Methods of array don't work for multidimensional array.

jagged array

```
int[][] arr = new int[4][];
```

* Collections
  1) → Regular collections
  2) → Generic collections

Generics

```
class MyClass<T>          T → data type
{
    void swap (ref T i, ref T j)
        {
            //swap
        }
}

Main()
{   Myclass<int> o1 = new Myclass<int>();
```

Constraints

```
class MyStack<T>
    where T : class   //T must be a
                        reference type
where T : struct  //T must be value type
    : ClassName   //T must be the
                    class specified or its
                    derived class
```

where T : InterfaceName // T must be a class that implements the specified interface

where T : new() // must have no parameter constructor

where T : ClassName, InterfaceName, new()

int? a = 100;

// Nullable<int> b = 100; using generics internally

Collection → Non Generic collections

Dynamic memory allocation, growable

using System.Collections;

IEnumerable

ICollection

Stack

Queue

IList

ArrayList

(Key, value)

IDictionary

HashTable

SortedList

Key - must be unique

value → you want to stored.

Add, Remove, Count → Ting

Ting

Ting

Read doning

ArrayList

- Add
- Remove (obj)
- RemoveAt(index)
- AddRange (obj2) — add at end on list
- Insert (0, "abc")
- InsertRange (index, Icollection)
- RemoveRange (index, count)
- Contains
- IndexOf
- LastIndexOf
- BinarySearch
- CopyTo (ar) — from arraylist to array
- GetRange, SetRange
- Reverse
- Sort
- ToArray
- Capacity
- TrimToSize → remove excess capacity

(used in database)

Capacity → block of memory instead of giving one at a time.

23/07/2022 Add remove count

Day 6

IDictionary
Hashtable (K,V)

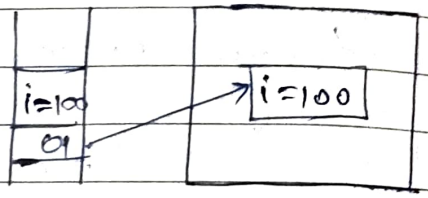
```
Hashtable o = new Hashtable();
o.Add(1, "Amey");
o.Add(1, "Akshay"); // error duplicate key

o[2] = "new";   // o[key]
o[1] = "Amey Patekar";  override the key
                        didn't give error

foreach (DictionaryEnty item in o)
{
    cw(item.key);
    cw(item.value);     //output is not in
}                         ordered

SortedList o = new SortedList();

o.Remove(1);    // 1 is key
o.RemoveAt(1)   // remove 1st index value

o.contains      } both are same
o.containskey   }
o.containsvalue
  CopyTo        convert into/copy into array
  GetByIndex    gets value
  Getkey
IList l=o.GetkeyList()
  GetvalueList
  IndexofKey
  IndexofValue
  Keys    → returns collection
  values  → returns collection
```

setByIndex(0, "aa")

Stack	Queue
Push	Enqueue
Pop	dequeue
Peek	peek

Boxing

```
object o1;
value type  int i = 100;
ref type →  o1 = 100;
```



converting value type into reference type is boxing
reading something from heap cause more time
avoid boxing

```
int j;
j = (int) o1;   unboxing
```

If you have box value type & reading value from box then it is called unboxing

Avoid boxing & unboxing

Generic collections is faster than non generic collection
non Generic collections use boxing
* Generic collections donot use boxing so its faster

Generic collections

IEnumerable<T>

ICollection<T>
stack<T>
Queue<T>

IList<T>
List<T>

IDictionary<Tkey, Tvalue>
SortedList HashTable<Tkey, Tvalue>
Dictionary<Tkey, Tvalue>

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

SortedList<int, string> o = new SortedList<int, string>();
        o.Add(1, "Amey");
foreach ( KeyValuePair<int, string> item in o)
    {
        cw(item.key)
        cw(item.value)
    }
```

List<T> is most frequently use class

```
List<Employee> lemp = new List<Employee>();

lemp.Add(new Employee { EmpNo=1, Name="Amey" });

foreach ( Employee item in lemp)
    {
        cw (item.EmpNo);
        cw (item.Name);
    }
```

Delegates
are like function pointers in C++

to delegate → assign task to someone else

delegates used in eventhandling, synchronous query

The delegate is a class that inherits from system.delegate class

public delegate delegateName
Also have signature of function that need to call function need to create object of delegate class

object
Delegate

Multicast Delegate → It can point to more than one function
Del1 same time (no limit)

TO DO ⇒ try add & substract

Action can call upto 16 parameters
Action can call for functions having void return type.

Func <> upto 16 parameter & return type
last is always return type.
non void return type
Predicate <> only one parameter
return type boolean.

Anonymous Methods
without method name.
don't forget to give semicolon

Lambda

```
Func<int,int> o2 = a => a*2;
    cw(o2(10));
```

o/p → 20

```
Func <int,int) o2 = () => 10;
    cw(o2()); //10
```

for zero parameters or other than 1 give parenthesis

```
Func<int,int,int> o3 = (a,b) => a+b;
```