Stack

➢ *Discussion topics:*

1. Definition

A **Stack** is a **linear data structure** that follows the **LIFO** principle:

- A **linear data structure** means that the elements are **arranged in a straight line**, one after another.
- **Last In, First Out** — the last item you add is the first one to be removed.

A **generic stack** means it can hold any type of data (integers, strings, custom objects, etc.), not just one specific type.

2. The Main Difference Between Stack and…

The key difference is **how elements are added and removed** — Stack is LIFO, Queue is FIFO, and Array/List allows random access.

Data Structure	Main Principle	Access
Stack	LIFO	Top only (push/pop/peek)
Queue	FIFO (First In, First Out)	Front & rear
List/Array	Indexed	Any element by index

Data Structure	Access
Stack	Top only — You can only add (push), view (peek), or remove (pop) the **last item you added** (the top).
Queue	Front and Rear — You add items from the **rear (end)**, and remove from the **front (start)**.
Array/List	Any element — You can access **any position** using an index like arr[0], arr[5], etc.

3. Key Operations of a Stack

1) **Push:** Adds an item to the top of the stack.
2) **Pop:** Removes and returns the item from the top of the stack.
3) **Peek:** Returns the item at the top of the stack without removing it.
4) **Count:** Returns the number of elements in the stack.

4. Examples from Real Life (Can't Be Solved Without Stack)

1) Browser History

- You go to page A → B → C
- Clicking "Back" goes from C → B → A (LIFO)

2) Undo Function in Text Editor

- Every change is stored on a stack
- Undo pops/remove the latest change

3) Function Calls

- Each function call is pushed onto a call stack
- Once done, it pops and returns

4) Reversing a Word or Sentence

- Push each character/word, then pop to reverse

What is a **Stack Overflow**?

A **stack overflow** happens when **too much data** is added to a stack — more than it can handle.

Why Does It Happen in Programming?

In programming, every program has a limited amount of memory for the **call stack** -> this is where **function calls** are stored.

If your program **keeps calling functions without stopping,** each call gets pushed onto the stack, until it **runs out of space**.

When that happens, the program throws a **"StackOverflowError"** or crashes.

EX:

```
void CallMe() {
   CallMe(); // keeps calling itself forever!
}
```

5. Creating and Using a Stack in C#

1) Import the namespace

```
using System.Collections.Generic; //allow us to use the built-in Stack<T> class
```

2) Create a Stack

```
Stack<int> numbers = new Stack<int>(); // Stack of integers
Stack<string> names = new Stack<string>(); // Stack of strings
//  <T> means it's generic, so you can store any type: int, string, object, etc.
```

3) Add items to the stack

```
numbers.Push(10);
numbers.Push(20);
numbers.Push(30);  // Stack now: [30, 20, 10] (top to bottom)
```

4) Remove top item from the stack

```
int top = numbers.Pop();  // Removes 30
Console.WriteLine("Popped: " + top);  // Output: Popped: 30
```

5) Remove all the items from the stack

```
    // Pop all items from the stack (removes them)
      Console.WriteLine("\nRemoving items:");
      while (numbers.Count > 0)
      {
         Console.WriteLine("Popped: " + numbers.Pop());
      }
//// ..... or ......
numbers. Clear();
```

6) View the top item of the stack

```
int topItem = numbers.Peek(); // Just looks at the top (20), doesn't remove it
Console.WriteLine("Top item: " + topItem);
```

7) View all items in the stack

```
// View all items in the stack (from top to bottom)
Console.WriteLine("Items in the stack:");
foreach (int num in numbers)
{
   Console.WriteLine(num);
}
```

8) Check if stack is empty

1. Count

```
if (numbers.Count == 0)
{
   Console.WriteLine("Stack is empty.");
}
// numbers.Count return int
```

2. Any()

```
numbers.Any() //return true and false

//you need to add using System.Linq; to use Any()
//Any() is more fast than count
```

9) Checks if the stack has a specific item

```
numbers.Contains(20); //return true or false
```

10) Converts the stack to an array

```
int[] arr = numbers.ToArray();
```

11) creates a **shallow copy**

```
Stack<int> clonedStack = new Stack<int>(numbers.Reverse()); // To clone a generic stack
Stack N = (Stack)numbers.Clone(); // To clone a non-generic Stack

/*
*A shallow copy means that the new stack N contains references to the same objects as
*the original stack numbers. If the stack holds value types (like integers), the values are
copied. However, if it holds reference types (like objects), both stacks will reference
the same objects.
*/
```