**What is Collections in C#?**

Collections are similar to Arrays, it provides a more flexible way of working with a group of objects. In arrays, you would have noticed that you need to define the number of elements in an array beforehand. This had to be done when the array was declared.

But in a collection, you don't need to define the size of the collection beforehand. You can add elements or even remove elements from the collection at any point of time. This chapter will focus on how we can work with the different collections available in C#.

|  |  |
| --- | --- |
| [ArrayList](https://www.guru99.com/c-sharp-arraylist) | The ArrayList collection is similar to the Arrays data type in C#. The biggest difference is the dynamic nature of the array list collection. |
| [Stack](https://www.guru99.com/c-sharp-stack.html) | The stack is a special case collection which represents a last in first out (LIFO) concept |
| [Queues](https://www.guru99.com/c-sharp-queue.html) | The Queue is a special case collection which represents a first in first out concept |
| [Hashtable](https://www.guru99.com/c-sharp-hashtable.html) | A hash table is a special collection that is used to store key-value items |
| SortedList | The SortedList is a collection which stores key-value pairs in the ascending order of key by default. |
| BitArray | A bit array is an array of data structure which stores bits |

## What is ArrayList in C#?

The ArrayList collection is similar to the Arrays data type in C#. The biggest difference is the dynamic nature of the array list collection.

For arrays, you need to define the number of elements that the array can hold at the time of array declaration. But in the case of the Array List collection, this does not need to be done beforehand. Elements can be added or removed from the Array List collection at any point in time. Let's look at the operations available for the array list collection in more detail.

### Declaration of an Array List

The declaration of an ArrayList is provided below. An array list is created with the help of the ArrayList Datatype. The "new" keyword is used to create an object of an ArrayList. The object is then assigned to the variable a1. So now the variable a1 will be used to access the different elements of the array list.

ArrayList a1 = new ArrayList()

### Adding elements to an array

The add method is used to add an element to the ArrayList. The add method can be used to add any sort of data type element to the array list. So you can add an Integer, or a string, or even a Boolean value to the array list. The general syntax of the addition method is given below

ArrayList.add(element)

Below are some examples of how the "add" method can be used. The add method can be used to add various data types to the Array List collection.

Below you can see examples of how we can add Integer's Strings and even Boolean values to the Array List collection.

* a1.add(1) – This will add an Integer value to the collection
* a1.add("Example") – This will add a String value to the collection
* a1.add(true) – This will add a Boolean value to the collection
* using System;
* using System.Collections;
* using System.Collections.Generic;
* using System.Linq;
* using System.Text;
* using System.Threading.Tasks;
* namespace DemoApplication
* {
* class Program
* {
* static void Main(string[] args)
* {
* ArrayList a1 = new ArrayList();
* a1.Add(1);
* a1.Add("Example");
* a1.Add(true);
* Console.WriteLine(a1.Count);
* Console.WriteLine(a1.Contains(2));
* Console.WriteLine(a1[1]);
* a1.RemoveAt(1);
* Console.WriteLine(a1[1]);
* Console.ReadKey();
* }
* }
* }

1. The first step is used to declare our Array List. Here we are declaring a1 as a variable to hold the elements of our array list.
2. We then use the add keyword to add the number 1 , the String "Example" and the Boolean value 'true' to the array list.
3. We then use the Console.WriteLine method to display the value of each array lists element to the console. You will notice that just like arrays, we can access the elements via their index positions. So to access the first position of the Array List, we use the [0] index position. And so on and so forth.

**Count**

This method is used to get the number of items in the ArrayList collection. Below is the general syntax of this statement.

ArrayList.Count() – This method will return the number of elements that the array list contains.

**Contains**

This method is used to see if an element is present in the ArrayList collection. Below is the general syntax of this statement

ArrayList.Contains(element) – This method will return true if the element is present in the list, else it will return false.

**RemoveAt**

This method is used to remove an element at a specific position in the ArrayList collection. Below is the general syntax of this statement

ArrayList.RemoveAt(index) – This method will remove an element from a specific position of the Array List.

sing System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

ArrayList al = new ArrayList();

al.Add(1);

al.Add("Example");

al.Add(true);

Console.WriteLine(a1.Count);

Console.WriteLine(a1.Contains(2));

Console.WriteLine(a1[1]);

a1.RemoveAt(1);

Console.WriteLine(a1[1]);

Console.ReadKey();

}

}

}

## What is Stack in C#?

The stack is a special case collection which represents a last in first out (LIFO) concept. To first understand LIFO, let's take an example. Imagine a stack of books with each book kept on top of each other.

The concept of last in first out in the case of books means that only the top most book can be removed from the stack of books. It is not possible to remove a book from between, because then that would disturb the setting of the stack.

Hence in C#, the stack also works in the same way. Elements are added to the stack, one on the top of each other. The process of adding an element to the stack is called a push operation. To remove an element from a stack, you can also remove the top most element of the stack. This operation is known as pop.

Let's look at the operations available for the Stack collection in more detail.

### Declaration of the stack

A stack is created with the help of the Stack Data type. The keyword "new" is used to create an object of a Stack. The object is then assigned to the variable st.

Stack st = new Stack()

**Adding elements to the stack**

The push method is used to add an element onto the stack. The general syntax of the statement is given below.

Stack.push(element)

**Removing elements from the stack**

The pop method is used to remove an element from the stack. The pop operation will return the topmost element of the stack. The general syntax of the statement is given below

Stack.pop()

**Count**

This property is used to get the number of items in the Stack. Below is the general syntax of this statement.

Stack.Count

**Contains**

This method is used to see if an element is present in the Stack. Below is the general syntax of this statement. The statement will return true if the element exists, else it will return the value false.

Stack.Contains(element)

Now let's see this working at a code level. All of the below-mentioned code will be written to our Console application. The code will be written to our Program.cs file.

In the below program, we will write the code to see how we can use the above-mentioned methods.

**Example 1**

In this example, we will see

* How a stack gets created.
* How to display the elements of the stack, and use the Count and Contain methods.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Stack st = new Stack();

st.Push(1);

st.Push(2);

st.Push(3);

foreach (Object obj in st)

{

Console.WriteLine(obj);

}

Console.WriteLine(); Console.WriteLine();

Console.WriteLine("The number of elements in the stack " +st.Count);

Console.WriteLine("Does the stack contain the elements 3 "+st.Contains(3));

Console.ReadKey();

}

}

}

1. The first step is used to declare the Stack. Here we are declaring "st" as a variable to hold the elements of our stack.
2. Next, we add 3 elements to our stack. Each element is added via the Push method.
3. Now since the stack elements cannot be accessed via the index position like the array list, we need to use a different approach to display the elements of the stack. The Object (obj) is a temporary variable, which is declared for holding each element of the stack. We then use the foreach statement to go through each element of the stack. For each stack element, the value is assigned to the obj variable. We then use the Console.Writeline command to display the value to the console.
4. We are using the Count property (**st.count**) to get the number of items in the stack. This property will return a number. We then display this value to the console.
5. We then use the Contains method to see if the value of 3 is present in our stack. This will return either a true or false value. We then display this return value to the console.

Now let's look at the "remove" functionality. We will see the code required to remove the topmost element from the stack.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Stack st = new Stack();

st.Push(1);

st.Push(2);

st.Push(3);

st.Pop();

foreach (Object obj in st)

{

Console.WriteLine(obj);

}

Console.ReadKey();

}

}

}

1. Stack stores the values in LIFO (Last in First out) style. The element which is added last will be the element to come out first.
2. Use the Push() method to add elements into Stack.
3. The Pop() method returns and removes elements from the top of the Stack. Calling the Pop() method on the empty Stack will throw an exception.
4. The Peek() method always returns top most element in the Stack

## What is Queue in C#?

The Queue is a special case collection which represents a first in first out concept. Imagine a queue of people waiting for the bus. Normally, the first person who enters the queue will be the first person to enter the bus. Similarly, the last person to enter the queue will be the last person to enter into the bus. Elements are added to the stack, one on the top of each other.

The process of adding an element to the queue is the enqueuer operation. To remove an element from a queue, you can use the dequeuer operation. The operation in queues is similar to stack we saw previously.

Let's look at the operations available for the Queue collection in more detail.

### Declaration of the Queue

The declaration of a Queue is provided below. A Queue is created with the help of the Queue Data type. The "new" keyword is used to create an object of a Queue. The object is then assigned to the variable qt.

Queue qt = new Queue()

### Adding elements to the Queue

The enqueue method is used to add an element onto the queue. The general syntax of the statement is given below.

Queue.enqueue(element)

### Removing elements from the queue

The dequeue method is used to remove an element from the queue. The dequeue operation will return the last element of the queue. The general syntax of the statement is given below

Queue.dequeue()

### Count

This property is used to get the number of items in the queue. Below is the general syntax of this statement.

Queue.Count

### Contains

This method is used to see if an element is present in the Queue. Below is the general syntax of this statement. The statement will return true if the element exists, else it will return the value false.

Queue.Contains(element)

Now, let's see this working at a code level. All of the below-mentioned code will be written to our Console application.

The code will be written to our Program.cs file. In the below program, we will write the code to see how we can use the above-mentioned methods.

### Example

In this example, we will see how a queue gets created. Next, we will see how to display the elements of the queue, and use the Count and Contain methods.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Queue qt = new Queue();

qt.Enqueue(1);

qt.Enqueue(2);

qt.Enqueue(3);

foreach (Object obj in qt)

{

Console.WriteLine(obj);

}

Console.WriteLine(); Console.WriteLine();

Console.WriteLine("The number of elements in the Queue " + qt.Count);

Console.WriteLine("Dose the Queue contain " + qt.Contains(3));

Console.ReadKey();

}

}

}

**Code Explanation**

1. The first step is used to declare the Queue. Here we are declaring qt as a variable to hold the elements of our Queue.
2. Next, we add 3 elements to our Queue. Each element is added via the "enqueue" method.
3. Now one thing that needs to be noted about Queues is that the elements cannot be accessed via the index position like the array list. We need to use a different approach to display the elements of the Queue. So here's how we go about displaying the elements of a queue.

* We first declare a temporary variable called obj. This will be used to hold each element of the Queue.
* We then use the foreach statement to go through each element of the Queue.
* For each Queue element, the value is assigned to the obj variable.
* We then use the Console.Writeline command to display the value to the console.

1. We are using the "Count" property to get the number of items in the Queue. This property will return a number. We then display this value to the console.
2. We then use the "Contains" method to see if the value of 3 is present in our Queue. This will return either a true or false value. We then display this return value to the console.

DEQUE

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Queue qt = new Queue();

qt.Enqueue(1);

qt.Enqueue(2);

qt.Enqueue(3);

qt.Dequeue();

foreach (Object obj in qt)

{

Console.WriteLine(obj);

}

Console.ReadKey();

}

}

}

Iterate Queue

Queue queue = new Queue();

queue.Enqueue(3);

queue.Enqueue(2);

queue.Enqueue(1);

queue.Enqueue("Four");

Console.WriteLine("Number of elements in Queue: {0}", queue.Count);

foreach (var i in queue.ToArray())

Console.WriteLine(i);

Console.WriteLine("Number of elements in Queue: {0}", queue.Count);

## Peek()

The Peek() method always returns the first item from a queue collection without removing it from the queue. Calling Peek() and Dequeue() methods on an empty queue collection will throw a run time exception "InvalidOperationException".

Peek() Method Signature: *object Peek();*

Example: Peek()

Queue queue = new Queue();

queue.Enqueue(3);

queue.Enqueue(2);

queue.Enqueue(1);

queue.Enqueue("Four");

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

Console.WriteLine(queue.Peek());

Console.WriteLine(queue.Peek());

Console.WriteLine(queue.Peek());

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

## Clear()

The Clear() method removes all the items from a queue.

Clear() Signature:*void Clear();*

Example: Clear()

Queue queue = new Queue();

queue.Enqueue(3);

queue.Enqueue(2);

queue.Enqueue(1);

queue.Enqueue("Four");

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

queue.Clear();

Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);

1. The Queue stores the values in FIFO (First in First out) style. The element which is added first will come out First.
2. Use the Enqueue() method to add elements into Queue
3. The Dequeue() method returns and removes elements from the beginning of the Queue. Calling the Dequeue() method on an empty queue will throw an exception.
4. The Peek() method always returns top most element.

## What is Hashtable in C#?

A hash table is a special collection that is used to store key-value items. So instead of storing just one value like the stack, array list and queue, the hash table stores 2 values. These 2 values form an element of the hash table.

Below are some example of how values of a hash table might look like.

{ "001" , ".Net" }

{ "002" , ".C#" }

{ "003" , "ASP.Net" }

Above we have 3 key value pairs. The keys of each element are 001, 002 and 003 respectively. The values of each key value pair are ".Net", "C#" and "ASP.Net" respectively.

Let's look at the operations available for the Hashtable collection in more detail.

### Declaration of the Hashtable

The declaration of a Hashtable is shown below. A Hashtable is created with the help of the Hashtable Datatype. The "new" keyword is used to create an object of a Hashtable. The object is then assigned to the variable ht.

Hashtable ht = new Hashtable()

### Adding elements to the Hashtable

The Add method is used to add an element on to the queue. The general syntax of the statement is given below

HashTable.add("key","value")

### Example 1:

Remember that each element of the hash table comprises of 2 values, one is the key, and the other is the value.

Now, let's see this working at a code level. All of the below-mentioned code will be written to our Console application.

The code will be written to our Program.cs file. In the below program, we will write the code to see how we can use the above-mentioned methods.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Hashtable ht = new Hashtable();

ht.Add("001",".Net");

ht.Add("002","C#");

ht.Add("003","ASP.Net");

ICollection keys = ht.Keys;

foreach (String k in keys)

{

Console.WriteLine(ht[k]);

}

Console.ReadKey();

}

}

}

**Code Explanation:-**

1. First, we declare the hashtable variable using the Hashtable data type by using keyword "New." The name of the variable defines is 'ht'.
2. We then add elements to the hash table using the Add method. Remember that we need to add both a key and value element when adding something to the hashtable.
3. There is no direct way to display the elements of a hash table.

* In order to display the hashtable , we first need to get the list of keys (001, 002 and 003) from the hash table.
* This is done via the ICollection interface. This is a special data type which can be used to store the keys of a hashtable collections. We then assign the keys of the hashtable collection to the variable 'keys'.

1. Next for each key value, we get the associated value in the hashtable by using the statement ht[k].

how we can use the "Containskey" and "ContainsValue" method.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

class Program

{

static void Main(string[] args)

{

Hashtable ht = new Hashtable();

ht.Add("001",".Net");

ht.Add("002","C#");

ht.Add("003","ASP.Net");

Console.WriteLine(ht.ContainsKey("001"));

Console.WriteLine(ht.ContainsValue("C#"));

Console.ReadKey();

}

}

}

1. First, we use the ContainsKey method to see if the key is present in the hashtable. This method will return true if the key is present in the hashtable. This method should return true since the key does exist in the hashtable.
2. We then use the ContainsValue method to see if the value is present in the hashtable. This method will return 'true' since the Value does exist in the hashtable.

Iterate Hashtable

Hashtable ht = new Hashtable()

{

{ 1, "One" },

{ 2, "Two" },

{ 3, "Three" },

{ 4, "Four" },

{ 5, null },

{ "Fv", "Five" },

{ 8.5F, 8.5 }

};

foreach (DictionaryEntry item in ht)

Console.WriteLine("key:{0}, value:{1}",item.Key, item.Value);

Access Hashtable using Keys & Values

Hashtable ht = new Hashtable();

{

{ 1, "One" },

{ 2, "Two" },

{ 3, "Three" },

{ 4, "Four" },

{ 5, null },

{ "Fv", "Five" },

{ 8.5F, 8.5 }

};

foreach (var key in ht.Keys )

Console.WriteLine("Key:{0}, Value:{1}",key , ht[key]);

Console.WriteLine("\*\*\*All Values\*\*\*");

foreach (var value in ht.Values)

Console.WriteLine("Value:{0}", value);

## Remove elements in Hashtable

The Remove() method removes the item with the specified key from the hashtable.

Remove() Method Signature: *void Remove(object key)*

Example: Remove()

Hashtable ht = new Hashtable()

{

{ 1, "One" },

{ 2, "Two" },

{ 3, "Three" },

{ 4, "Four" },

{ 5, null },

{ "Fv", "Five" },

{ 8.5F, 8.5 }

};

ht.Remove("Fv"); // removes {"Fv", "Five"}

## Check for Existing Elements

Contains() and ContainsKey() check whether the specified key exists in the Hashtable collection. ContainsValue() checks whether the specified value exists in the Hashtable.

Contains(), ContainsKey() and ContainsValue() Signatures:

bool Contains(object key)

bool ContainsKey(object key)

bool ContainsValue(object value)

Example: Contains

Hashtable ht = new Hashtable()

{

{ 1, "One" },

{ 2, "Two" },

{ 3, "Three" },

{ 4, "Four" }

};

ht.Contains(2);// returns true

ht.ContainsKey(2);// returns true

ht.Contains(5); //returns false

ht.ContainsValue("One"); // returns true

Clear()

Hashtable ht = new Hashtable()

{

{ 1, "One" },

{ 2, "Two" },

{ 3, "Three" },

{ 4, "Four" },

{ 5, null },

{ "Fv", "Five" },

{ 8.5F, 8.5 }

};

ht.Clear(); // removes all elements

Console.WriteLine("Total Elements: {0}", ht.Count);

1. Hashtable stores key-value pairs of any datatype where the Key must be unique.
2. The Hashtable key cannot be null whereas the value can be null.
3. Hashtable retrieves an item by comparing the hashcode of keys. So it is slower in performance than Dictionary collection.
4. Hashtable uses the default hashcode provider which is object.GetHashCode(). You can also use a custom hashcode provider.
5. Use DictionaryEntry with foreach statement to iterate Hashtable.

# How to create a SortedList

SortedList class is a collection of **(key, value) pairs** which are sorted according to keys. Those pairs can be accessible by key and as well as by index(zero-based indexing). This comes under **System.Collections**namespace.

**Properties of SortedList:**

* Internally the object of SortedList maintains two arrays. The first array is used to store the elements of the list i.e. keys and the second one is used to store the associated values.
* A key cannot be null but value can be.
* As SortedList used sorting which makes it slower in comparison to Hashtable.
* The capacity of a SortedList can be dynamically increased through reallocation.
* The keys in the SortedList cannot be duplicated but values can be.
* The SortedList can be sorted according to the keys using the IComparer(Either in ascending or descending order).

## Add elements in SortedList

Use the Add() method to add key-value pairs into a SortedList.

Add() method signature: *void Add(object key, object value)*

Key cannot be null but value can be null. Also, datatype of all keys must be same, so that it can compare otherwise it will throw runtime exception.

Example: Add key-value pairs in SortedList

SortedList sortedList1 = new SortedList();

sortedList1.Add(3, "Three");

sortedList1.Add(4, "Four");

sortedList1.Add(1, "One");

sortedList1.Add(5, "Five");

sortedList1.Add(2, "Two");

SortedList sortedList2 = new SortedList();

sortedList2.Add("one", 1);

sortedList2.Add("two", 2);

sortedList2.Add("three", 3);

sortedList2.Add("four", 4);

SortedList sortedList3 = new SortedList();

sortedList3.Add(1.5, 100);

sortedList3.Add(3.5, 200);

sortedList3.Add(2.4, 300);

sortedList3.Add(2.3, null);

sortedList3.Add(1.1, null);

SortedList key can be of any data type, but you cannot add keys of different data types in the same SortedList. The key type of the first key-value pair remains the same for all other key-value pairs. The following example will throw run time exception because we are trying to add the second item with a string key:

Example: Key of different datatypes throws exception:

SortedList sortedList = new SortedList();

sortedList.Add(3, "Three");

sortedList.Add("Four", "Four"); // Throw exception: InvalidOperationException

sortedList.Add(1, "One");

sortedList.Add(8, "Five");

sortedList.Add(2, "Two");

Access SortedList

SortedList sortedList = new SortedList()

{

{"one", 1},

{"two", 2},

{"three", 3},

{"four", "Four"}

}

int i = (int) sortedList["one"];

int j = (int) sortedList["two"];

string str = (string) sortedList["four"];

Console.WriteLine(i);

Console.WriteLine(j);

Console.WriteLine(str);

Accessing Values using For Loop

SortedList sortedList = new SortedList()

{

{3, "Three"},

{4, "Four"},

{1, "One"},

{5, "Five"},

{2, "Two"}

};

for (int i = 0; i < sortedList.Count; i++)

{

Console.WriteLine("key: {0}, value: {1}",

sortedList.GetKey(i), sortedList.GetByIndex(i));

}

ccess values using foreach

SortedList sortedList = new SortedList()

{

{3, "Three"},

{4, "Four"},

{1, "One"},

{5, "Five"},

{2, "Two"}

};

foreach(DictionaryEntry kvp in sortedList )

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

## Remove elements from SortedList

Use the Remove() or RemoveAt() method to remove elements from a SortedList.

Remove() signature: *void Remove(object key)*

RemoveAt() signature: *void RemoveAt(int index)*

Example: Remove elements in SortedList

SortedList sortedList = new SortedList();

sortedList.Add("one", 1);

sortedList.Add("two", 2);

sortedList.Add("three", 3);

sortedList.Add("four", 4);

sortedList.Remove("one");//removes element whose key is 'one'

sortedList.RemoveAt(0);//removes element at zero index i.e first element: four

foreach(DictionaryEntry kvp in sortedList )

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

## heck for an existing key in SortedList

The Contains() & ContainsKey() methods determine whether the specified key exists in the SortedList collection or not.

Contains() signature: *bool Contains(object key)*

ContainsKey() signature: *bool ContainsKey(object key)*

The ContainsValue() method determines whether the specified value exists in the SortedList or not.

ContainValue() signature: *bool ContainValue(object value)*

Example: Contains

SortedList sortedList = new SortedList();

{

{3, "Three"},

{4, "Four"},

{1, "One"},

{8, "Eight"},

{2, "Two"}

};

sortedList.Contains(2); // returns true

sortedList.Contains(4); // returns true

sortedList.Contains(6); // returns false

sortedList.ContainsKey(2); // returns true

sortedList.ContainsKey(6); // returns false

sortedList.ContainsValue("One"); // returns true

sortedList.ContainsValue("Ten"); // returns false

* SortedList stores the key-value pairs in ascending order of the key. Key must be unique and cannot be null whereas value can be null or duplicate.
* Non-generic SortedList stores keys and values of any data types. So values needs to be cast to appropriate data type.
* Key-value pair can be cast to DictionaryEntry.
* Access individual value using indexer. SortedList indexer accepts key to return value associated with it.