Lesson 12 - Lists in C# .NET

[C# .NET](https://www.ict.social/csharp) [OOP](https://www.ict.social/csharp/oop) Lists in C# .NET

In the previous lesson, [Date and time in C# .NET](https://www.ict.social/csharp/oop/date-and-time-in-csharp-net), we talked about a date and time in C# .NET. Today, we're going to take a look at a collection that is smarter than an array. Mainly, because it allows us to add or remove items from it at will.

We've already mentioned the term "collection". A collection is a structure where multiple objects can be stored. There are lots of collections in the .NET framework designed for different purposes, and can be used in different ways. So far, we only know one collection - arrays. However, for future lessons, we need something smarter into which we can simply **add and remove items** during run-time. It would surely be useful to be able to manage a database of objects in memory. We know that an array has a constant size which is the price to be paid for its high speed. Now, let's go over *Lists* **which can be seen as an extension of the array data type**.

**List**

The List type belongs to a group that we call **generic collections**. The term "genericity" will be fully explained in the collections course, yes, there is a full course dedicated to them. For now, just know that when we declare a List, we must specify the data type of the objects which will be stored in it. Let's keep it simple and make a List of numbers that we'll generate randomly in a lotto simulator application.

**The lottery**

The program will ask whether we want to generate another lot (number) and that will eventually be added to the List. If we won't want to lot anymore, the program will print the generated numbers in ascending order. We'll start by creating a new project, calling it *ListLottery*, and adding a *Lottery* class to it. The class will include a List of the int type where the numbers will be stored. The List will be private and serve only as an internal storage of the class, so it won't be accessible from the outside. Here's how we declare a list:

List<**int**> numbers;

**We specify the data type for generic collections in angle brackets**. A List is an object, just like everything else. Next, we'll initialize the variable before using it:

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

Notice the brackets that indicate a constructor. We'll place the list into our class, along with the random generator - Random. In the constructor, we'll initialize these fields:

**class** Lottery

{

**private** List<**int**> numbers;

**private** Random random;

**public** Lottery()

{

random = **new** Random();

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

}

}

We will also add *Lot()* and *Print()* methods, where *Lot()* will add a new random number to the list and return it and *Print()*will return a string containing all the generated numbers, sorted and separated by spaces.

We already know how to generate random numbers from the lesson with the RollingDie object. This time, we'll generate numbers from 1 to 100. The number will be added to the List using the **Add()** method:

**public** **int** Lot()

{

**int** number = random.Next(100) + 1;

numbers.Add(number);

**return** number;

}

Easy, right? The list collection is pretty complicated internally so I won't go into the "behind the scenes" details until later. Plus, one of the main reasons the .NET framework was made was to offer sophisticated components of high quality that are easy to use.

Printing the numbers will be even easier. To sort them, we'll use the **Sort()** method on the List. The Sort() method is similar to the one on the Array class. The method doesn't return anything, it just sorts our List inside.

**public** **string** Print()

{

**string** s = "";

numbers.Sort();

**foreach** (**int** i **in** numbers)

s += i + " ";

**return** s;

}

Done.

Let's move to *Main()* and make it so the program allows the user to control the object using a while loop. We've already done something like this before, where we ask the user whether he wishes to calculate another problem. We did it in the calculator sample program in one of our first couple of lessons. The same idea applies here.

We'll control the program by entering the following options - 1, 2, 3 (lot another number, print, quit). We'll read them as chars, not strings, using the Console.ReadKey() method. Don't forget we declare characters using apostrophes, not quotation marks.

Lottery lottery = **new** Lottery();

Console.WriteLine("Welcome to our lottery program.");

**char** choice = '0';

// main loop

**while** (choice != '3')

{

// option list

Console.WriteLine("1 - Lot the next number");

Console.WriteLine("2 - Print numbers");

Console.WriteLine("3 - Quit");

choice = Console.ReadKey().KeyChar;

Console.WriteLine();

// reaction to choice

**switch** (choice)

{

**case** '1':

Console.WriteLine("You got a: {0}", lottery.Lot());

**break**;

**case** '2':

Console.WriteLine("Numbers drawn: {0}", lottery.Print());

**break**;

**case** '3':

Console.WriteLine("Thanks for using our Lotto program");

**break**;

**default**:

Console.WriteLine("Invalid option. Please, try again.");

**break**;

}

}

The program flow is clear from the code. First of all, we set a default value so the loop could start. Then, we read an option from the keyboard as a char. The character is processed by a switch and the program performs the appropriate action. If the user enters an invalid option, the default case prints a message that tells them to enter a valid option.

Console application

...

1 - Lot the next number

2 - Print numbers

3 - Quit

1

You got a: 52

1 - Lot the next number

2 - Print numbers

3 - Quit

1

1 - Lot the next number

2 - Print numbers

3 - Quit

1

1 - Lot the next number

2 - Print numbers

3 - Quit

2

Numbers drawn: 10 12 13 14 21 22 23 24 28 28 42 45 52 52 57 58 59 70 71 72 79 83 86 89

1 - Lot the next number

2 - Print numbers

3 - Quit

Now we have visually confirmed that we are able to continue and add more and more new numbers to the list. The List collection can do pretty much everything an array can do and much more. However, if we want, we could work with it just like we would with an array.

We can index using brackets, but beware, there must be an item at that position in order for it to function properly:

List<**string**> l = **new** List<**string**>();

l.Add("First");

Console.WriteLine(l[0]);

l[0] = "First item";

Console.WriteLine(l[0]);

l[1] = "Second item"; // causes an error

We create a List of strings. We add the "first" item and then print an item at index 0, which will print "first". Of course, we can also write at that position; however, we can't work with the second item since we haven't added it to the List. When working with arrays, we have to specify its size and the array generates empty "boxes", indexed variables, for us. With Lists, we don't specify the size and we add "boxes" manually.

Let's look at the List collection in detail and go over some of the methods it provides that we will use in the near future:

**Constructors**

We are not limited to adding single objects to Lists. We could also create a copy of another List, array or any other type of collection. All we have to do is pass the collection to the constructor:

[Run code](https://www.ict.social/) Click to edit

**string**[] stringArray = {"First", "Second", "Third"};

List<**string**> l = **new** List<**string**>(stringArray);

Console.WriteLine(l[2]);

The code will print "Third" and array items will be copied into the new List. Similarly, we could pass another List in the constructor and get the same results.

**List properties**

* **Count** - Works as Length does on an array, it returns the number of items in the collection.

**List methods**

* **Add(item)** - This method takes an item as a parameter and adds to the end of the list.
* **AddRange(collec­tion)** - Adds multiple items to the list, e.g. from an array.
* **Clear()** - Removes all items in the List.
* **Contains(item)** - Returns true/false depending on whether the List contains the given item.
* **CopyTo(array)** - Copies all items in the list to a given array. We can specify the starting index and the number of items.
* **IndexOf(item)** - Returns the index of the first occurrence of a given item in the list (like with an array). Returns -1 in case of failure.
* **Insert(index, item)** - Inserts an item at the given index, position, in the List.
* **InsertRange(index, collection)** - Inserts multiple items from the collection at the given index in the List.
* **LastIndexOf(item)** - Returns the index of the last occurrence of a given item in the list. Returns -1 in case of failure.
* **Remove(item)** - Removes the first occurrence of an item.
* **RemoveAt(index)** - Removes the item at the specified index.
* **RemoveRange(index, count)** - Removes a specified number of items starting at a specific index.
* **Reverse()** - Works the same as it would in an array. It reverses the List so the first item is the last and vice versa. The method doesn't return anything, the changes are made directly to the List.
* **Sort()** - This method sorts the items in the List. The method doesn't return anything either.
* **ToArray()** - Copies the items from the List to a given array and returns it.

**Other methods**

List also provides all of the methods that are present in arrays:

* **Average()** - Returns the average of the items in the List as a double.
* **Distinct()** - Returns all of the items of which there is only one instance in the List.
* **First()** - Returns the first item.
* **Last()** - Returns the last item.
* **Intersect(collec­tion)** - Returns the intersection of the List with the specified collection.
* **Union()** - Returns the union of the List with the specified collection.
* **Min** - Returns the smallest element.
* **Max** - Returns the largest element.
* **Sum** - Returns the sum of the elements.
* **Take(number)** - Returns a specified number of elements starting at the beginning.

Now we have seen that the List collection can do much more than arrays. The biggest advantage to it is being able to add and remove items. Drawbacks in the performance are insignificant to us now. In the collections course, we'll cover all of the other methods that List provides, but for now, the ones we covered today will suffice.

A number storing program is surely interesting, but it would be more useful to be able to store objects instead of numbers. In the next lesson, [Diary with a database in C# .NET](https://www.ict.social/csharp/oop/diary-with-a-database-in-csharp-net), we'll use a List to create a database, specifically, an electronic diary! :)

Download

[Download ListLottery.zip](https://www.ict.social/api/Articles-Article/file/570e51336559c)

Downloaded 17x (38.63 kB)   
 Application includes source codes in language C#

[Mark this lesson as absolved](https://www.ict.social/)

Article has been written for you by [David Capka](https://www.ict.social/portfolio/1)

[](https://www.ict.social/portfolio/1)

Do you like this article?

Top of Form

Bottom of Form

2 votes

The author is a programmer, who likes web technologies and being the lead/chief article writer at ICT.social. He shares his knowledge with the community and is always looking to improve. He believes that anyone can do what they set their mind to.



<https://www.ict.social/csharp/oop/lists-in-csharp-net>