C#

0 : INTRO dotNET

- INTRO
- COMPILATION PROCESS

1 : INTRO C# DATA TYPES

- INTRO
- DATA TYPES
- TYPE CASTING
- LITERALS

2 : STRING ARRA & MATH

- CONSOLE CLASS
- OPERATORS
- FUNCTIONS
- STRING CLASS
- MATH CLASS
- ARRAY

3 : DATA TYPES

- DATA TYPES
- VALUE TYPES
- REFREACE TYPES

4 : EXCEPTION HADLING

- EXCEPTION
- LOGGING EXCEPTION
- EXCEPTION FROM JAVA

5 : COLLECTION

- COLLECTION
- IEnumerable
- COLLECTION INTERFACES
- GENERICS

6 : EVENTS

- ROLES
- EVENTS
- CUSTOM EVENTS CREATION
- SUBSCRIBING & UNSUBSCRIBING
- DELEGATES

7 : THREADS

- INTRO
- THREAD LIFE CYCLE
- SYNCHRONIZATION IN THREAD
- ASYNC AWAIT
- TACK CLASS

8 : MEMORY MANAGEMENT

- INTRO
- Garbage Collector
- Stack & Heap Memory
- Assemblies
- CLR
- JIT

CH:05 COLLECTIONS

(#1) COLLECTIONS

* Collection

- Collection is a DS that allows, storing, managing & manipulating multiple Elements Efficiently
- Collections are dynamic arrays, which can grow their size as per need

* Types of Collection

1) Generic Collection: store elements of a specific data type.
   - Ensures type safety, better performance & avoid boxing - unboxing
   - These comes under the namespace as System.Collections.Generics
   - eg: List<T>
     Dictionary<T, T>
     SortedList<T>
     List<T>
     Stack<T>
     Queue<T>
     HashSet<T>
     LinkedList<T>
   - They use dynamic Array for internal implementation.

2) Non Generic: stores Elements As object type, meaning they require type Casting during retrival, leading to performance overhead due to boxing/unboxing
   - Even you can store any DataType value in between, not type safe
   - Syntax: ArrayList obj = new ArrayList();
   - These classes comes Under System.Collection namespace

non Generic	Generic	~~Description~~ Description
ArrayList	List<T>	A dynamic Array
HashTable	Dictionary<TKey, TVal>	Stores key value pair
SortedList	SortedList<T, T>	key value pair
Stack	Stack<T>	FILO
Queue	Queue<T>	FIFO

3) Concurrent Collection : these classes are under System.Collection.Concurrent namespace. These are thread safe and optimized for multi-threading applications

- These collections allow synultaneous access without race conditions
- lock in C# keyword for thread safety
- Internally it maintains an array of Locks
- it uses fine-grading locking to allow multiple threads to read & write simultaneously
- Concurrent Collections are same as Generic Collections but are Tread safe, meaning no two parallel running Tread (process) can insert data in these collection at the same time
- Concurrent Collection will automatically implement lock internally when one thread is accessing the Collection

eg: Collection	Description
ConcurrentBag<T> :	A thread safe unordered Collection like List
ConcurrentDictionary<TKey, TVal> :	A thread safe key-value pair
ConcurrentStack<T>	
ConcurrentQueue<T>	

4) Specialized Collections : These are under Collection.specialized namespace, these are used for special data storage

eg: NameValueCollection

- These collections are non Generick, don't take dataTypes

```
eg : NameValueCollection data = new NameValuesCollection();
     data.Add("lang", "C++");
     data.Add("lang", "C#");
```

- it allows multiple values for a specific key.
- Stores key-value pair, where multiple values can be associated with single key

5) Object Model Collections : under System.Collections.ObjectModel namespace,

- These collections provide only read & observable collections, making them ideal for data binding in UI applications like WPF
- eg: ReadOnlyCollection<T> : A collection that does not allow modification after Creation

#2 IEnumerable

* IEnumerable : this is the parent class or you can say this IEnumerable Interface is inherited by all the collection.

- your foreach loop works bcoz all collection are inheriting from IEnumerable, IEnumerable has a Method called GetEnumerator() & this method implements logic for the "foreach" loop
- Every collection class inhirits IEnumrable, & bcoz of this in Every collection there is one method GetEnumerator(), you need to implement this method in Custom Created collection to make for each loop to run
- In predefined Collection class it is already defined
- So IEnumerable is a Interface that is implemented by all the collection class, & bcoz they implement this IEnumerable interface, Every Collection class contains a GetEnumerator method() & bcoz of this method only we can use foreach loop on that collection
- if you want to make your own Custom collection, then you need to Implement this IEnumerable interface & implement GetEnumerator Method for working of foreach loop on that collection

#13 COLLECTION INTERFACES

*1) IEnumerable <T>

- Its a part of System.Collections.Generics namespace & is used for defining class (Creating Custom Collections) that can be enumerated (Iterate) using foreach loop
- basically this interface has a method called GetEnumerator() which returns a Enumrator (iterator) through which you can iterate over the complex collection types which you pass as <T> (like classes), when you create your own Custom Collections
- IEnumerable<T> is the base interface for all generic collection & all these collections inhirit it & provide implementation for the GetEnumerator() method so that you can use foreach loop to iterate the collection
- So when you create Custom Collections, you need to inhirit IEnumerator<T> interface & provide a implementation, so that you can use foreach loop to traverse over the Collection

eg: Creating Custom Collection & implementing IEnumerable<T> interface so that foreach loop can work on that custom collection

- while creating Custom Collections, internally it uses List<T> only.

```
namespace LibraryManagement
{
    public class Book
    {
        public string Tittle {get; set;}
        public Author {get; set;}
        public Book (string tittle, string author)
        {
            Tittle = tittle
            Author = author
        }
    }
```

```
public class Library : IEnumerable<Book>   // Custom Collection class
{
    Private List<Book> books = new List<Book>();
    public void AddBook (Book b)
    {
        books.Add(b);
    }

    public IEnumerator<Book> GetEnumerator()   // This method returns IEnumerator<T> (which is a Iterator)
    {
        // if you don't write implementation for this method then foreach loop will not work
        return books.GetEnumerator();   // as we know all generic class inherit IEnumerator or Interface & have implementation for the GetEnumerator() method so we are returning the GetEnumerator() method from the List<T> class, which has already implemented, the method & as all GetEnumerator method returns IEnumerator<T>. So you can use this
    }
    // you can also write custom implementation
}

public class Program
{
    static void Main()
    {
        var library = new Library();

        library.AddBook(new Book("name1", "F. Lavada"));
        library.AddBook(new Book("name2", "F. Lavada2"));
        library.AddBook(new Book("name3", "F. Lavada3"));

        foreach (var i in library)
        {
            Col($"{i.Title} by {i.Author}");
        }   // since library is a custom collection of books type (books is class), so if you not provide implementation for GetEnumerator() fun from IEnumerable interface then foreach loop will not work
        ∴ provide GetEnumerator() implementation, you can also just return Any Collections GetEnumerator() method as they have implemented this method
    }
}
```

* ICollection<T> Interface

- This ICollection<T> is part of the System.Collections.Generics namespace & extends IEnumerable<T>.
- This ICollection<T> provides methods for managing the collections such as adding, Removing, counting etc
- Every generic collection inherits this ICollection interface, & using this collections Methods like .Add, the genric classes implement it, so that you can use these methods

* IComparer<T> Interface

- This interface is used to define custom sorting or comparision logic for objects of type T.
- its mainly used to implement sorting logic when you make custom classes
- you need to implement the Compare() method from this interface which takes 2 <T> types of object as parameters & returns their comparision

* IList<T> Interface

- it extends the ICollection<T> interface.
- IList<T> provides functionalites for collections that support indexed based access to items like List
- All classes (generic collection classes) inherits this interface which does indexed based access.

* Heracichy

IEnumerable<T> : base class for all generic collection
- add functionality foreach loop

↳ ICollection : it inherits IEnumerable
- Add functionality of .Add, .Remove etc functions

* ISet<T> Interface

- using this ISet<T> interface, it provides collection of unique elements where duplicate values (elements) are not allowed.
- all the generic class like (collection class) hashTable, dictionary inherit it

#14 Generics

* Covariance & Contravariance

- **Generics** : Generics are a way to write code that works with any data type, while still being type-safe
- it provides code Reusibility & follows DRY principal.
- it also increases performance by Avoiding unnecessary type conversions (boxing / Unboxing)
- without Generics like the non Generic collections, they store data internally in object Type, so a lot of boxing (value to Refrence) & Unboxing is performed internally so use Generic collections
- Covariance & Contravariance describe how type parameters in generics behave when dealing with inheritance

* Covariance (out keyword)

Out: out key word is used to pass argument by refrence.
- variables passed as out don't need to be initialized before passing them. However the method must assign a value to the out parameter (at run time) before it returns
- so once the value is assign, you can use that variable anywhere bcoz the value assign is to the refrence. so, Even if multiple variable are there, they will point the same memory location.

eg:
```
static void calculate (int a, int b, out int sum)
{
    sum = a+b; //resultSum will be also pointing to same memory location as of sum
}
static void Main()
{
    int num1 = 5;
    int num2 = 10;
    int resultSum;
    Calculate (num1, num2, out resultSum);
}
```

↳ this sum will carry refrence of resultSum. //so sum & resultSum points same memory location

↳ // this will send refrence of resultSum

- So sum & resultSum will have value as 15.

Covariance : Covariance allows you to use a more derived type (dog) then originally specified (Animal)

- preserves assignment Compatibility
- eg: you have animal class & dog class. dog class inhirits Animals → if single animal type can be treated as Dog, then List of Dog can also be treated as List of Animal

```
Animal obj = new Dog();
List<Animal> obj2 = new List<Dog>(); // Below 3.5 version of .dotnet it was not allowed but in newer version its allowed.
```

- you can achive this using out keyword
- it applies to return types
- it is Useful for read-only operations (eg. retriving data, not modifing it)
- the out keyword makes it Covariant : meaning we can return derived class type (child) where base class type (parent) was expected.
- Covariance is applied when you only read data, that is return data from a generic type
- Covariance is Use ful when you need type flexibility like you want vehical class to return but you are returning car class
- Covariance ~~reference~~ refers to the ability to use more derived type (child class type) where a less derived type (base class) is expected without causing error. if you don't use out keyword then you will get Error.
- in simple word it allows you to treat an object of child class as if it was an object of parent class
- Useful when you are working with delegate
- you can use out keyword in interface<T> like interface<out T>, you can return derived types where base types are Expected.
- it Ensures type safty by allowing, only data retrival, not modification or insertion
- eg: public class Animal{}
  public class dog : Animal {}

// Covariance allows you to store a collection of dog objects and pass them as collection of Animal objects bcoz every Dog is Animal

- Use Case :

1) File Exporting System : you have base class as document & other derived class as PDF, txt, word, PPt etc & you want a system that exports document in various format then you can use Covariance
- Out keyword will allows returning derived types (PDF, word) as their base type (document)
- IFileExporter<out T> : interface allows us to return derived types format where base class was Expected. Because you are only retriving data, so this works out

```
namespace FileExportingSystem
{
    public class Document
    {
        Public string Tittle {get; set;}
        Public string Content {get; set;}
    }                                          // properties

    public class PdfDoc : Document
    {
        Public string PdfVersion {get; set;}
    }

    public class WordDoc : Document
    {
        public int WordVersion {get; set;}
    }

    public interface IFileExporter <out T>
    {
        T ExportDoc ();
    }

    Public class PdfExporter : IFileExporter <PdfDoc>
    {
        public PdfDoc ExportDoc ()
        {
            return new PdfDoc
            {
                Tittle = "My PDF Doc",
                Content = "This is PDF document";
                PdfVersion = "1.7".
            };
        }
    }

    Public class WordExporter : IFileExporter <WordDoc>
    {
        Public WordDoc ExportDoc ()
        {
            return new WordDoc
            {
                title = "My word Document",
                Content = "This is word",
                WordVersion = 2019
            };
        }
    }
```

// if you don't use "out" then covariance is not possible.
IFileExporter<PdfDoc> to IFileExporter<Document> this is not possible, you will get compile time error. Because of type safety

- Type safty: if interface is not covariant, then compiler enforces strict type matching
- Read only purpose: out keyword ensures that the T is used only for return types or read only purpose. Without out, the compiler assumes that you may want to both read & write T objects which breaks type safty
- out, tells the compiler that interface is read only for T types (its safe to treat PdfDoc or WordDoc as Document type when used for only reading or returning

```
// Syntax:
IEnumerable<Animal> a = new List<Dog> ();
```

Setting the Properties

```
class Program
{
    static void Main()
    {
        // Creating Specific Exporters.
        IFileExporter<PdfDoc> pdfExport = new PdfExporter();
        //decalare variable pdfExport as a refrence which points the
          obj of PdfExporter class
        // variable is of type interface which has been implemented
          by the obj which that variable is pointing
        IFileExporter<WordDoc> WordExport = new WordExporter();

        // Covariance : treating specific exporters as generic document Exporters
        IFileExporter<Document> generalPDFExport = pdfExport;
        //it shows covariance, since the interface is decalare with out, it
          allows assignment of IFileExporter<PdfDoc> to IFileExporter<Document>.
          this works becoz "PdfDoc" is derived from Document class

        //Exporting document PDF & displaying its details
        Document Pdf = generalPDFExport.ExportDoc();
        //calls the ExportDoc() method from generalPDFExport object
          (which is actually a PdfExport, PdfExport is obj of
          interface pointing to PdfExporter class object)
        //this returns a PdfDoc class object but it is stored as a
          Document refrence becoz of polymorphism

        Console.WriteLine($"PDF Doc title : {pdf.Title}");
    }
}
```

- Covariance allows delegate or interface to be assigned to a more generic type
- Covariance : you are returning a specific type from a general type using out keyword
- eg : you Ask for basket of fruits & you get basket of apples, since Apple is fruit, so you accept it
- Covariance allows collection of derived type (Dog) to be treated as a collection of their base type (Animal)
- our interfaces which are generic, all input parameters have in keyword for T & all output parameters have out for T type. eg : IEnumerable<T> here we have <out T> in its defination. Its Even same for delegate also (Action<in T, out T>)

```
namespace Covariance : eg using Collection for Covariance
{
    public class Animal
    {
        public string AnimalName { get; set; }
        public void virtual Speak()
        {
            Console.WriteLine("Animal is making sound");
        }
    }

    public class Dog : Animal
    {
        public override void Speak()
        {
            Console.WriteLine($"{Name} says : Woof! Woof! ");
        }
    }
    class Program
    {
        static void Main()
        {
            List<Dog> d = new List<Dog>{ //List<T> inherits IEnumerable<out T> interface
                new Dog { Name = "Buddy" },
                new Dog { Name = "Max" },
                new Dog { Name = "Jacki" },
            };
            IEnumerable<Animal> A = d; //you are creating refrence of interface<Animal> & pointing it to List of dogs
            foreach (Animal i in A)
            {
                i.speak() // it calls Dog speak method
            }
        } // IEnumerable<out T> interface allows covariance, enabling List<Dogs> to be treated as IEnumerable<Animal>
    }
}
// A specific type is treated as more generic type in Covariance using out keyword
```

Covariance - Output flexibility
- I expect an Animal, but fine if you give me a Dog

Covariance Meaning

//Covariance : Applies to return types only
- Returns a Dog where Animal was Expected
- more Specific (Dog) can be used (return) where more generic (Animal) was expected

eg: IEnumerable<Animal> G1 = new List<Dog>()
- you have used Dog where Animal was Expected as Return type

//you have inserted 3 dog in List d. by Initializing only

//it assigns List<Dog> d object to an IEnumerable<Animal> interface

since d is type of List<dog>. & Every List generic class inherits IEnumerable<T> interface where <out T> is defined

* ContraVariance (in keyword)

- **in**: in keyword is used for achieving ContraVariance with Generics interfaces & delegates
- it makes type as input-only (allowed for operation on type T), Meaning you can only pass data to it but cannot return it
- it allows Assignment Compatability with base type
- **Contravariance** : Accepts Animal class where Dog is expected
- Using ContraVariance you can assign a more Generic type (Animal) to a more specific type (Dog)
- Syntax :

```
IProcessor<Dog> dogProcess = new Animal(); // जहा पर Dog class की object चाहिए वहा animal भेज दिया
// in place where Dog (more derived) was Expected, Animal (more General) was send
```

- ContraVariance is only Applied for input <T> parameters, (it cannot be used for Return types)
- you Cannot Return <T> from the method where (in) ContraVariance is being applied.
- ContraVariance - Input flexibility : Animal obj can be passed where Dog obj was Expected

```
Interface<Dog> d = new Animal();
```

* Real-Life Use Case

eg: using System;

```
namespace ContraVariance
{
    public class Animal
    {
        public string Name {get; set;}
    }

    public class Dog : Animal
    {
        public string Breed {get; set;}
    }

    public interface IProcessor<in T> //in keyword Enables ContraVariance
    {
        void Process(T item);
    }

    public class AnimalProcessing : IProcessor<Animal>   // Animal is Expected but sending Dog
    {
        public void Process(Animal obj)
        {
            Console.WriteLine($" Processing Dog : {obj.Name} ");
        }
    }

    class Program
    {
        static void Main()
        {
            IProcessor<Animal> animalObj = new AnimalProcessing();

            IProcessor<Dog> dogobj = animalObj;

            Dog d = new Dog {Name = "Parshay", Breed = "Jermansh."};

            dogObj.Process(d);

        }
    } //output:
```

Processing Dog : Parshay

CH : 06 COLLECTION FRAMEWORK

#1) INTRO

* ALTERNATIVE TO CPP STL

- As we have STL in cpp, Similarly we have Collection framework in Java
- Alternatives for following

Sr no.	CPP	Java	C#
1	1) Vector< > (dynamic Array) : eg: vector<int> vec;	ArrayList<>, Vector<> ArrayList<Integer> list_1; Vector<Integer> v1;	List<T> List<int> list_1;
2	2) set<> (ordered unique collection) : eg: set<int> s;	TreeSet<> (sorted set) TreeSet<Integer> s;	SortedSet< > (sorted set) SortedSet<int> s;
	3) unsorted_set< > eg: unsorted_set<int> s;	HashSet<datatype> HashSet<Integer> s;	HashSet< > HashSet<int> s;
3	4) map< > (ordered key-value pair) : eg: map<int, string> mp;	TreeMap<Integer, String> (sorted map) TreeMap<Integer, String> mp;	SortedDictionary< > (Sorted map) SortedDictionary<int, int>
	5) unordered_map< > (unordered map) eg unordered_map<int, int>	HashMap< > HashMap<Integer, String> mp;	Dictionary< > Dictionary<int, int> md;
4	6) pair< > (store key-value pair) eg: pair<int, char> p;	make custom class for pair	KeyValuePair< > KeyValuePair<int, int> p;

Sr no.	CPP	Java	C##
5	7) Stack < > (last in, first out) eg: Stack < int > st; st.push (10); st.pop ( );	Stack < > Stack<Integer> st ; st.push(10); st.pop ( );	Stack < > Stack<int > st ; St.Push(20); St.pop ( );
6	8) queue < > (first in, first out) eg: queue<int > qu qu.push (10); qu.pop( );	Queue < > Queue <Integer > qu; qu.add (10); qu.poll ( );	Queue < > Queue <int> qu; qu.Enque (100); qu.Dequeue ( );

#2 VECTORE, ARRAYLIST, LIST STL

1* Vector c++

- Vector is a STL library in c++, which means dynamic array
- syntax:

1) `vector<int> vec ; // default initialization`
2) `vector <int> vec (size) ; // initialization with size`
3) `vector <int> vec (size, value) // all the index will be filled with given value`
4) `vector <int> vec = {1, 2, 3, 4} // assign specific values`
5) `vector < vector <int> > matrix ; // 2D vector initialization`

* Methods of vector library

1*) `vec.push_back(ele)` : adds an element passed at the end of the vector

eg:
```
int main()
{
    vector<int> vec;
    vec.push_back (10);
    vec.push_back (20);
}
```

2) `vec.pop_back()` : removes the last element from vector

eg:
```
int main()
{
    vector <int> vec = {10, 20, 30, 40}
    vec.pop_back()
}
```

3) `vec.size()` : returns the no. of elements in vector

eg:
```
int main()
{
    vector<int> vec = {10, 20, 30, 40};
    cout << vec.size() << endl;
```

4) vec.empty() : returns true if vector is Empty

5) vec.at(index) : returns the element at that index

eg:
```
int main()
{
    vector<int> vec = {10, 20, 30, 40}
    if (!vec.empty())
    {
        cout << "at index 1" << vec.at(1) << endl;
    }
}
```

6) vec.insert(index, value) : it inserts the value at the specified index

eg:
```
int main()
{
    vector<int> vec = {10, 20, 100, 50}
    vec.insert(2, 30);
    vec.insert(vec.begin()+3, 40)
    for (int i : vec)
    {
        cout << i << " " << endl; //output:
                                  10, 20, 30, 40
    }
}
```

7) vec.front() : returns the first element

8) vec.back() : returns the back element

9) vec1.swap(vec2) : swaps the elements of 2 vectors

eg:
```
int main()
{
    vector<int> v1 = {1,2,3,4}
    vector<int> v2 = {10,20,30}
    vec1.swap(v2);
    for (int i : v1)
        cout << i;
```

10) vec.begin() : returns the beginning iterator for traversal

11) vec.end() : returns the end iterator for traversal

* Iterators in vector

Syntax:

```
vector <data_type> :: iterator i;
```

eg:
```
int main()
{
    vector <int> vl = {10, 20, 30, 40};
    vector <int> :: iterator i;
    i = vl.begin();
    for (i=vl.begin(); i!=vl.end(); i++)
    {
        cout << vl[i] << endl;
    }
}
```

Syntax:

```
auto i = vec.begin()   // "auto" automatically creates an iterator for traversal
```

eg:
```
int main()
{
    vector <int> vl = {100, 200, 300};
    auto i = vec.begin();
    for ( ; i!=vec.end(); i++)
    {
        cout << vec[i] << " ";
    }
}
```

* find() : its a fun from algorithm library, used to find an element, returns the index if element is find or else return the last index

eg:
```
int main()
{
    vector <int> vl = {10, 20, 30, 40};
    auto it = find(vec.begin(), vec.end(), 30);
    if (it != vec.end())
        cout << "Element found";
    else
        cout << "not found";
}
```

* sort the vector

- using the sort() fun of algorithm library you can sort anything like vector or string
- syntax:

```
sort (vect.begin(), vect.end());
```

eg:
```
int main()
{
    vector<int> v1 = {10, 30, 90, 20, 0, -10}
    sort (v1.begin(), v1.end());
    for (int i : v1)
    {
        cout<<i;
    }
}
```

* reverse a vector

- using the reverse fun of algorithm library you can revoke anything

eg:
```
int main()
{
    vector<int> v1 = {10, 20, 30, 40}
    reverse (v1.begin(), v1.end());
    for (int val : v1)
        cout<<val;
}
```

NOTE
- you can access vector elements like array indices

* swap values

- using swap fun from algorithm

syntax:
```
swap (value1, value_2)
```

eg:
```
int main()
{
    swap (v1[3], v1[0]);
    for (int i : v1)
        cout<<i;
}
```

#12 List in C#

- List <data_type> is a genric collection provided by System.collections.Generics namespace in C#
- it is similar to vectors of c++, used to store dynamic size array
- you can access the List index similar as of array & vector in c++
- syntax:

1) `List<data_type> v1 = new List<int>();` // initializing empty list

2) `List <int> v2 = new List <int> (size);` // initialization by specifing size of list

3) `int[] arr = {10, 20, 30};`
`List <int> v3 = new List<int> (arr);` // you can pass existing array or other list

4) `List<int> v4 = new List<int> {10, 20, 30}` // specifying the elements initially

5)
```
List < List <int> > matrix = new List < List <int> > ();
// initializing 2D matrix list
foreach (var i in matrix) // displaying matrix using for each loop
{
    foreach (var j in i)
    {
        Console.Write (j + " ");
    }
    Console.Write ();
}
```

* Methods of List

1) ll.Add(ele) : Adds an element (object, List etc) to the end of list

2) ll.AddRange(newList) : Adds the elements of a collection to the end of ll List

eg:
```
namespace project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            List<int> num1 = new List<int>();
            num1.Add(10);
            num1.Add(20);
            List<int> num2 = new List<int>{30,40,50};
            num1.AddRange(num2);
            foreach(var ele in num1)
                Console(ele);
        }
    }
}
```

3) ll.Insert(index, element) : Inserts the element at specified index & shifts all elements after that one place ahead.

4) ll.InsertRange(index, List-new) : Inserts the list at the specified index by shifting reaming elements one place ahead

eg:
```
static void Main(string[] args)
{
    List<int> num1 = new List<int>();
    num1.Insert(0,1);
    num1.Insert(1,100);
    List<int> num2 = new List<int>{30,40,50};
    num1.InsertRange(1,num2);
    foreach(var ele in num1)
    {
        Cwl(ele);
    }
}
```

5) ll.RemoveAt(index) : removes the element at the specified index

6) ll.RemovesAll (condition) : removes all the elements that matches the condition defined by predicate

eg: Static void Main (string[] args)
{
List<int> num1 = new List<int> { 1, 2, 3, 4 };
num1.RemoveAt (0);
num1.RemoveAt (1);
num1.RemovesAll (x => x > 3); // removes element greater than 3
}

7) ll.Contains (element) : returns true if the element is present in the list or not

8) ll.IndexOf (element) : returns the index of first occurance of a specified element, or else -1 is returned

eg: Static void Main (string[] args)
{
List<int> num1 = new List<int> {1, 2, 3, 4};
Console.WriteLine (num1.Contains (3)); // index of element 3 ~~will be returned~~ :: true
int ind = num1.IndexOf (4); // index of element 4 will be returned
Console.WriteLine (ind);

9) ll.Sort() : Sorts the elements of List in ascending order

10) ll.Reverse () : Reverse the order of elements in the list

eg: Static void Main (string[] args)
{
List<int> num1 = new List<int> { 1, 2, 3, 4, 5 };
num1.sort ();
foreach (var ele in num1)
Cwl (ele);
num1.reverse ();
foreach (var ele in num1)
Cwl (ele);
}

11) ll.ToArray () : Converts List into array & returns it

eg: List<int> num1 = new List<int> {1, 2, 3, 4, 5};
int[] arr = num1.ToArray ()

12) li.Count : returns the size of the list

* Accessing Indices

- you can access the index directly like array
- it will return the value at that index

eg:
```
int ele = list[2];
int lastele = list[list.count-1]; // get last element
```

* Iterating List

Syntax :
```
foreach (var ele in list)
    Console.WriteLine(ele);
```

eg:
```
List<int> num1 = new List<int>{1,2,3,4};
foreach (var e in num1)
{
    Console.WriteLine(e);
}
```

NOTE
- you can access List element like array indices

Syntax :
```
for (int i=0; i<list.Count ; i++)
{
    CWL(list[i]);
}
```

eg:
```
List<int> num1 = new List<int>{1,2,3,5,6};
for (int i=0; i<num1.Count ; i++)
{
    Console.WriteLine(num1[i]);
}
```

* How to get any element

syntax:
```
int ele = num[0]; //ele = 1
int ele1 = num[2]; //ele1 = 3
```

#13 SET , TREESET , SORTEDSET STL

#1 Set C++

- Set is a C++ STL library, it is a collection of unique sorted elements
- Syntax:
  1) `set<int> s;`
  2) `set<int> s = {10, 20, 40, 50};`
  3) `int arr[] = {5, 10, 20, 15, 25};`
     `set<int> s(arr, arr+5);`

* Methods of set library

1) `s.insert(ele)` : adds an element in the set, if it already exists, it does not insert a duplicate value

2) `s.find(ele)` : Searches for an element in the set, returns the iterator if element found, or else returns `s.end()` if not found

eg:
```
int main()
{
    set<int> s = {10, 20, 300, 40, 5};
    auto it = s.find(20);
    if (it != s.end())
    {
        cout << "20 found at" << it;
    }
    s.insert(200);
    s.insert(150);
}
```

3) `s.count(ele)` : Returns the no. of occureence of an element (always 0 or 1)

4) `s.size()` : Returns the no. of element in the set

eg:
```
int main()
{
    set<int> s = {10, 20, 30, 40, 50};
    cout << s.size();
}
```

5) s.empty() : returns true if the set is empty

6) s.begin() : returns the iterator to first element for traversal

7) s.end() : returns the iterator to last element for traversal

* traversing set

syntax :

```
for (int ele : myset)
{
    cout<<ele;
}
```

NOTE
- you cannot access set element using array style indices

eg:
```
int main()
{
    set<int> s = {10, 20, 30, 40};
    for (int ele : s)
    {
        cout<<ele;
    }
}
```

syntax :

```
for (auto it = s.begin(); it!=s.end(); it++)
{
    cout<<*it;
}
```

eg:
```
int main()
{
    set<int> s = {10, 20, 30, 40, 50}
    for (auto it = s.begin(); it!=s.end(); it++)
    {
        cout<<*it;
    }
}
```

8) s.earse(ele) : specified element will be removed from set

eg:
```
int main()
{
    set<int> s = {10, 20, 30, 40};
    s.earse(30);
}
```

#2 HashSet & SortedSet in C#

- **HashSet** : is the unsorted collection of unique elements similar to C++ unsorted_set, from the system.Collections.Generic namespace
- **SortedSet** : is the sorted collection of unique elements
- set does not allow duplicates elements entry

* Methods in SortedSet C#

1) S.Add(ele) : Adds an item (element) to the set

2) initialization syntax:
   - HashSet<int> s = new HashSet<int>(); //initialize normally
   - HashSet<int> s = new HashSet<int>{1,2,3,4};
   - HashSet<int> s = new HashSet<int>(list_1); //send list elements

3) S.Contains(ele) : returns true if that element exists in the set

eg:
```
namespace Project1
{
    internal class Program
    {
        static void Main(string[] arg)
        {
            SortedSet<int> s = new SortedSet<int>{1,2,3,4};
            S.Add(10);
            S.Add(20);
            S.Add(30);
            S.Remove(2); //removes element 2
            if (s.Contains(10))
            {
                Console.WriteLine("10 is present");
            }
        }
    }
}
```

4) S.Removes(ele) : removes the element from the set

5) S.Count : returns the no. of elements in the set

* traversing Set in C#

syntax:

```
foreach (int ele in s)
{
    cwl(ele);
}
```

6) S.ElementAt(index) : returns the element at that index, used for traversing

eg:
```
static void Main()
{
    SortedSet<int> s = new SortedSet<int> {50, 40, 30, 10, 20}
    for (int i=0; i< s.Count ; i++)
    {
        Console.WriteLine (s.ElementAt (i));
    }
}
```

7) S.ToList () : returns a List of that set elements
- Set elements are converted to List & List is returned

eg:
```
static void main ()
{
    HashSet<int> s = new HashSet<int> {10, 20, 30, 40, 50}
    List<int> l = s.ToList ();
    for (int i=0; i< l.Count ; i++)
    {
        Console.WriteLine (l[i]);
    }
}
```

NOTE
- We cannot access set elements like array indices

#4 MAP , TREEMAP , SORTEDDICTIONARY STL

#1 map C++

- map in C++ is a STL library which stores key - value pair ,
- it only stores unique keys, no duplicate keys
- syntax:

1)
```
map<int, int> mp ;
mp[1] = "one";   // assign value like array
mp[2] = "two";      map[key] = value
```

2)
```
map<int, string> mp = {
        {1, "one"},
        {2, "two"},
        {3, "three"}
    };
```

3)
```
map <int, string> mp ;
mp.insert ({1, "one"});
mp.insert ({2, "two"});
mp.insert (make_pair (4, "four"));
```

* methods

1) mp.insert (key, value) : inserts a "key value" pair at the end of your map

2) mp.find (key) : searches for an element with the specified key . if found then returns an iterator by which you can access that key & value (i.first , i.second) , otherwise it returns an iterator to mp.end()

eg:
```
int main ()
{
    map<int, string> mp = {{1, "one"}, {2, "two"}};
    auto it = mp.find (1);
    if (it != mp.end())
            cout << it.first << it.second ;
}
```

3) mp.size() : returns the no. of key-value pair in map

4) mp.begin() : returns to first iterator

5) mp.end() : returns to end iterator

* traversing

Syntax:

```
map<int, string> ::iterator it = mp.begin();
for (it=mp.begin() ; it!=mp.end() ; it++)
{
    cout << it.key << it.second
}
```

Syntax:

```
for (auto& pair : mp)
{
  cout<<pair.first << pair.second ;
}
```

#2 Dictionary in C#

- Dictionary : it uses a hash table for implementing it, it is similar to map of C++, only unique keys are inserted
- SortedDictionary : it uses Red Black tree.
- Syntax :

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

2)
```
Dictionary<int, string> dict = new Dictionary<int, string>{
        {1, "one"},
        {2, "two"},
        {3, "three"}
};
dict[4] = "four"; //assign value using []
dict[5] = "five";      [key] = "value"
```

* Methods

1) `Add(key, value)` : Adds a key value pair in dictionary

2) `dict.ContainsKey(key)` : returns true if that key is present.

3) `TryGetValue(key, out datatype val)` : gives value associated with that specific key, if key does not exists returns false

eg:
```
namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> dict = new Dictionary<int, string>();
            dict.Add(1, "apple");
            dict.Add(2, "banana");
            if (dict.ContainsKey(2))
            {
                cwl("key 2 exists " + dict[2]);
            }
            if (dict.TryGetValue(1, out string val));
            {
                cwl("value" + val);
            }
            else
                cwl("key 1 not found");
        }
    }
}
```

4) dict.Remove(key) : Removes the specified key value pair the map

5) dict.Count : Returns the no. of "key-value" pairs in map

6) dict.ContainsValue (v) : Returns true if specified value exists in the map, or else false

eg: internal class Program

```
{
    static void Main (string [] args)
    {
        Dictionary<int, string> dict = new Dictionary<int, string> ();
        dict [1] = "apple" ;
        dict [2] = "banana";
        dict [3] = "pineapple";
        dict.Remove (3);

        foreach (var pair in dict)
        {
            Console.WriteLine ($" key is {pair.key}, value : {pair.Value}");
        }
        Console.WriteLine (dict.Count);
        if (dict.ContainValue ("apple");
        {
            cw ("apple is there");
        }
    }
}
```

// insert element like array
dict [key] = "value"

7) dict.ToList () : Returns & converts the dictionary to a list which stores the pair

eg:

```
static void Main (string () args)
{
    Dictionary <int, string> dict = new Dictionary <int, string> {
                {1, "apple"},
                {2, "banana"},
                {3, "cat"}
    };
    List<KeyValuePair<int, string> list = dict.ToList ();
    foreach (var i in dict)
    {
        cw ($" key is {i.key} value is : {i.value}");
    }
}
```

* traversing Dictionary

syntax:

```
foreach (var i in dict)
{
    Console.WriteLine ($" {i.key} {i.value} ");
}
```

syntax:

```
foreach (var key in dict.keys)
{
    CWL(key);
}
foreach (var val in dict.Values)
{
    CWL(val);
}
```

eg:

```
static void Main()
{
    Dictionary<int, string> dict = new Dictionary<int, string> ();
    dict.Add (1, "apple");
    dict.Add (2, "banana");

    foreach (var i in dict)
    {
        CWL ($" {i.key} {i.Value} ");
    }

    foreach (var key in dict.keys)
    {
        CWL (key);
    }

    foreach (var val in dict.Values)
    {
        CWL (val);
    }
}
```

8) dict.keys : keys returns collection of all keys. used to print keys

9) dict.Value : Value returns collection of all values. used to print value

* Sorting Dictionary

Syntax:

1)
```
var sortdict = dictionary.OrderBy (pair => pair.key);
foreach (var i in sortdict)
{
    Cwl (i.key + i.value);
}
```
↳ // sort on basis of keys

2)
```
var sortdict = dictionary.OrderByDecending (pair => pair.value);
foreach (var i in sortdict)
{
    Cwl (i.key + i.value)
}
```
↳ // order in decending order

↳ // sorting on basis of values

eg:
```
static void Main (string [] args)
{
    Dictionary <int, string> dict = new Dictionary <int, string> ()
    dict [1] = "apple";
    dict [2] = "banana";
    dict [10] = "kela";
    dict [5] = "lavada";

    var sortdict = dict.OrderBy (pair => pair.key);
    foreach (var i in sortdict)
    {
        cwl (i.key + i.value);
    }

    var revsortdict = dict.OrderByDecending (pair => pair.value);
    foreach (var i in revsortdict)
    {
        cwl (i.key + i.value);
    }
}
```

#5 PAIR , KEYVALUEPAIR STL

#1 Pair<> C++

- Pair is a STL library that allows you to store a pair of two values. It is not a collection like vector
- Its useful when you need to return two values from a fun or store - key value pair in map
- Syntax :

```
1) pair<int, int> p;

2) pair<int, int> p1 (1, "apple");

3) auto p2 = make_pair (2, "banana"); // "auto" keyword automatically assigns the type

3) vector < pair<int, string >> vec
   vec.insert (makepair (1, "banana");
```

- you cannot iterate elements of pair like in arrays
- Pair does not store elements in sorted order

* Methods

1) p.first & p.second : these first & second members allows you to access the two values stored in pair

2) make_pair ("key", value) : make_pair() is a function to that creates a pair without explicitly specifying the types of elements . it deduces the types automatically

eg:

```
int main()
{
    pair<int, string> p(1, "kela");
    cout<< p.first << p.second <<endl;

    auto p2 = make_pair(2, "mck");

    map<int, string> mp;   // using make_pair() with map
    mp.insert(make_pair(1, "one");
    mp.insert (make_pair (2, "two");
    for(auto& p: mp)<
    cout << p.first << p.second ;
    }

}
```

#2 KeyValuePair<> C#

- KeyValuePair<> is a structure in C# used to represent key value pair. its mostly used in collections like Dictionary
- you can't store values like collection, contains only one key pair
- KeyValuePair is not a collection like List
- Syntax:

```
KeyValuePair<int, string> pair = new KeyValuePair<int, string>(
        (1, "one");
}
```

* Methods

1). .key & .Value : key returns you the key & value returns you the value stored in that pair

2) new KeyValuePair<int, int>(1, 111) : using "new" you can create a new pair of key value to insert it in dictionary or List or Array

- eg:

```
void main()
{
    Dictionary<int, string> dict = new Dictionary<>(
            {1, "two"},
            {2, "three"},
            {3, "four"}
    );
    foreach (KeyValuePair i in dict)   // data type of dictionary is KeyValuePair
    {
        cout(i.key + i.Value);
    }
    //using List with Pair
    List<KeyValuePair<int, string>> List = new List<>();
    {
        new KeyValuePair<int, string>(1, "one"); // inserts the pair in List
        new KeyValuePair<int, string>(2, "two");
    };
    foreach (var i in list)
    {
        cout(i.key + i.Value);
    }
}
```

#16 Stack STL

#1) Stack <> C#

- A stack in C# is a collection that follows LIFO
- syntax :

```
Stack<int> st = new Stack<int>();
```

* Methods

1) st.Push(ele) : Push fun Adds an element passed in the stack
2) st.Pop() : Removes and returns the top most element from stack
3) st.Peek() : Returns the element at the top of stack without removing it
4) st.Count : Count returns the no. of elements inside stack
5) st.Contains(ele) : returns true if that element is present inside stack
6) st.ToArray() : returns all stack elements converted to Array

eg:

```
static void Main(string[] args)
{
    Stack<int> st = new Stack<int>();
    st.Push(1);
    st.Push(2);
    st.Push(3);
    st.Pop(); // 3 will be removed
    st.Peek(); // 2 will be returned
    int no = st.Count
    Cwl(no); // 2 elements are there :. 2 is the count

    st.Contains(1); // returns true

    int[] arr = st.ToArray(); // converted to array
    foreach(var i in arr)
    {
        Cwl(i);
    }
}
```

NOTE
- you cannot sort stack, Queue bcoz it's LIFO & FIFO order

* traversing Stack

- Syntax:

```
foreach (var ele in st)
{
    cwl(ele);
}
```

#12 Stack C++

- Stack is a STL collection in C++ which follows FILO
- syntax:

```
stack<int> st;
```

* methods

1) st.push(ele) : push (Adds) element passed into the stack
2) st.pop() : Removes the element from top
3) st.top() : Returns the element at top but not remove it
4) st.size() : Returns the no. of elements in stack

eg:
```
int main()
{
    stack<int> st;
    st.push(10);
    st.push(20);
    st.push(30);
    st.pop(); // 30 is removed
    st.top(); // 20 is returned
    cout << st.size(); // 2 is printed
}
```

* traversing

- copy all elements of stack into other temp stack & then use pop & top fun to travers the stack
- eg:

```
int main()
{
    stack<int> st;
    st.push(10);
    st.push(20);
    st.push(30);
    stack<int> temp = s;   // copy the stack to temporary stack & then using pop & top fun traverse the tempory stack

    while(!temp.empty())   // st.empty() : fun returns true if stack is empty
    {
        cout << temp.top();
        temp.pop();
    }
}
```

#7 Queue STL

#1 Queue C++

- The Queue library in C++ is part of STL & provides a FIFO container for storing elements
- Syntax :

```
queue<int> q;
```

* Methods

1) q.push(ele) : Adds an element in queue at back
2) q.pop() : removes an element from front of queue, do not return the removed element
3) q.front() : returns the front element
4) q.back() : Returns the back element
5) q.empty() : returns true if queue is empty
6) q.size() : returns no. of elements in queue

* Traversing

- for traversing, copy the queue elements into another queue & use pop & front operation to get the element & traverse queue

- eg :

```
int main()
{
    queue<int> q;
    q.push(10);
    q.push(20);
    q.push(30);
    q.pop();
    if(!q.empty)
        cout<<" not Empty";
    cout<<q.size();

    queue<int> temp = q;
    while(!temp.empty()); ----> // for traversing queue
    {
        cout<<temp.front();
        temp.pop();
    }

}
```

#12 Queue<> C#

- Queue<> class in collection.Generic namespace is FIFO
- Syntax:

```
Queue<int> Q = new Queue<int>();
```

*Methods

1) Q.Enqueue(ele) : Adds an Element in Queue at back
2) Q.Dequeue() : Removes & returns the front element from Queue
3) Q.Peek() : Returns the front element from Queue
4) Q.Count() : Returns the no. of elements in Queue
5) Q.Contains(e) : checks if Queue contains specified element, if yes then return true

*Traversing : foreach loop can be used for traversing

eg:

```
namespace Project1
{
    internal class Program
    {
        static void Main()
        {
            Queue<int> q = new Queue<int>();
            q.Enqueue(10);
            q.Enqueue(20);
            q.Enqueue(30);
            q.Dequeue();
            Cwl(q.Count());
            if(q.Contains(10))
            {
                Cwl("10 is there");
            }

            foreach(var i in q) // traversing queue
            {
                Cwl(i);
            }
        }
    }
}
```