C# Collections

`IEnumerable<T> – ICollection<T> – IList<T>`

{ .... VERSUS .... }

IEnumerable<T>	Provides minimum functionality (enumeration only)
ICollection<T>	Provides medium functionality (e.g., the `Count` property)
IList<T>/IDictionary	Provide maximum functionality (including “random” access by index/key)

What is IEnumerable in C#

IEnumerable in C# is the basic collection .

it is an interface that defines one method, GetEnumerator which returns an IEnumerator interface. This allows readonly access to a collection then a collection that implements IEnumerable can be used with a for-each statement.

Enumeration interfaces don't provide a mechanism to determine the *size* of the collection, access a member by *index*, *search* , or *modify* the collection. For such functionality, the .NET Framework defines these collections …

ICollection
IList
IDictionary

ICOLLECTION<T>

ICollection<T> is the standard interface for countable collections of objects. It provides the ability to determine the size of a collection (*Count*), determine whether an item exists in the collection (*Contains*), copy the collection into an array (*ToArray*), and determine whether the collection is read-only (*IsReadOnly*). For writable collections, you can also *Add* , *Remove* , and *Clear* items from the collection. And because it extends IEnumerable<T>, it can also be traversed via the foreach statement.

ILIST<T>

IList<T> is the standard interface for collections *indexable* by position. In addition to the functionality inherited from ICollection<T> and IEnumerable<T>, it provides the ability to read or write an element by position (via an *indexer*) and insert / remove by position .

List<T> Definition

List<T> class provide dynamically sized array of objects and are among the most commonly used of the collection classes.

And has three constructors:

```
public List();
public List(IEnumerable<T> collection);
public List(int capacity);
```

Add Methods

```
public void Add (T item);

public void AddRange (IEnumerable<T> collection);

public void Insert (int index, T item);

public void InsertRange (int index, IEnumerable<T> collection);
```

Remove Methods

```
public bool Remove (T item);

public void RemoveAt (int index);

public void RemoveRange (int index, int count);

public int RemoveAll (Predicate<T> match);
```

INDEXING Methods

```
public T this [int index] { get; set; }

public List<T> GetRange (int index, int count);

public Enumerator<T> GetEnumerator();
```

Export and copy and Converting Methods

```
public T[ ] ToArray();

public void CopyTo (T[ ] array);

public void CopyTo (T[ ] array, int arrayIndex);

public void CopyTo (int index, T[ ] array, int arrayIndex, int count);

public ReadOnlyCollection<T> AsReadOnly();

public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput> converter);
```

Other Methods

```
public void Reverse();

public int Capacity { get; set; }

public void TrimExcess();

public void Clear();
```

HASHSET<T> - SORTEDSET<T>

- They do not store duplicate elements and silently ignore requests to add duplicates.
- Both collections implement `ICollection<T>` and offer methods that you would expect, such as `Contains`, `Add`, and `Remove`.
- Their `Contains` methods execute quickly using a hash-basedlookup.
- You cannot access an element by position.

Usefull Methods

`UnionWith` adds all the elements in the second set to the original set(excluding duplicates).

`IntersectWith` removes the elements that are not in both sets.

`ExceptWith` removes the specified elements from the source set.

`SymmetricExceptWith` removes all but the elements that are unique to one set or the other

DICTIONARIES

- Sorted Dictionaries
- ListDictionary
- OrderedDictionary
- Dictionary<TKey,TValue>
- IDictionary
- IDictionary<TKey,TValue>

Dictionaries

A dictionary is a collection in which each element is a key/value pair.

Dictionaries are most commonly used for lookups and sorted lists.

The classes in each dictionary differ in the following regard

- Whether or not items are stored in sorted sequence
- Whether or not items can be accessed by position (index) as well as by key
- Whether generic or nongeneric
- Whether it’s fast or slow to retrieve items by key from a large dictionary

Immutable Collections

Restricting the ability to write (mutate) a collection simplifies software and reduces bugs.

The immutable collections extend this principle

The immutable collections are built into .NET Core.

The disadvantage of immutability is that when you need to make a change, you must create a whole new object .

The key difference is that the methods that appear to alter the collection (such as `Add` or `Remove`) don’t alter the original collection; instead they return a new collection with the requested item added or removed.

Creating Immutable Collections

Each immutable collection type offers a `Create<T>()` method, which accepts optional initial values and returns an initialized immutablecollection:

```
ImmutableArray<int> array = ImmutableArray.Create<int> (1, 2, 3);
```

You can also create an immutable collection from an existing `IEnumerable<T>`, using appropriate extension methods(`ToImmutableArray`, `ToImmutableList`, `ToImmutableDictionary , ...`).

Finish …