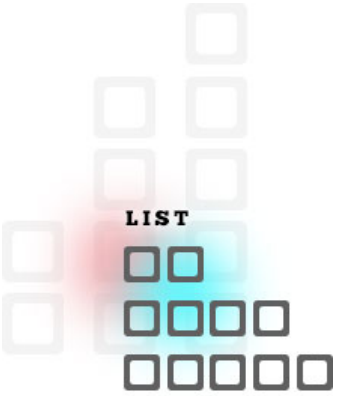
C# List

An array does not dynamically resize. A **List** does. With it, you do not need to manage the size on your own. This type is ideal for linear collections not accessed by keys. Dynamic in size, with many methods, List makes life easier.

Array



> **Note:**
> List is a generic (constructed) type. You need to use < and > in the List declaration. Lists handle any element type.

Add

First, we declare a List of int values. We create a new List of unspecified size and adds four prime numbers to it. Values are stored in the order added. There are other ways to create Lists—this is not the simplest.

> **Tip:**
> The angle brackets are part of the declaration type. They are not conditional (less or more than) operators.

► C# CSV
► C# Int
► Convert C# to VB Net

Based on:

```
.NET 4.5
```

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);
    }
}
```

The example adds a primitive type (int) to a List collection. But a List can also hold reference types and object instances. It has no special support for ints. Other types work just as well.

Add

> **Note:**
> For adding many elements at once—adding an array to a List—we use the AddRange method.
> **AddRange**

Loops

You can loop through your List with the for and foreach-loops. This is a common operation when using List. The syntax is the same as that for an array, except you use Count, not Length for the upper bound.

Tip:

You can loop backwards through your List. Start with list.Count - 1, and decrement 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

```
(Repeated twice)
2
3
7
```

Count, Clear

To get the number of elements in your List, access the Count property. This is fast to access—just avoid the Count extension method.

Count,

on the List type,

is equal to Length on arrays.

Here we use the Clear method, along with the Count property, to erase all the elements in a 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.

But:

After assigning to null, you must call the constructor again to avoid getting a NullReferenceException.

List Clear

Program that counts List: C#

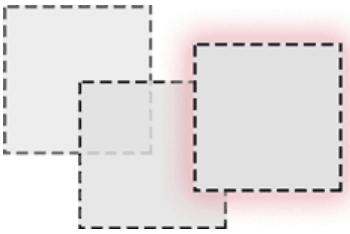
```
using System;
using System.Collections.Generic;

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

```
3
0
```

Copy array

Next we create a new List with the elements in an array that already exists. We use the List constructor and pass it the array. List receives this parameter and fills its values from it.



Caution:

The array element type must match the List element type or compilation will fail.

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

```
(Indicates number of elements.)

3
```

Find

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. More advanced List methods are also available to find matches in the List.

Note:
The Find method declaratively searches. We pass it a lambda expression. It can sometimes result in shorter code.
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

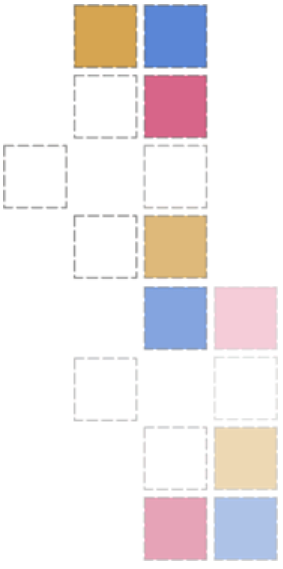
```
Contains 3
```

Several search methods are available on List. The Contains, Exists and IndexOf methods all provide searching. They vary in arguments and return values. With Predicate methods, we influence what elements match.

Note:
Contains returns only a bool. Exists receives a Predicate and returns a bool. IndexOf returns the position of the element found.
Contains
Exists
IndexOf

Capacity

You can use the Capacity property on List, or pass an integer into the constructor, to improve allocation performance. Capacity can improve performance by nearly two times for adding elements.
Capacity



However:
Adding elements, and resizing List, is not usually a performance bottleneck in programs that access data.

Also, there is the TrimExcess method on List. But its usage is limited. It reduces the memory used by lists with large capacities. And as MSDN

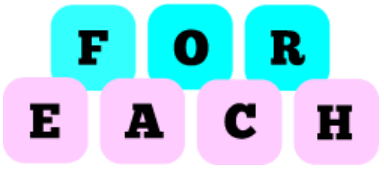
states, TrimExcess often does nothing.

> The TrimExcess method does nothing if the list is at more than 90 percent of capacity.
> **TrimExcess: MSDN**

BinarySearch

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

ForEach



Sometimes you may not want to write a regular foreach loop, which makes ForEach useful. It accepts an Action. Be cautious when you use Predicates and Actions. These objects can decrease the readability of code.

The TrueForAll method accepts a Predicate. If the Predicate returns true for each element in the List, the TrueForAll method will also return true. It checks the entire list—unless an element doesn't match and it returns false early.

Join string List

Next, we use string.Join on a List of strings. This is helpful when you need to turn several strings into one comma-delimited string. It requires the ToArray instance method on List. This ToArray is not an extension method.

> **Tip:**
> The biggest advantage of Join here is that no trailing comma is present on the resulting string.

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

```
New York,Mumbai,Berlin,Istanbul
```

Keys in Dictionary

We use the List constructor to get a List of keys from a Dictionary. This is a simple way to iterate over Dictionary keys or store them elsewhere. The Keys property returns an enumerable collection of keys.

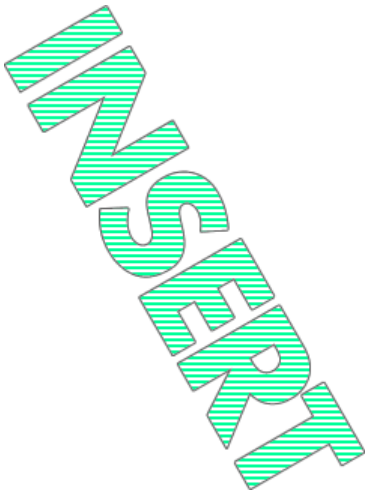
Program that converts Keys: C#

```
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

```
3, 5
```

Insert



You can insert an element into a List at any position. The string here is inserted into index 1. This makes it the second element. If you have to Insert often, please consider Queue and LinkedList.

> **Also:**
> A Queue may allow simpler usage of the collection in your code. Your intent may be clearer.
> **Queue**

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, "dalmatian"); // Contains: spaniel, dalmatian, beagle

        foreach (string dog in dogs) // Display for verification
        {
            Console.WriteLine(dog);
```

```
        }
    }
}
```

Output

```
spaniel
dalmatian
beagle
```

Also, Insert and InsertRange provide insertion on Lists. Please be aware that these can impact performance in a negative way—successive elements must be copied. Another collection type might be needed.

Insert
InsertRange

Remove

We present examples for Remove,
RemoveAt,
RemoveAll
and RemoveRange. In general Remove operates the same way as Insert. It too hinders performance. The removal methods on List are covered in depth.

Remove
RemoveAt
RemoveAll

Sort

Sort orders the elements in the List. For strings it orders alphabetically. For integers or other numbers it orders from lowest to highest. It acts upon elements depending on type. It is possible to provide a custom method.

Sort

Reverse

This next example program uses Reverse on a List. The strings contained in the List are left unchanged. But the order they appear in the List is inverted. No sorting occurs—the original order is intact but inverted.

Program that uses Reverse: C#

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

class Program
{
    static void Main()
    {
        List<string> list = new List<string>();
        list.Add("anchovy");
        list.Add("barracuda");
        list.Add("bass");
        list.Add("viperfish");

        // Reverse List in-place, no new variables required
```

```
        list.Reverse();

        foreach (string value in list)
        {
            Console.WriteLine(value);
        }
    }
}
```

Output

```
viperfish
bass
barracuda
anchovy
```

The List Reverse method, which internally uses the Array.Reverse method, provides an easy way to reverse the order of the elements in your List. It does not change the individual elements in any way.

Array.Reverse

Convert

Conversion of data types is a challenge. 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.

List to Array

CopyTo

Strings:

Some string methods can be used with the List type. We use, with List, the Concat and Join methods.

string.Concat

Join

GetRange

You can get a range of elements in your List using the GetRange method. This is similar to the Take and Skip methods. It has different syntax. The result List can be used like any other List.

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

```
amazon
yangtze
```

DataGridView

We can use the List type with a DataGridView. But sometimes it is better to convert the List to a DataTable. For a List of string arrays, this will make the DataGridView correctly display the elements.

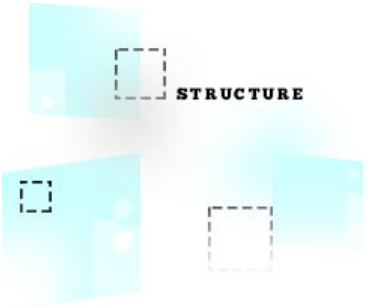
Convert List, DataTable

Equality

Sometimes we need to test two Lists for equality, even when their elements are unordered. We can do this by sorting both of them and then comparing, or by using a custom List equality method. Custom algorithms are possible.

> **Note:**
> This site contains an example of a method that tests lists for equality in an unordered way. A set collection might be better.
> **List Element Equality**

Structs



When using List, we 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 a complex, advanced optimization.

> **However:**
> Using structs will actually decrease performance when they are used as parameters in methods such as those on the List type.
> **Structs**

Var

We 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.

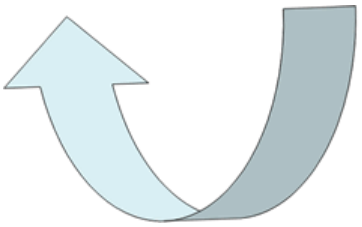
Var

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
    }
}
```

GetEnumerator



Programs are built upon many abstractions. With List even loops can be abstracted into an Enumerator. This means you can use the same methods to loop over a List or an array or certain other collections.

GetEnumerator

Initialize

There are several syntax forms we use to initialize a List collection. Most of them have equivalent performance. In fact most are compiled to the same intermediate language instructions.

Initialize List

Uses

Programs are complex. They have many contexts, many paths. The List type in particular is useful in many places. We use it with methods from System.Linq and System.IO. We manipulate it and serialize it.

List Concat

Remove Duplicates

Serialize List

Types:

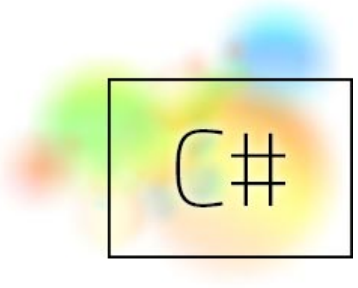
We test integer Lists, string Lists, static Lists, nested Lists and null Lists. They are used in similar ways.

Nested List

Null List

Static List

Summary



List is a constructed, parametric type. It is powerful and performs well.

It provides flexible allocation
and growth,
making it easier to use than arrays.

The syntax is at first confusing but you become used to it.

Therefore:

In most programs lacking strict memory or performance constraints, List is ideal.

► C# Example
► String C#
► C# Array