Collections in C#

In c#, the **collection** is a class that is useful to manage a group of objects in a flexible manner to perform various operations like insert, update, delete, get, etc., on the object items in a dynamic manner based on our requirements.

Generally, while working in c# applications, we will need to create or manage a group of related objects. In that case, we have two ways to create group objects in c#, i.e., using the arrays and collections.

In the previous section, we learned about arrays in c#, but those are useful only when working with a fixed number of strongly-typed objects. To solve this problem, Microsoft has introduced collections in c# to work with a group of objects that can grow or shrink dynamically based on our requirements.

In c#, the collection is a class. Hence, we need to declare an instance of the class before performing any operations like add, delete, etc., on the defined collection. The collections are implemented using the **IEnumerable** interface to access collection items by using a foreach loop.

C# Collection Types

In c#, we have a different type of collection classes are available; those are

- **Non-Generic** (System.Collections)
- **Generic** (System.Collections.Generic)
- **Concurrent** (System.Collections.Concurrent)

C# Non-Generic Collections

In c#, non-generic collection classes are useful to store elements of different data types, and these are provided by **System.Collections** namespace. These collection classes are legacy types, so whenever possible, try to use generic collections (**System.Collections.Generic**) or concurrent collections (**System.Collections.Concurrent**).

Following are the different type of non-generic collection classes which are provided by **System.Collections** namespace.

Class	Description
ArrayList	It is useful to represent an array of objects whose size is dynamically increased as required.
Queue	It is useful to represent a FIFO (First In, First Out) collection of objects.
Stack	It is useful to represent a LIFO (Last in, First Out) collection of objects.

Class	Description
Hashtable	It is useful to represent a collection of key/value pairs organized based on the hash code of the key.

C# Generic Collections

In c#, generic collections will enforce a type safety so we can store only the elements with the same data type, and these are provided by **System.Collections.Generic** namespace.

Following are the different type of generic collection classes which are provided by **System.Collections.Generic** namespace.

Class	Description
List	It is useful to represent a list of objects that can be accessed by an index.
Queue	It is useful to represent a FIFO (First In, First Out) collection of objects.
Stack	It is useful to represent a LIFO (Last in, First Out) collection of objects.
SortedList<K,V>	It is useful to represent a collection of key/value pairs that are sorted by a key.
Dictionary<K,V>	It is useful to represent a collection of key/value pairs organized based on the key.

C# Concurrent Collections

In c#, concurrent collections are useful to access collection items from multiple threads, and these are available from .NET Framework 4 with **System.Collections.Concurrent** namespace.

If we are using multiple threads to access a collection concurrently, then we need to use concurrent collections instead of **non-generic** and **generic** collections.

Following are the different type of concurrent collection classes which are provided by **System.Collections.Concurrent** namespace.

Class	Description
BlockingCollection	It is useful to provide blocking and bounding capabilities for thread-safe collections.
ConcurrentBag	It is useful to represent a thread-safe, unordered collection of objects.
ConcurrentDictionary<K,V>	It is useful to represent a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.
ConcurrentQueue	It is useful to represent a thread-safe FIFO (First In, First Out) collection.
ConcurrentStack	It is useful to represent a thread-safe LIFO (Last in, First Out) collection.
Partitioner	It is useful to provide partitioning strategies for arrays, lists, enumerables.
OrderablePartitioner	It is useful to provide a specific way of splitting an orderable data source into multiple partitions.

C# Arraylist

In c#, **ArrayList** is useful to store elements of different data types. The size of ArrayList can grow or shrink dynamically based on the need of our application, like adding or removing elements from ArrayList.

In c#, arraylists are same as arrays, but the only difference is that arrays are used to store a fixed number of the same data type elements.

C# ArrayList Declaration

In c#, ArrayList is a **non-generic** type of collection, and it is provided by **System.Collections** namespace.

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

```
ArrayList arrList = new ArrayList();
```

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

C# ArrayList Properties

The following are some of the commonly used properties of an arraylist in the c# programming language.

Property	Description
Capacity	It is useful to get or set the number of elements an arraylist can contain.
Count	It is useful to get the number of elements in an arraylist.
IsFixedSize	It is useful to get a value to indicate that the arraylist has a fixed size or not.
IsReadOnly	It is useful to get a value to indicate that the arraylist is read-only or not.
Item	It is used to get or set an element at the specified index.
IsSynchronized	It is used to get a value to indicate that access to arraylist is synchronized (thread-safe) or not.

C# ArrayList Methods

The following are some of the commonly used methods of an arraylist to add, search, insert, delete or sort elements of arraylist in c# programming language.

Method	Description
Add	It is useful to add an element at the end of the arraylist.
AddRange	It is useful to add all the elements of the specified collection at the end of the arraylist.
Clear	It will remove all the elements in the arraylist.
Clone	It will create a shallow copy of the arraylist.

Method	Description
Contains	It is useful to determine whether the specified element exists in an arraylist or not.
CopyTo	It is useful to copy all the elements of an arraylist into another compatible array.
GetRange	It is useful to return a specified range of arraylist elements starting from the specified index.
IndexOf	It is useful to search for a specified element and return a zero-based index if found; otherwise, return -1 if no element is found.
Insert	It is useful to insert an element in the arraylist at the specified index.
InsertRange	It is useful to insert all the specified collection elements into an arraylist starting from the specified index.
Remove	It is useful to remove the first occurrence of a specified element from the arraylist.
RemoveAt	It is useful to remove an element from the arraylist based on the specified index position.
RemoveRange	It is useful to remove a range of elements from an arraylist.
Reverse	It reverses the order of arraylist elements.
Sort	It sorts the elements in an arraylist.
ToArray	It copies all the elements of an arraylist to a new array object.

C# HashTable

In c#, **Hashtable** is used to store a collection of key/value pairs of different data types, and those are organized based on the hash code of the key.

Generally, the hashtable object will contain buckets to store elements of the collection. Here, the bucket is a virtual subgroup of elements within the hashtable, and each bucket is associated with a hash code, which is generated based on the key of an element.

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

When compared with dictionary objects, the hashtable will provide a lower performance because the hashtable elements are of object type, so the boxing and unboxing process will occur when we store or retrieve values from the hashtable.

C# HashTable Declaration

In c#, hashtable is a non-generic type of collection to store a key/value pair elements of different data types, and it is provided by **System.Collections** namespace.

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

```
Hashtable htbl = new Hashtable();
```

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

C# HashTable Properties

The following are some of the commonly used properties of hashtable in the c# programming language.

Property	Description
Count	It is used to get the number of key/value pair elements in the hashtable.
IsFixedSize	It is used to get a value to indicate that the hashtable has a fixed size or not.
IsReadOnly	It is used to get a value to indicate that the hashtable is read-only or not.
Item	It is used to get or set the value associated with the specified key.
IsSynchronized	It is used to get a value to indicate that access to hashtable is synchronized (thread-safe) or not.

Property	Description
Keys	It is used to get the collection of keys in the hashtable.
Values	It is used to get the collection of values in the hashtable.

C# HashTable Methods

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

Method	Description
Add	It is used to add an element with a specified key and value in a hashtable.
Clear	It will remove all the elements from the hashtable.
Clone	It will create a shallow copy of hashtable.
Contains	It is useful to determine whether the hashtable contains a specific key or not.
ContainsKey	It is useful to determine whether the hashtable contains a specific key or not.
ContainsValue	It is useful to determine whether the hashtable contains a specific value or not.
Remove	It is useful to remove an element with a specified key from the hashtable.
GetHash	It is useful to get the hash code for the specified key.

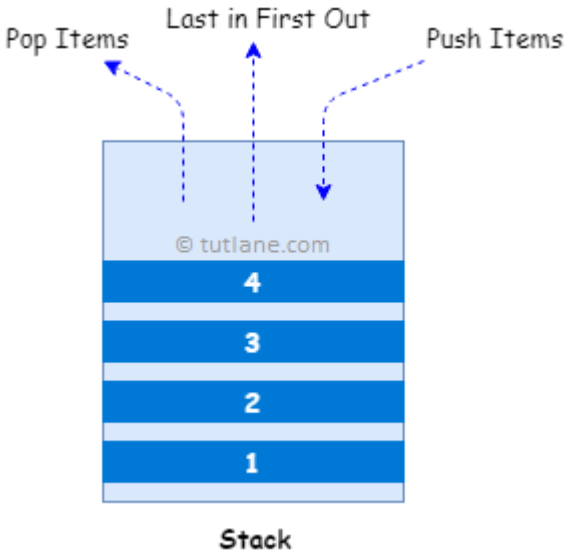
C# Stack with Examples

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.

Generally, stacks are useful when you want to access elements from the collection in a last-in-first-out style, and we can store multiple null and duplicate values in a stack based on our requirements.

By using **Push()** and **Pop()** / **Peek()** methods, we can add or retrieve an elements from the stack. Here, the **Push()** method is useful to add elements to the stack, and the **Pop()** / **Peek()** method is useful to retrieve elements from the stack.

Following is the pictorial representation of the stack process flow in the c# programming language.



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 stk = new Stack();
```

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

C# Stack Properties

The following are some of the commonly used properties of a stack in the c# programming language.

Property	Description
Count	It will return the total number of elements in a stack.
IsSynchronized	It is used to get a value to indicate that access to the stack is synchronized (thread-safe) or not.

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.
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# Queue with Examples

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.

Using **Enqueue()** and **Dequeue()** methods, we can add or delete an element from the queue. Here, the **Enqueue()** method is useful to add elements at the end of the queue, and the **Dequeue()** method is useful to remove elements start from the queue.

Following is the pictorial representation of the queue process flow in the c# programming language.

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

The following are some of the commonly used properties of a queue in the c# programming language.

Property	Description
Count	It will return the total number of elements in the queue
IsSynchronized	It is used to get a value to indicate that access to the queue is synchronized (thread-safe) or not.

C# Queue Methods

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

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.
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.