

What is List?

The list can be defined as an abstract data type in which the elements are stored in an ordered manner for easier and efficient retrieval of the elements. List Data Structure allows repetition that means a single piece of data can occur more than once in a list. In the case of multiple entries of the same data, each entry of that repeating data is considered as a distinct item or entry. It is very much similar to the array but the major difference between the array and the list data structure is that array stores only homogenous data in them whereas the list (in some programming languages) can store heterogeneous data items in its object. List Data Structure is also known as a sequence.

The list can be called Dynamic size arrays, which means their size increased as we go on adding data in them and we need not to pre-define a static size for the list.

Implementation of List in C#

The **Collection classes** are a group of classes designed specifically for grouping together *objects* and performing *tasks* on them.

List class is a collection and defined in the It's the replacement for **arrays, linked lists, queues**, and most other one-dimensional data structures.

This is because it has all kinds of extra functionality, including the ability to **grow** in size on-demand.

List<T> class Characteristics:

- It is different from the arrays. A **List<T> can be resized dynamically** but arrays cannot.
- List<T> class can accept null as a valid value for reference types and it also allows duplicate elements.
- If the Count becomes equals to Capacity, then the capacity of the List increased automatically by reallocating the internal array. The existing elements will be copied to the new array before the addition of the new element.
- List<T> class is the generic equivalent of ArrayList class by implementing the IList<T> generic interface.
- This class can use both equality and ordering comparer.
- List<T> class is not sorted by default and elements are accessed by zero-based index.

For very large List<T> objects, you can increase the **maximum capacity to 2 billion elements** on a 64-bit system by setting the enabled attribute of the configuration element to true in the run- time environment

The **List<T> class** implements **eight** different interfaces that provide different

functionalities. Hence, the List<T> object can be assigned to any of its interface type variables. However, it is recommended to create an object of List<T> and assign it to IList<T> or List<T> type variable, as shown below.

Example: List<T>

```
List<int> intList = new List<int>();
```

//Or

```
IList<int> intList = new List<int>();
```

In the above example, the first statement uses List<T> type variable, whereas the second statement uses IList<T> type variable. The List<T> is a concrete implementation of IList<T> interface. In the object-oriented programming, it is advisable to program to **interface** rather than **concrete class**. So use **IList<T>** type variable to create an object of List<T>.

However, List<T> includes more helper methods than IList<T> interface. The following table lists the important properties and methods of List<T> class:

Table 3. List Property

Property	Usage
Items	Gets or sets the element at the specified index
Count	Returns the total number of elements exists in the List<T>

Table 4. List Method

Methods	Usage
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the specified element exists or not in a List<T>.
Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
InsertRange	Inserts elements of another collection at the specified index.
Remove	Removes the first occurrence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match with the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.
TrueForAll	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

Example Program 5 Adding elements into a list

```

IList<int> intList = new List<int>();
intList.Add(10);
intList.Add(20);
intList.Add(30);
intList.Add(40);

IList<string> strList = new List<string>();
strList.Add("one");
strList.Add("two");
strList.Add("three");
strList.Add("four");
strList.Add("four");
strList.Add(null);
strList.Add(null);

IList<Student> studentList = new List<Student>();
studentList.Add(new Student());
studentList.Add(new Student());
studentList.Add(new Student());

```

You can also add elements at the time of initialization using object initializer syntax as below:

Example Adding elements using object initializer syntax

```

IList<int> intList = new List<int>() { 10, 20, 30, 40 };
//Or
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID=1, StudentName="Bill"},
    new Student() { StudentID=2, StudentName="Steve"},
    new Student() { StudentID=3, StudentName="Ram"},
    new Student() { StudentID=1, StudentName="Moin"}
};

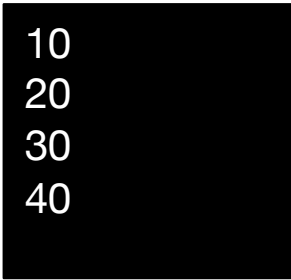
```

Example using AddRange()

Use List.AddRange() method adds all the elements from another collection.
AddRange() signature: void AddRange(IEnumerable<T> collection)

Example using AddRange

```
IList<int> intList1 = new List<int>();  
intList1.Add(10);  
intList1.Add(20);  
intList1.Add(30);  
intList1.Add(40);  
List<int> intList2 = new List<int>();  
intList2.AddRange(intList1);
```



10
20
30
40