.Generic;

class Program

{

static void Main()

{

**var list1** = new List<int>(); // <- var keyword used

List<int> list2 = new List<int>(); // <- Is equivalent to

}

}

**Summary**

In this tutorial, we saw lots of examples with the List constructed type. You will find that List is powerful and performs well. It provides flexible allocation and growth, making it much easier to use than arrays. In most programs that do not have memory or performance constraints and must add elements dynamically, the List constructed type in the C# programming language is ideal.

[**See System.Collections.Generic Namespace.**](http://dotnetperls.com/system-collections-generic)

[**See List Overview.**](http://dotnetperls.com/~list)

[**© 2007-2011 Sam Allen. All rights reserved.**](http://dotnetperls.com/c)