# C# Collections

In C#, collection represents group of objects. By the help of collections, we can perform various operations on objects such as

* store object
* update object
* delete object
* retrieve object
* search object, and
* sort object

## **Types of Collections in C#**

There are 3 ways to work with collections. The three namespaces are given below:

* **System.Collections.Generic** classes
* **System.Collections** classes (Now deprecated)
* **System.Collections.Concurrent** classes

### **System.Collections.Generic classes**

It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries. These collections are type-safe because they are generic means only those items that are type-compatible with the type of the collection can be stored in a generic collection, it eliminates accidental type mismatches. Generic collections are defined by the set of interfaces and classes.

The System.Collections.Generic namespace has following classes:

* List
* Stack
* Queue
* LinkedList
* HashSet
* SortedSet
* Dictionary
* SortedDictionary
* SortedList

|  |  |
| --- | --- |
| **Class name** | **Description** |
| **Dictionary<TKey,TValue>** | It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class. |
| [List<T>](https://www.geeksforgeeks.org/c-list-class/) | It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class. |
| **Queue<T>** | A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class. |
| **SortedList<TKey,TValue>** | It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class. |
| **Stack<T>** | It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class. |
| [HashSet<T>](https://www.geeksforgeeks.org/c-sharp-hashset-class/) | It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection. |
| [LinkedList<T>](https://www.geeksforgeeks.org/c-sharp-linkedlist-class/) | It allows fast inserting and removing of elements. It implements a classic linked list. |

### **2) System.Collections classes**

These classes are legacy. It is suggested now to use System.Collections.Generic classes. The System.Collections namespace has following classes:

* ArrayList
* Stack
* Queue
* Hashtable

### **3) System.Collections.Concurrent classes**

The System.Collections.Concurrent namespace provides classes for thread-safe operations. Now multiple threads will not create problem for accessing the collection items.

The System.Collections.Concurrent namespace has following classes:

* BlockingCollection
* ConcurrentBag
* ConcurrentStack
* ConcurrentQueue
* ConcurrentDictionary
* Partitioner
* Partitioner
* OrderablePartitioner

# C# List<T>

C# List<T> class is used to store and fetch elements and provides the methods to search, sort, and manipulate lists. **It can have duplicate elements**. It is found in **System.Collections.Generic** namespace. It represents the list of objects which can be accessed by index.

**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.

| **Constructor** | **Description** |
| --- | --- |
| **List<T>()** | Initializes a new instance of the List<T> class that is empty and has the default initial capacity. |
| **List<T>(IEnumerable<T>)** | Initializes a new instance of the List<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied. |
| **List<T>(Int32)** | Initializes a new instance of the List<T> class that is empty and has the specified initial capacity. |

| **Property** | **Description** |
| --- | --- |
| [Capacity](https://www.geeksforgeeks.org/c-capacity-of-a-list/) | Gets or sets the total number of elements the internal data structure can hold without resizing. |
| [Count](https://www.geeksforgeeks.org/c-count-the-total-number-of-elements-in-the-list/) | Gets the number of elements contained in the List<T>. |
| [Item[Int32]](https://www.geeksforgeeks.org/c-gets-or-sets-the-element-at-the-specified-index-in-the-list/) | Gets or sets the element at the specified index. |

| **Method** | Description |
| --- | --- |
| [**Add(T)**](https://www.geeksforgeeks.org/c-adding-an-element-to-the-list/) | Adds an object to the end of the List<T>. |
| [**AddRange(IEnumerable<T>)**](https://www.geeksforgeeks.org/c-adding-the-elements-of-the-specified-collection-to-the-end-of-the-list/) | Adds the elements of the specified collection to the end of the List<T>. |
| [**AsReadOnly()**](https://www.geeksforgeeks.org/c-creating-a-read-only-wrapper-for-listt/) | Returns a read-only ReadOnlyCollection<T> wrapper for the current collection. |
| [**BinarySearch()**](https://www.geeksforgeeks.org/list-binarysearch-method-in-c-sharp/) | Uses a binary search algorithm to locate a specific element in the sorted List<T> or a portion of it. |
| [**Clear()**](https://www.geeksforgeeks.org/c-removing-all-the-elements-from-the-list/) | Removes all elements from the List<T>. |
| [**Contains(T)**](https://www.geeksforgeeks.org/c-how-to-check-whether-a-list-contains-a-specified-element/) | Determines whether an element is in the List<T>. |
| **ConvertAll(Converter)** | Converts the elements in the current List<T> to another type, and returns a list containing the converted elements. |
| **CopyTo()** | Copies the List<T> or a portion of it to an array.  Exceptions:-  [ArgumentNullException](https://learn.microsoft.com/en-us/dotnet/api/system.argumentnullexception?view=net-7.0)  array is null.  [ArgumentOutOfRangeException](https://learn.microsoft.com/en-us/dotnet/api/system.argumentoutofrangeexception?view=net-7.0)  arrayIndex is less than 0.  [ArgumentException](https://learn.microsoft.com/en-us/dotnet/api/system.argumentexception?view=net-7.0)  The number of elements in the source [List<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-7.0) is greater than the available space from arrayIndex to the end of the destination array.  This method uses [Array.Copy](https://learn.microsoft.com/en-us/dotnet/api/system.array.copy?view=net-7.0) to copy the elements.  The elements are copied to the [Array](https://learn.microsoft.com/en-us/dotnet/api/system.array?view=net-7.0) in the same order in which the enumerator iterates through the [List<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-7.0).  This method is an O(n) operation, where n is [Count](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.count?view=net-7.0). |
| **ToArray()** | Copies the elements of the List<T> to a new array. |
| **ToString()** | Returns a string that represents the current object. |
| [**Equals(Object)**](https://www.geeksforgeeks.org/c-check-if-two-listt-objects-are-equal/) | Determines whether the specified object is equal to the current object. |
| [**Exists(Predicate<T>)**](https://www.geeksforgeeks.org/c-how-to-check-whether-a-list-contains-the-elements-that-match-the-specified-conditions/) | Determines whether the List<T> contains elements that match the conditions defined by the specified predicate. |
| [**Find(Predicate<T>)**](https://www.geeksforgeeks.org/c-sharp-first-occurrence-in-the-list-that-matches-the-specified-conditions/) | Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>. |
| [**FindLast(Predicate<T>)**](https://www.geeksforgeeks.org/c-sharp-how-to-get-the-last-occurrence-of-the-element-in-the-list-that-match-the-specified-conditions/) | Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List<T>. |
| [**FindAll(Predicate<T>)**](https://www.geeksforgeeks.org/c-how-to-get-all-elements-of-a-list-that-match-the-conditions-specified-by-the-predicate/) | Retrieves all the elements that match the conditions defined by the specified predicate. |
| [**FindIndex()**](https://www.geeksforgeeks.org/list-findindex-method-in-c-sharp-with-examples/) | Searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the first occurrence within the List<T> or a portion of it. This method returns -1 if an item that matches the conditions is not found. |
| [**FindLastIndex()**](https://www.geeksforgeeks.org/list-findlastindex-method-in-c-sharp-set-1/) | Searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the last occurrence within the List<T> or a portion of it. |
| **IndexOf()** | Returns the zero-based index of the first occurrence of a value in the List<T> or in a portion of it. |
| **LastIndexOf()** | Returns the zero-based index of the last occurrence of a value in the List<T> or in a portion of it. |
| [**ForEach(Action<T>)**](https://www.geeksforgeeks.org/c-how-to-perform-a-specified-action-on-each-element-of-the-list/) | Performs the specified action on each element of the List<T>. |
| [**GetEnumerator()**](https://www.geeksforgeeks.org/c-get-an-enumerator-that-iterates-through-the-list/) | Returns an enumerator that iterates through the List<T>. |
| **GetHashCode()** | Serves as the default hash function. |
| **GetRange(Int32, Int32)** | Creates a shallow copy of a range of elements in the source List<T>. |
| **GetType()** | Gets the Type of the current instance. |
| **Insert(Int32, T)** | Inserts an element into the List<T> at the specified index. |
| [**InsertRange(Int32, IEnumerable<T>)**](https://www.geeksforgeeks.org/c-how-to-insert-the-elements-of-a-collection-into-the-list-at-the-specified-index/) | Inserts the elements of a collection into the List<T> at the specified index. |
| **MemberwiseClone()** | Creates a shallow copy of the current Object. |
| [**Remove(T)**](https://www.geeksforgeeks.org/c-removing-the-specified-element-from-the-list/) | Removes the first occurrence of a specific object from the List<T>. |
| [**RemoveAll(Predicate<T>)**](https://www.geeksforgeeks.org/c-remove-all-elements-of-a-list-that-match-the-conditions-defined-by-the-predicate/) | Removes all the elements that match the conditions defined by the specified predicate. |
| [**RemoveAt(Int32)**](https://www.geeksforgeeks.org/c-how-to-remove-the-element-from-the-specified-index-of-the-list/) | Removes the element at the specified index of the List<T>. |
| [**RemoveRange(Int32, Int32)**](https://www.geeksforgeeks.org/c-removing-a-range-of-elements-from-the-list/) | Removes a range of elements from the List<T>. |
| [**Reverse()**](https://www.geeksforgeeks.org/c-sharp-reverse-the-order-of-the-elements-in-the-entire-list-or-in-the-specified-range/) | Reverses the order of the elements in the List<T> or a portion of it. |
| [**Sort()**](https://www.geeksforgeeks.org/how-to-sort-list-in-c-sharp-set-1/) | Sorts the elements or a portion of the elements in the List<T> using either the specified or default IComparer<T> implementation or a provided Comparison<T> delegate to compare list elements. |
| [**TrimExcess()**](https://www.geeksforgeeks.org/c-list-trimexcess-method/) | Sets the capacity to the actual number of elements in the List<T>, if that number is less than a threshold value. |
| [**TrueForAll(Predicate<T>)**](https://www.geeksforgeeks.org/c-check-if-every-list-element-matches-the-predicate-conditions/) | Determines whether every element in the List<T> matches the conditions defined by the specified predicate. |

# C# | HashSet Class

A **HashSet<T>** is an **unordered collection** of the **unique elements**. It comes under ***System.Collections.Generic*** namespace. It is used in a situation where we want to prevent duplicates from being inserted in the collection. As far as performance is concerned, it is better in comparison to the list.

**Characteristics of HashSet Class:**

* The HashSet<**T**> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.
* The capacity of a HashSet<**T**> object is the number of elements that the object can hold.
* A HashSet<**T**> object’s capacity automatically increases as elements are added to the object.
* A HashSet<**T**> collection is not sorted and cannot contain duplicate elements.
* HashSet<**T**> provides many mathematical set operations, such as set addition (unions) and set subtraction.