Collections

# Concurrent Collections

Concurrent collections (Namespace: **System.Collections.Concurrent**) are basically thread safe collections that are designed to be used in multithreading environment. They also provide type safety due to the generic implementation.

These collections should be used when they are getting changed or data is added/updated/deleted by multiple threads. If the requirement is only read in multithreaded environment, then generic collections can be used.

If locking is needed at a few places, manual locking or synchronization techniques can also be used however if it is required at several places, using concurrent collection is a good choice.

Concurrent collections are designed to be used in cases when excessive thread safety is required, overly using manual locking can lead to deadlock and other issues.

**Most commonly used Concurrent Collections**

1. **ConcurrentStack<T>**
2. **ConcurrentQueue<T>**
3. **ConcurrentDictionary<TKey,TValue>**

## ****ConcurrentDictionary<T>****

The following are the important methods.

### TryAdd()

TryAdd returns true if it’s successfully added else the key/value pair returns false primarily due to duplicate key.

### TryGetValue()

If key is present, TryGetValue fetches the value and returns true and if key isn’t present, it simply returns false without throwing any exception.

### TryRemove()

If key is present, **TryRemove** deletes the element and returns true else it simply returns false without throwing any exception.

### AddOrUpdate()

If key is present, **AddOrUpdate** updates the existing value with a new one and returns the new value else it adds the key with the value passed in the second parameter and returns the same. It is designed not to fail irrespective if key is present or not as it handles both the possible cases. Let’s have a look at the code below to understand how to use it.

### GetOrAdd()

If key is present, **GetOrAdd** fetches the value else it adds the key with the value passed in the second parameter and returns the same. Similar to AddOrUpdate, it is also designed not to fail irrespective if key is present or not as it handles both the possible cases.

# Immutable Collections

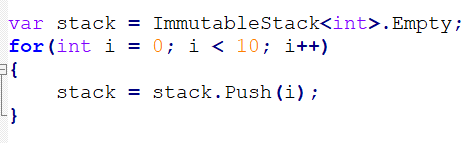
Any type that somewhat restricts changes to its state or the state of its instances can be described as immutable in some sense. The System.String type is an immutable type in the sense that the size of the string, the characters, and their order can’t change. The System.MulticastDelegate type, which is the parent of all delegate types, is immutable just like System.String. Both use an array as an underlying data structure and make a copy of it to satisfy a requested change, no matter how small the change is.

## The System.Collections.Immutable namespace

Immutable collections are those whose members cannot change once they have been created. The System.Collections.Immutable namespace comprises several immutable collections. This namespace contains immutable versions of Lists, Dictionaries, Arrays, Hashes, Stacks, and Queues.

The ImmutableStack<T> can be used to push and pop elements much the same way we do with mutable stacks. However, since ImmutableStack<T> is an immutable collection, its elements cannot be altered. So, when you make a call to the pop method to pop an element from the stack, a new stack is created for you and the original stack remains unaltered.

Let’s illustrate this with an example. The following code snippet shows how you can push elements onto an immutable stack.



## Immutable Lists

## Immutable Stacks

## Immutable Arrays

## Immutable Dictionary

# References

1. Concurrent Collections: <https://www.c-sharpcorner.com/article/concurrent-collections-in-net-concurrentdictionary-part-one/>