

UNIT-5.2 Collection Classes

Introduction –

The Collections in C# are a set of predefined classes that are present in the System.Collections namespace that provides greater capabilities and functionalities than the traditional arrays.

A Collection in C# is a dynamic array.

That means the collections in C# have the capability of storing multiple values but with the following features.

- *Adding and inserting items to a collection*
- *Removing items from a collection*
- *Finding, sorting, searching items*
- *Replacing items*
- *Copy and clone collections and items*
- *Capacity and Count properties to find the capacity of the collection and number of items in the collection*

.NET supports two types of collections Classes,

1) Non - Generic collections [System.Collections classes]

The Non-Generic collection classes in C# operate on objects, and hence can handle any type of data, but not in a safe-type manner.

2) Generic collections [System.Collections.Generic classes]

Generic collections with work generic data type.

It provides a generic implementation of standard data structures like linked lists, stacks, queues, and dictionaries. These collection classes 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.

Non-generic	Generic
<i>ArrayList</i> ----->	<i>List</i>
<i>HashTable</i> ----->	<i>Dictionary</i>
<i>Stack</i> ----->	<i>Stack</i>
<i>Queue</i> ----->	<i>Queue</i>

1) Non - Generic collections [System.Collections classes]

In non-generic collections, each element can represent a value of a different type.

The collection size is not fixed.

Items from the collection can be added or removed at runtime.

1.1) C# ArrayList

ArrayList class is a collection that can be used for any types or objects.

1. *ArrayList is a class that is similar to an array, but it can be used to store values of different types.*
2. *An ArrayList doesn't have a specific size.*
3. *Any number of elements can be stored.*

Commonly used Methods

Method	Description
<u>Add(Object)</u>	<i>Adds an object to the end of the ArrayList.</i>
<u>Clear()</u>	<i>Removes all elements from the ArrayList.</i>
<u>Clone()</u>	<i>Creates a shallow copy of the ArrayList.</i>
<u>Contains(Object)</u>	<i>Determines whether an element is in the ArrayList.</i>
<u>CopyTo(Array)</u>	<i>Copies the entire ArrayList to a compatible one-dimensional Array, starting at the beginning of the target array.</i>
<u>CopyTo(Array, Int32)</u>	<i>Copies the entire ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array.</i>
<u>Equals(Object)</u>	<i>Determines whether the specified object is equal to the current object.</i>

<u>Insert(Int32, Object)</u>	<i>Inserts an element into the ArrayList at the specified index.</i>
<u>Remove(Object)</u>	<i>Removes the first occurrence of a specific object from the ArrayList.</i>
<u>RemoveAt(Int32)</u>	<i>Removes the element at the specified index of the ArrayList.</i>
<u>RemoveRange(Int32, Int32)</u>	<i>Removes a range of elements from the ArrayList.</i>
<u>Repeat(Object, Int32)</u>	<i>Returns an ArrayList whose elements are copies of the specified value.</i>
<u>Reverse()</u>	<i>Reverses the order of the elements in the entire ArrayList.</i>
<u>Reverse(Int32, Int32)</u>	<i>Reverses the order of the elements in the specified range.</i>
<u>Sort()</u>	<i>Sorts the elements in the entire ArrayList.</i>

Properties in the ArrayList

S.No	Property Name	Description
1	Capacity	This property is used to set or get the size to the ArrayList.
2	Count	It returns the total number of elements in the ArrayList.
3	IsFixedSize	It returns the Whether the ArrayList is fixed size or not. It returns the Boolean value.
4	IsReadOnly	It returns the Whether the ArrayList is Read-only or not. It returns the Boolean value

```
using System.Collections;
namespace ArrayListDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList al = new ArrayList();//Create

            al.Add(101);//Insert
            al.Add("SAM");//Insert
            al.Add(10.2f);//Insert
            al.Add("aadi");//Insert

            Console.WriteLine("Items In ArrayList are....."); //display
            foreach (object o in al)
                Console.WriteLine(o);

            ArrayList dl =(ArrayList) al.Clone();//Copy
            Console.WriteLine("Items In Destination ArrayList are.....");
            foreach (object o in dl)
                Console.WriteLine(o);

            al.Remove("aadi");//Remove
            Console.WriteLine("Items In ArrayList after removal one item are.....");
            foreach (object o in al)
                Console.WriteLine(o);

            al.Clear();//Clears all items in Arraylist
            Console.WriteLine("Total Number Of items after clear()..." + al.Count);

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

OUTPUT -

```

Items In ArrayList are.....
101
SAM
10.2
aaadi
Items In Destination ArrayList are.....
101
SAM
10.2
aaadi
Items In ArrayList after removal one item are.....
101
SAM
10.2
Total Number Of items after clear()...0

```

1.2) C# HashTable

*The **Hashtable** class represents a collection of key-and-value pairs that are organized based on the hash code of the key.*

It uses the key to access the elements in the collection.

*A hash table is used when you need to access elements by using **key**, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.*

The following table lists some of the commonly used properties of the Hashtable class –

Sr.No.	Property & Description
1	Count Gets the number of key-and-value pairs contained in the Hashtable.
2	IsFixedSize Gets a value indicating whether the Hashtable has a fixed size.

3	IsReadOnly Gets a value indicating whether the Hashtable is read-only.
4	Item Gets or sets the value associated with the specified key.
5	Keys Gets an ICollection containing the keys in the Hashtable.
6	Values Gets an ICollection containing the values in the Hashtable.

The following table lists some of the commonly used **methods** of the **Hashtable** class –

Sr.No.	Method & Description
1	public virtual void Add(object key, object value); Adds an element with the specified key and value into the Hashtable.
2	public virtual void Clear(); Removes all elements from the Hashtable.
3	public virtual bool ContainsKey(object key); Determines whether the Hashtable contains a specific key.
4	public virtual bool ContainsValue(object value); Determines whether the Hashtable contains a specific value.
5	public virtual void Remove(object key); Removes the element with the specified key from the Hashtable.

Example-**namespace** CollectionsDemo

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable ht = new Hashtable();
            ht.Add(101, "RAM");
            ht.Add(102, "SHAM");
            ht.Add(103, "RAJ");

            Console.WriteLine("ITEMS IN HASHTABLE .....");
            foreach(object t in ht.Keys)
            {
                Console.WriteLine(t);
            }

            foreach(DictionaryEntry dt in ht)
            { Console.WriteLine("RollNum=" + dt.Key + "----->" + "NAME="
+ dt.Value);
            }

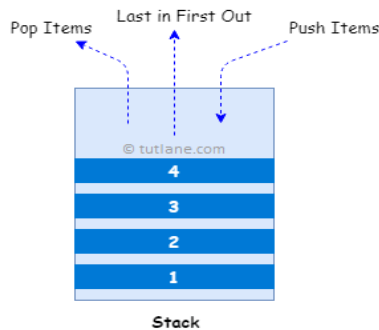
            ht.Remove(101);

            foreach (DictionaryEntry dt in ht)
            {
                Console.WriteLine("RollNum=" + dt.Key + "----->" + "NAME=" +
dt.Value);
            }

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

1.3) C# Stack

In c#, **Stack** is useful for representing a collection of objects that store elements in **LIFO** (Last in, First out) style, i.e., the element that added last will be the first to come out.



C# Stack Declaration

Generally, c# will support both generic and non-generic types of stacks. Here, we will learn about non-generic queue [collections](#) by using the **System.Collections** namespace so you can add elements of different [data types](#).

As discussed, the [collection](#) is a [class](#), so to define a stack, you need to declare an instance of the stack [class](#) before we perform any operations like add, delete, etc. like as shown below.

Stack obj = new Stack();

*If you observe the above stack declaration, we created a new stack (**obj**) with an instance of stack class without specifying any size.*

C# Stack Properties

Property	Description
Count	It will return the total number of elements in a stack.

C# Stack Methods

The following are some of the commonly used stack methods to perform operations like add, delete, etc., on elements of a stack in the c# programming language.

Method	Description
Push	It is used to insert an object at the top of a stack.
Pop	It will remove and return an object at the top of the stack.

Method	Description
Clear	It will remove all the elements from the stack.
Clone	It will create a shallow copy of the stack.
Contains	It is used to determine whether an element exists in a stack or not.
Peek	It is used to return a top element from the stack.

C# Stack Example

```
namespace stackDemo
{
class Program
    {
static void Main(string[] args)
    {
        Stack st = new Stack();
        st.Push(101);
        st.Push("RAM");
        st.Push('S');
        st.Push(12.3f);

Console.WriteLine("Total Items..." + st.Count);

foreach (object i in st)
    Console.WriteLine(" " + i);

Console.WriteLine("Top most item is "+st.Peek());

st.Pop();
Console.WriteLine("Total Items after pop()..." + st.Count);

foreach (object i in st)
    Console.WriteLine(" " + i);

st.Clear();
Console.WriteLine("Total Items after removal..." +st.Count);
Console.ReadKey();
    }
}
```

1.4) C# Queue-

In c#, Queue is useful for representing a collection of objects that stores elements in FIFO (First In, First Out) style, i.e., the element that added first will come out first.

In a queue, elements are inserted from one end and removed from another end.

Generally, queues are useful when you want to access elements from the [collection](#) in same order that is stored, and you can store multiple null and duplicate values in a queue based on our requirements.



C# Queue Declaration

Generally, c# will support both generic and non-generic types of queues. Here, we will learn about non-generic queue [collections](#) by using the **System.Collections** namespace.

As discussed, the [collection](#) is a [class](#). To define a queue, you need to declare an instance of the queue [class](#) before performing any operations like add, delete, etc. like as shown below.

Queue que = new Queue();

If you observe the above queue declaration, we created a new queue (**que**) with an instance of a queue class without specifying any size.

C# Queue Properties

Property	Description
Count	It will return the total number of elements in the queue

C# Queue Methods

Method	Description
Enqueue	It is used to add elements at the end of the queue.
Dequeue	It will remove and returns an item from the starting of a queue.

Method	Description
Clear	It will remove all the elements from the queue.
Clone	It will create a shallow copy of the queue.
Contains	It is used to determine whether an element exists in a queue or not.
Peek	It is used to get the first element from the queue.
TrimToSize	It is used to set the capacity of a queue to an actual number of elements in the queue.

C# Queue Example

```
class Program
{
    static void Main(string[] args)
    {
        Queue q = new Queue();
        q.Enqueue(101);
        q.Enqueue("ABC");
        q.Enqueue('X');
        q.Enqueue(15.6f);

        Console.WriteLine("Total Items are..." + q.Count);

        Console.WriteLine("All Items in q QUEUE are...");
        foreach (object i in q)
            Console.WriteLine(i);

        Console.WriteLine("TOP MOST ITEM IS..." + q.Peek());

        q.Dequeue();

        Console.WriteLine("All Items in q QUEUE AFTER REMOVAL are...");
        foreach (object i in q)
            Console.WriteLine(i);

        Console.ReadKey();
    }
}
```

2) Generic collections [System.Collections.Generic classes]

Generic collections with work generic data type.

These collection classes 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.

2.1) C# List (List<T>)

In c#, List is a generic type of collection, so it will allow storing only strongly typed objects, i.e., elements of the same data type.

The size of the list will vary dynamically based on our application requirements, like adding or removing elements from the list.

In c#, the list is same as an ArrayList, but the only difference is ArrayList is a non-generic type of collection, so it will allow storing elements of different data types.

C# List Declaration

```
List<T> lst = new List<T>();
```

If you observe the above list declaration, we created a generic list (**lst**) with an instance of list class using type parameter (**T**) as a placeholder with an angle (<>) brackets.

Here, the angle (<>) brackets will indicate that the list is a **generic** type, and type parameter (**T**) is used to represent the type of elements to be accepted by the list.

In c#, the generic list (**List<T>**) is an implementation of the **ICollection<T>** interface, so we can also use **ICollection<T>** interface to create an object of the generic list (**List<T>**) like as shown below.

```
ICollection<T> lst = new List<T>();
```

C# List Initialization

```
List<string> lst = new List<string>();
```

*If you observe the above example, we defined a list (**lst**) with **string** data type to store only string elements.*

C# List Properties

Property	Description
Capacity	It is used to get or set the number of elements a list can contain.
Count	It is used to get the number of elements in the list.
Item	It is used to get or set an element at the specified index.

C# List Methods

Method	Description
Add	It is used to add an element at the end of the List.
AddRange	It is used to add all the elements of the specified collection at the end of the List.
Clear	It will remove all the elements from the List.
Contains	It is used to determine whether the specified element exists in the List or not.
CopyTo	It is used to copy the entire List to a compatible one-dimensional array.
Find	It is used to search for an element that matches the conditions defined by the specified predicate and returns the first occurrence of the List.
ForEach	It is used to iterate through the List to access elements.

Method	Description
Insert	It is used to insert an element into the List at the specified index.
InsertRange	It is used to insert all the elements of the specified collection into a List starting from the specified index.
Remove	It is used to remove the first occurrence of a specified element from the List.
RemoveAt	It is used to remove an element from the List based on the specified index position.
RemoveRange	It is used to remove a range of elements from the List.
Reverse	It reverses the order of List elements.
Sort	It sorts the elements in the List.

C# Generic List (List<T>) Example

```

using System;
using System.Collections.Generic;

namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {

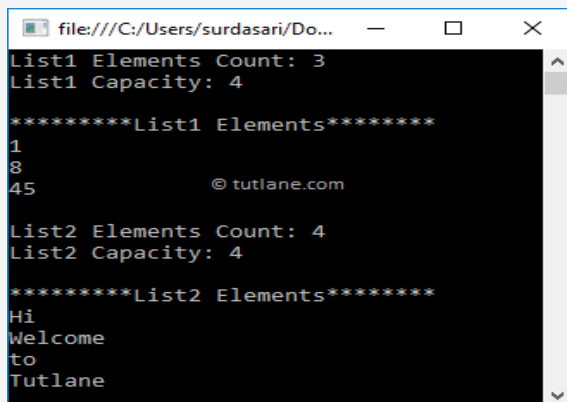
// Creating and initializing list
List<int> lst = new List<int>();
        lst.Add(1);
        lst.Add(8);
        lst.Add(45);
        Console.WriteLine("List1 Elements Count: " + lst.Count);
        Console.WriteLine("List1 Capacity: " + lst.Capacity);

```

```
Console.WriteLine("*****List1 Elements*****");
    // Accessing list elements
    foreach (var item in lst)
    {
        Console.WriteLine(item);
    }
List<string> lst2 = new List<string>();
lst2.Add("Hi");
lst2.Add("Welcome");
lst2.Add("to");
lst2.Add("Tutlane");

Console.WriteLine("List2 Elements Count: " + lst2.Count);
Console.WriteLine("List2 Capacity: " + lst2.Capacity);
Console.WriteLine("*****List2 Elements*****");
    foreach (var item in lst2)
    {
        Console.WriteLine(item);
    }
    Console.ReadLine();
}
}
```

Output



2.2) C# Dictionary

In c#, **Dictionary** is a generic type of collection, and it is used to store a collection of key/value pairs organized based on the key.

The dictionary in c# will allow to store only the strongly-typed objects, i.e., the key/value pairs of the specified data type.

In c#, while storing the elements in the dictionary object, **you need to make sure that the keys are unique because the dictionary object will allow us to store duplicate values, but the keys must be unique.**

The size of the dictionary object will vary dynamically so that you can add or remove elements from the dictionary based on our requirements.

In c#, the dictionary object is same as the hashtable object, but the only difference is the dictionary object is used to store a key-value pair of same data type elements.

C# Dictionary Declaration

```
Dictionary<TKey, TValue> dct = new Dictionary<TKey, TValue>();
```

If you observe the above dictionary declaration, we created a generic dictionary (**dct**) with an instance of dictionary class using type parameters (**TKey, TValue**) as placeholders with angle (<>) brackets.

Here, the angle (<>) brackets will indicate that the dictionary is a generic type and type parameter **TKey** represents a type of keys to be accepted by the dictionary, and **TValue** is used to represent a type of values to be accepted by the dictionary.

In c#, the generic dictionary (**Dictionary<T>**) is an implementation of **IDictionary<TKey, TValue>** interface, so we can also use **IDictionary<TKey, TValue>** interface to create an object of the generic dictionary (**Dictionary<TKey, TValue>**) like as shown below.

```
IDictionary<TKey, TValue> dct = new Dictionary<TKey, TValue>();
```

C# Dictionary Initialization

```
Dictionary<int,string> dct = new Dictionary<int, string >();
```

If you observe the above example, we defined a dictionary (**dct**) with the required key and value types to store. Here, the dictionary object will store a key of **int** type and value of **string** type.

C# Dictionary Properties

Property	Description
Count	It is used to get the number of key/value pair elements in the dictionary.
Item[TKey]	It is used to get or set the value associated with the specified key.
Keys	It is used to get a collection of keys in the dictionary.
Values	It is used to get a collection of values in the dictionary.

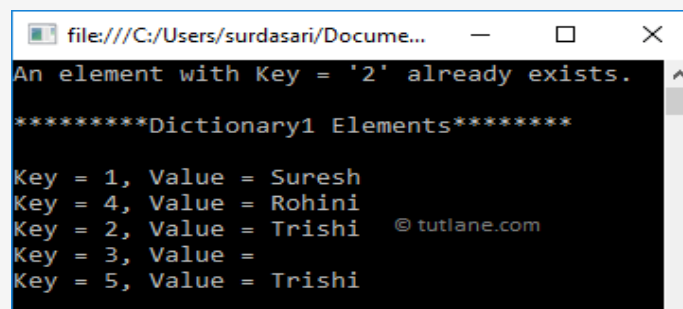
C# Dictionary Methods

Method	Description
Add	It is used to add elements to the dictionary object with a specified key and value.
Clear	It will remove all the keys and values from the Dictionary<TKey, TValue>.
ContainsKey	It is used to determine whether the specified key exists in Dictionary<TKey, TValue> or not.
ContainsValue	It is used to determine whether the specified value exists in Dictionary<TKey, TValue> or not.
Remove	It is used to remove an element from Dictionary<TKey, TValue> with the specified key.

Example

```
using System;
using System.Collections.Generic;
namespace Tutlane
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create a new dictionary with int keys and string values.
            Dictionary<int, string> dct = new Dictionary<int, string>();
            // Add elements to the dictionary object.
            // No duplicate keys allowed but values can be duplicate
            dct.Add(1, "Suresh");
            dct.Add(4, "Rohini");
            dct.Add(2, "Trishi");
            dct.Add(3, null);
            // Another way to add elements.
            // If key not exist, then that key adds a new key/value pair.
            dct[5] = "Trishi";
            // Add method throws exception if key already in dictionary
            try
            {
                dct.Add(2, "Praveen");
            }
            catch (ArgumentException)
            {
                Console.WriteLine("An element with Key = '2' already exists.");
            }
            Console.WriteLine("*****Dictionary1 Elements*****");
            // Accessing elements as KeyValuePair objects.
            foreach (KeyValuePair<int, string> item in dct)
            {
                Console.WriteLine("Key = {0}, Value = {1}", item.Key, item.Value);
            }

            Console.ReadLine();
        }
    }
}
```

Output-

```
file:///C:/Users/surdasari/Docume...
An element with Key = '2' already exists.
*****Dictionary1 Elements*****
Key = 1, Value = Suresh
Key = 4, Value = Rohini
Key = 2, Value = Trishi  © tutlane.com
Key = 3, Value = 
Key = 5, Value = Trishi
```

2.3) C# Genric Stack-

[Stack<T>](#) *It represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.*

Syntax- Stack<T> obj = new Stack<T>();

Ex. Stack<int> iob = new Stack<int>();

*If you observe the above example, we defined a Stack (iob) with **int** data type to store only integer elements.*

Example-

```
using System;
using System.Collections.Generic;
namespace GenericStackCollection
{
    public class Program
    {
        public static void Main()
        {
            Stack<int> stack = new Stack<int>();
            stack.Push(10);
            stack.Push(20);
            stack.Push(30);

            Console.WriteLine("Generic Stack Elements");
            foreach (var item in stack)
            {
                Console.WriteLine(item);
            }

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

2.4) C# Generic Queue-

Queue<T>: It represents a first-in, first-out collection of similar specified types of objects.

Syntax- Queue<T> obj = new Queue<T>();

Ex. Queue<int> iq = new Queue<int>();

If you observe the above example, we defined a Queue (iq) with **int** data type to store only integer elements.

Example

```
using System;
using System.Collections.Generic;
namespace GenericQueueCollection
{
    public class Program
    {
        public static void Main()
        {
            //Creating a Queue to Store Integer Values
            Queue<int> queue = new Queue<int>( );

            //Adding Elements to the Queue using Enqueue Method
            queue.Enqueue(10);
            queue.Enqueue(20);
            queue.Enqueue(30);
            //Adding Duplicate
            queue.Enqueue(30);

            Console.WriteLine("Generic Queue Elements");
            foreach (var item in queue)
            {
                Console.WriteLine(item);
            }

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