C# Collections Documentation

1. What is a Collection in C#?

A collection in C# is a class or interface that stores multiple elements (objects) in a single unit. Collections provide mechanisms to add, remove, search, sort, and manipulate data efficiently.

- Collections are part of System.Collections, System.Collections.Generic, and System.Collections.Concurrent namespaces.
- They eliminate the need to manage arrays manually when dynamic resizing or complex operations are required.

2. Types of Collections in C#

2.1 Non-Generic Collections

- Namespace: System.Collections
- Store objects as object type (boxing/unboxing required for value types).
- Type safety is not guaranteed at compile time.

Common Non-Generic Collections and Creation: - ArrayList:

```
ArrayList list = new ArrayList();
```

- Hashtable:

```
Hashtable table = new Hashtable();
```

- Queue:

```
Queue queue = new Queue();
```

- Stack:

```
Stack stack = new Stack();
```

- SortedList:

```
SortedList sortedList = new SortedList();
```

- BitArray:

```
BitArray bits = new BitArray(10);
```

2.2 Generic Collections

- Namespace: System.Collections.Generic
- Type-safe; store elements of a specified type <T>
- No boxing/unboxing overhead for value types.

Common Generic Collections and Creation: - List<T>:

```
List<int> list = new List<int>();
```

- Dictionary<TKey, TValue>:

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

- Queue<T>:

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

- Stack<T>:

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

- SortedList<TKey, TValue>:

```
SortedList<int, string> sortedList = new SortedList<int, string>();
```

- HashSet<T>:

```
HashSet<int> set = new HashSet<int>();
```

- LinkedList<T>:

```
LinkedList<string> linkedList = new LinkedList<string>();
```

- SortedSet<T>:

```
SortedSet<int> sortedSet = new SortedSet<int>();
```

2.3 Concurrent Collections

- Namespace: System.Collections.Concurrent
- Designed for thread-safe operations in multithreaded environments.

Common Concurrent Collections and Creation: - ConcurrentQueue<T>:

```
ConcurrentQueue<int> cQueue = new ConcurrentQueue<int>();
```

- ConcurrentStack<T>:

```
ConcurrentStack<int> cStack = new ConcurrentStack<int>();
```

- ConcurrentDictionary<TKey, TValue>:

```
ConcurrentDictionary<int, string> cDict = new ConcurrentDictionary<int,
string>();
```

- BlockingCollection<T>:

```
BlockingCollection<int> bCollection = new BlockingCollection<int>();
```

3. Differences Between Common Collections

3.1 List<T> vs HashSet<T>

Feature	List<T>	HashSet<T>
Type	Ordered, index-based	Unordered, no index
Duplicate Elements	Allowed	Not allowed
Access by Index	Yes	No
Search Time	O(n)	O(1) average
Use Case	Maintain insertion order, allow duplicates	Fast lookup, unique items
Methods	Add(), Remove(), Insert(), Contains(), IndexOf()	Add(), Remove(), Contains(), UnionWith(), IntersectWith()
Memory	Less overhead	Higher overhead

3.2 Queue<T> vs Stack<T>

Feature	Queue<T>	Stack<T>
Ordering	FIFO	LIFO
Access	Enqueue(), Dequeue(), Peek()	Push(), Pop(), Peek()
Use Case	Tasks scheduling, BFS	Undo functionality, DFS
Index Access	No	No
Thread-safe version	ConcurrentQueue<T>	ConcurrentStack<T>

3.3 Dictionary<TKey, TValue> vs Hashtable

Feature	Dictionary<TKey, TValue>	Hashtable
Generic	Yes	No
Key Type	Strongly typed	Object
Value Type	Strongly typed	Object
Boxing/Unboxing	No	Yes
Thread Safety	Not thread-safe	Not thread-safe
Performance	Faster	Slower
Namespace	System.Collections.Generic	System.Collections

3.4 SortedList<TKey, TValue> vs SortedSet<T>

Feature	SortedList<TKey, TValue>	SortedSet<T>
Elements	Key-value pairs	Single unique items
Ordering	Sorted by key	Sorted ascending
Duplicate Elements	No duplicate keys	No duplicate items
Index Access	Yes	No
Methods	Add(), Remove(), IndexOfKey(), IndexOfValue()	Add(), Remove(), UnionWith(), IntersectWith()
Use Case	Fast lookup by sorted key	Unique sorted collection

3.5 LinkedList<T> vs List<T>

Feature	LinkedList<T>	List<T>
Memory	More overhead (nodes)	Less overhead (array-based)
Access by Index	No (O(n))	Yes (O(1))
Insert/Delete at Middle	O(1) if node reference known	O(n)
Use Case	Frequent insertions/removals	Frequent random access

4. Summary Tips for Choosing Collections

- Use List<T> when you need indexed access and ordered elements.
- Use HashSet<T> when you need unique elements and fast lookup.
- Use Queue<T> for FIFO behavior.
- Use Stack<T> for LIFO behavior.
- Use Dictionary<TKey, TValue> for key-value mapping.
- Use Hashtable only for legacy code.
- Use LinkedList<T> for frequent insertions/deletions.
- Use Concurrent collections for multithreaded scenarios.