C# Collections

In C# Collections are classes that provide an easy way to work with a group of objects.

Types of Collections

- Generic Collection
- Non Generic Collection

Generic Collection

The System.Collections.Generic classes help us to create a generic collection. In this, we store type compatible data elements. This collection does not allow us to store different types of elements. Internally Generic collections store the elements in arrays of respective types.

In C#, following are the classes that come under System.Collections.Generic namespace:

1. List Class
2. Stack Class
3. Queue Class
4. SortedList Class

List<T> class

The List<T> class is used to store multiple elements of the same data type that can be accessed using the indexes. We can add, insert and remove elements inside the list. Moreover, we can dynamically change the size of the list.

Create a List

- To create List<T> in C#, we need to use the System.Collections.Generic namespace.
- Here is how we can create List<T>.For example,

```
// list containing integer values
List<int> items = new List<int>(){10,43,23,56};

// list containing string values
List<string> names = new List<string>(){"John","Wilson","Anish"};


// list containing double values
List<double> data = new List<double>(){3.5,4.5,7.5};
```

Access the List Elements

We can access List using index notation []. For example,

```
using System;
using System.Collections.Generic;
class Program
```

```
{
    static void Main(string[] args)
       {
           List<int> items = new List<int>(){10,42,67,89,90};
           //to access 10
           Console.WriteLine(items[0]);
           //to access 67
           Console.WriteLine(items[2]);
           Console.ReadKey();
       }
}
```

Output:
10
67

Since the index of the list starts from 0:

- items[0] - accesses the first element
- items[2] - accesses the third element

Iterate the List

In C#, we can also loop through each element of List<T> using a for loop.

For example: using for loop with index

```
using System;
using System.Collections.Generic;
class Program
 {
    static void Main(string[] args)
       {
           List<int> items = new List<int>(){10,42,67,89,90};
           for(int i = 0; i < items.Count; i++)
           {
              Console.WriteLine(items[i]);
           }
           Console.ReadKey();
       }
}
```

Note: The Count property returns the total number of elements inside the list.

For example: using foreach loop

```
using System;
using System.Collections.Generic;
class Program
 {
    static void Main(string[] args)
```

```
        {
            List<int> items = new List<int>(){10,42,67,89,90};
            foreach(int i in items)
            {
               Console.WriteLine(i);
            }
            Console.ReadKey();
        }
 }
```

Output:

10
42
67
89
90

Basic Operations on List

Add Elements:

- To add a single element to the List, we use the Add() method of the List<T> class.
- For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            List<int> items = new List<int>();
            items.Add(11);
            items.Add(32);
            items.Add(34);
            items.Add(67);
            foreach(int i in items)
            {
               Console.WriteLine(i);
            }
            Console.ReadKey();
        }
 }
```

Output:

11
32
34
67

Insert Element in a List
To insert an element to a specified index in List, we use the Insert(index, data) method of the List<T> class.

For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            List<int> items = new List<int>();
            items.Add(11);
            items.Add(32);
            items.Add(34);
            items.Add(67);
            items.Insert(2, 567);
            foreach(int i in items)
            {
               Console.WriteLine(i);
            }
            Console.ReadKey();
        }
 }
```

Output:
11
32
567
34
67

In the above example,

items.Insert(2, 567) inserts 567 at the 2nd index position

Remove Elements from the List
We can delete one or more items from List<T> using two methods:

- Remove(value) - removes the first occurrence of an element from the given list
- RemoveAt(index) - removes the elements at the specified position in the list

Example: Remove() Method

```
using System;
using System.Collections.Generic;
class Program
```

```
{
    static void Main(string[] args)
       {
          List<string> cars = new
                          List<string>(){"BMW","Tesla","Honda","Fiat"};

           //to delete "Tesla"
           cars.Remove("Tesla");

           foreach(string i in cars)
           {
              Console.WriteLine(i);
           }
           Console.ReadKey();
       }
}
```

Output:
BMW
Honda
Fiat

Example: RemoveAt() Method

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            List<string> cars = new
                          List<string>(){"BMW","Tesla","Honda","Fiat"};

            //to delete "Honda"
            cars.RemoveAt(2);

            foreach(string i in cars)
            {
               Console.WriteLine(i);
            }
            Console.ReadKey();
        }
 }
```

Output:
BMW
Tesla
Fiat

Changing Elements in List

The List<T> element can be changed by using the index position.

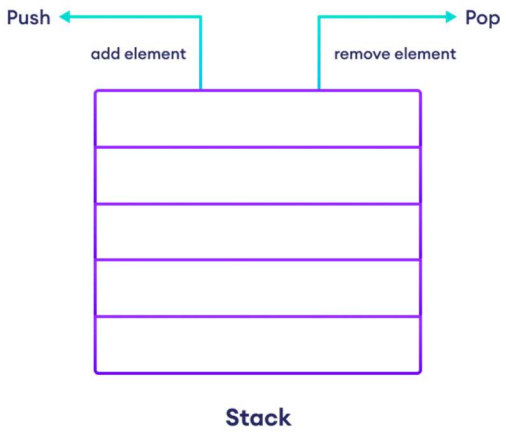
```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            List<string> cars = new
                           List<string>(){"BMW","Tesla","Honda","Fiat"};

            //to replace "Tesla" as "Ford"
            cars[1] = "Ford";

            foreach(string i in cars)
            {
               Console.WriteLine(i);
            }
            Console.ReadKey();
        }
 }
```

Output:
BMW
Ford
Fiat

Stack<T> Class

The Stack<T> class is also generic, which means we store data elements of the same data type. In stack, the elements are stored in LIFO(Last In First Out) manner. With the help of methods, we can perform operations in stack as given below:

- Push() - insert elements
- Pop() - remove elements



Creating a Stack<T> collection:

To create Stack<T> in C#, we need to use the System.Collection.Generic namespace.

Here is how we can create Stack<T> in C#,

```
Stack<dataType> stackName = new Stack<dataType>();
```

For example,

```
Stack<int>  items = new Stack<int>();

Stack<string> names = new Stack<string>();
```

C# Stack Methods

C# provides 3 major Stack<T> methods. These methods are:

- Push() - adds element to the top of the stack
- Pop() - removes and returns an element from the top of the stack
- Peek() - returns an element from the top of the stack without removing

Stack Push() Method

To add an element to the top of the stack, we use the Push() method. For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
           Stack<string> colors= new Stack<string>();
           colors.Push("red");
           colors.Push("yellow");
```

```
            colors.Push("blue");
            colors.Push("green");

             // print elements inside the colors Stack
             foreach (string c in colors) {
                Console.WriteLine(c);
             }
             Console.ReadKey();
        }
 }
```

Output:
green
blue
yellow
red

Note: In above program, when we do iteration on Stack<string> it generates data from backwards due to LIFO(Last In First Out) data structure.

Stack Pop() Method

To remove an element from the top of the stack, we use the Pop() method. For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            Stack<string> colors= new Stack<string>();
            colors.Push("red");
            colors.Push("yellow");
            colors.Push("blue");
            colors.Push("green");

            Console.WriteLine(colors.Pop());

            Console.ReadKey();
        }
 }
```

Output:
green

Stack Peek() Method

The Peek() method returns the object at the top of the stack without removing it. For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
           Stack<string> colors= new Stack<string>();
           colors.Push("red");
           colors.Push("yellow");
           colors.Push("blue");
           colors.Push("green");

           Console.WriteLine(colors.Peek());

           Console.ReadKey();
        }
 }
```

Output:
green

How to check whether an element is present inside a stack?

We can use the Contains() method to check whether an element is present inside the stack or not. The method returns True if a specified element exists in the stack. For example,

```
Stack<string> colors= new Stack<string>();
colors.Push("red");
colors.Push("yellow");
colors.Push("blue");
colors.Push("green");

Console.WriteLine(colors.Contains("yellow"));
```

Output:
true

How to remove all the elements from Stack?

C# provides the Clear() method using which we can remove all the elements from the stack.

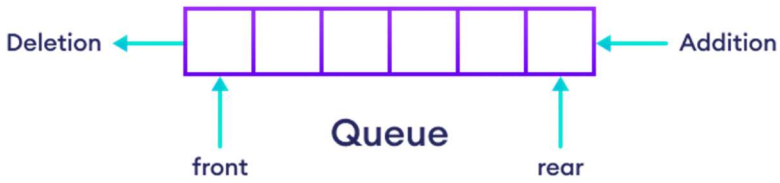
```
Stack<string> colors= new Stack<string>();
colors.Push("red");
colors.Push("yellow");
colors.Push("blue");
colors.Push("green");
```

```
colors.Clear();     //delete all elements
```

Queue<T>

A Queue<T> is a generic class that arranges elements of a specified data type using First In First Out (FIFO) principles. Here, the elements are inserted at one end and removed from the other. It is implemented using a circular queue. We can perform operations using methods like:

- Enqueue()- add elements
- Dequeue() - remove elements



Create a Queue in C#

To create Queue<T> in C#, we need to use the System.Collection.Generic namespace.

```
Queue<dataType> queueName = new Queue<dataType>();
```

Here, dataType indicates the queue's type. For example,

// create integer type stack

```
Queue<int> queue1 = new Queue<int>();
```

// create string type stack

```
Queue<string> queue2 = new Queue<string>();
```

C# Queue Methods

C# provides three major Queue<T> methods. These methods are:

- Enqueue() - adds an element to the end of the queue
- Dequeue() - removes and returns an element from the beginning of the queue
- Peek() - returns an element from the beginning of the queue without removing

Enqueue() Method

To add an element to the end of the queue, we use the Enqueue() method. For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            Queue<int> numbers = new Queue<int>();
            numbers.Enqueue(10);
            numbers.Enqueue(81);
            numbers.Enqueue(67);
            numbers.Enqueue(90);

            foreach (int item in numbers)
            {
                Console.WriteLine(item);
            }
            Console.ReadKey();
        }
 }
```

Output:

10

81

67

90

Note: Since the queue follows FIFO principle, the element added at the first (10) is displayed at the first in the output.

Dequeue() Method

To remove an element from the beginning of the queue, we use the Dequeue() method. For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
```

```
            Queue<int> numbers = new Queue<int>();
            numbers.Enqueue(10);
            numbers.Enqueue(81);
            numbers.Enqueue(67);
            numbers.Enqueue(90);

            int data = numbers.Dequeue();
            Console.WriteLine("Removed Data:" + data);

            Console.ReadKey();
        }
 }
```

Output:

Removed Data:10

Peek() Method

The Peek() method returns the element from the beginning of the queue without removing it.

For example,

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
            Queue<int> numbers = new Queue<int>();
            numbers.Enqueue(10);
            numbers.Enqueue(81);
            numbers.Enqueue(67);
            numbers.Enqueue(90);

            // returns element from the beginning of the planet queue
            Console.WriteLine("Data: " + numbers.Peek());

        Console.ReadKey();
        }
 }
```

Output:

Data: 10

Non Generic Collection

ArrayList class

In C#, the System.Collections classes help us to create a non-generic collection. For this we use System.Collections namespace. Using this we can create classes where we can add data elements of multiple data types.

ArrayList Class

- ArrayList is non-generic which means we can store elements of multiple data types.
- We use the ArrayList class to implement the functionality of resizable arrays.
- Duplicate elements are allowed inside ArrayList.
- We can use the sort method to sort the elements inside it.

Create an ArrayList

To create **ArrayList** in C#, we need to use the System.Collections namespace. Here is how we can create an arraylist in C#.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          Console.ReadKey();
        }
 }
```

Basic Operations on ArrayList

In C#, we can perform different operations on **ArrayList**. We will look at some commonly used **ArrayList** operations as given below:

- Add Elements
- Access Elements
- Insert Elements
- Change Elements
- Remove Elements

Add Elements

C# provides a method Add() using which we can add elements in ArrayList. For example,

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);
          mylist.Add(10000);
          Console.ReadKey();
        }
 }
```

Note: **ArrayList** stores elements with **different data types also**.

Add Elements in an ArrayList using Object Initializer Syntax

Object Initializer allows us to assign values at the time of creating an object.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList(){10,"program",5.6,10000};
          Console.ReadKey();
        }
 }
```

In the above example, we have created an **ArrayList** named mylist and assigned values at the same time using curly brackets.

This is how we use object initializer syntax.

Access ArrayList Elements

We use indexes to access elements in ArrayList. The indexing starts from 0. For example,

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
```

```
            mylist.Add(4.5);
            mylist.Add(10000);

            //to access first element
            Console.WriteLine(mylist[0]);

           //to access third element
            Console.WriteLine(mylist[2]);

            Console.ReadKey();
          }
 }
```

Output:

```
John
4.5
```

Iterate ArrayList

In C#, we can also loop through each element of ArrayList using a for loop.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);
          mylist.Add(10000);

          for(int i=0; i < mylist.Count; i++)
          {
            Console.WriteLine(mylist[i]);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
John
10
4.5
10000
```

Insert Elements in ArrayList

Add Elements in an ArrayList at specified index using Insert(index, data) method

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);

          //Insert 789 at Index 1 (Second Element)
          mylist.Insert(1, 789);

          for(int i=0;i<mylist.Count;i++)
          {
            Console.WriteLine(mylist[i]);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
John
789
10
4.5
```

Change ArrayList Elements

We can change the value of elements in ArrayList using index:

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);

          //replace 10 with 90.45
          mylist[1] = 90.45;

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

```
          {
            Console.WriteLine(mylist[i]);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
John
90.45
4.5
```

Remove ArrayList Elements

C# provides methods like Remove(), RemoveAt(), RemoveRange() to remove elements from ArrayList.

Example: Remove() method – used to delete an element directly.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);

          //delete 10
          mylist.Remove(10);

          for(int i=0;i<mylist.Count;i++)
          {
            Console.WriteLine(mylist[i]);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
John
4.5
```

Example: RemoveAt() method – used to delete an element using index position.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);

          //delete 4.5 using index
          mylist.RemoveAt(2);

          for(int i=0;i<mylist.Count;i++)
          {
            Console.WriteLine(mylist[i]);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
John
10
```

Example: RemoveRange(index, count) method – used to delete an element using index position.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          ArrayList mylist = new ArrayList();
          mylist.Add("John");
          mylist.Add(10);
          mylist.Add(4.5);
          mylist.Add(1000);
          mylist.Add("David");

          //delete 10,4.5,1000
          mylist.RemoveRange(1,3);

          for(int i=0;i<mylist.Count;i++)
          {
            Console.WriteLine(mylist[i]);
          }
```

```
            Console.ReadKey();
        }
 }
```

Output:

```
John
David
```

SortedList

A **SortedList** can be used as generic or non-generic collection that contains **key/value** pairs where keys are sorted in an order if types are compatible. For example,

Create an SortedList

To create SortedList in C#, we need to use the **System.Collections** namespace. Here is how we can create **SortedList**:

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          Console.ReadKey();
        }
 }
```

Add Elements in SortedList

C# provides a method Add(key,data) using which we can add elements in SortedList. For example,

Example 1: string keys

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add("4", 10);
          list.Add("8", 20);
          list.Add("2", 50);
          Console.ReadKey();
        }
 }
```

In the above SortedList, the elements will be sorted based on the keys as given below

First Item: 2 – 50

Second Item: 4 – 10

Third Item: 8 - 20

Example 2: integer keys

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
          list.Add(11, "cse");
          Console.ReadKey();
        }
 }
```

In the above SortedList, the elements will be sorted based on the keys as given below

1-50

5-10

9-arun

11-cse

Access the SortedList

We can access the elements inside the SortedList using it's keys with square bracket.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
```

```
            list.Add(11, "cse");

            //to access 50
            Console.WriteLine(list[1]);
            //to access "cse"
            Console.WriteLine(list[11]);

            Console.ReadKey();
        }
 }
```

Output:

50

cse

In the above example, we have accessed the elements using their keys:

list[11] - accesses the element whose key is 11

list[1] - accesses the element whose key is 1

Note: While accessing, if we pass the key which does not exist, the compiler throws an error.

Iterate through SortedList

In C#, we can also loop through each element of SortedList using a for loop with help of the following methods

- **GetKey(index)** – returns the key value of the specified element using index in the SortedList
- **GetByIndex(index)** – returns the value of the specified element using index in the SortedList.

Example:

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
          list.Add(11, "cse");
          for (int i = 0; i < list.Count; i++)
          {
            Console.WriteLine("{0} : {1}", list.GetKey(i), list.GetByIndex(i));
          }
          Console.ReadKey();
        }
```

```
 }
```

Output:

```
1 : 50
5 : 10
9 : arun
11 : cse
```

Replace values in SortedList

We can replace the elements inside the SortedList using it's keys with square bracket.

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
          list.Add(11, "cse");

          //replace value 50 as 100 at key 1
          list[1] = 100;

          for (int i = 0; i < list.Count; i++)
          {
            Console.WriteLine("{0}:{1}", list.GetKey(i), list.GetByIndex(i));
          }
          Console.ReadKey();
        }
 }
```

Output:

```
1 : 100
5 : 10
9 : arun
11 : cse
```

Remove SortedList Elements

We can delete one or more items from SortedList using 2 methods:

- Remove(key) - removes the element according to the specified key

- RemoveAt(index) - removes the element according to the specified index

Example: Remove(key) - removes the element according to the specified key

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
          list.Add(11, "cse");

          //remove "arun" at key 9
          list.Remove(9);

          for (int i = 0; i < list.Count; i++)
          {
            Console.WriteLine("{0}:{1}", list.GetKey(i), list.GetByIndex(i));
          }
          Console.ReadKey();
        }
 }
```

Output:

```
1 : 100
5 : 10
11 : cse
```

Example: RemoveAt(index) - removes the element according to the specified index

```
using System;
using System.Collections;
class Program
 {
     static void Main(string[] args)
        {
          SortedList list = new SortedList();
          list.Add(5, 10);
          list.Add(9, "arun");
          list.Add(1, 50);
          list.Add(11, "cse");

          //remove "arun" at using index position 1
          list.RemoveAt(1);
```

```
          for (int i = 0; i < list.Count; i++)
          {
            Console.WriteLine("{0}:{1}", list.GetKey(i), list.GetByIndex(i));
          }
          Console.ReadKey();
        }
 }
```

Output:

```
1 : 100
5 : 10
11 : cse
```

Creating Generic SortedList

The generic SortedList is defined in System.Collections.Generic namespace. We need to include the System.Collections.Generic namespace for creating SortedList with specified Key_Type and Value_Type.

For example: To create SortedList with integer keys and string data

```
using System;
using System.Collections.Generic;
class Program
 {
     static void Main(string[] args)
        {
          SortedList<int,string> list = new SortedList<int,string>();
          list.Add(5, "apple");
          list.Add(9, "arun");
          list.Add(1, "mango");
          list.Add(11, "cse");
          foreach(KeyValuePair<int,string> temp in list)
          {
            Console.WriteLine("{0} : {1}", temp.Key, temp.Value);
          }
          Console.ReadKey();
        }
 }
```

Output:

```
1 : 100
9 : arun
5 : 10
11 : cse
```

Note:

The KeyValuePair class stores a pair of values (key and data) in a single list with C#. In the above example, the foreach loop generate every element for SortedList and store the Key and Data in KeyValuePair List. Later, the Key and Data can be accessed by variables such as temp.Key and temp.Value

Other operations such as Replace and Remove are same as Non-Generic SortedList

---