1. What is a Stack?

A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle.
The last inserted element is removed first.

Stack in Java (Collection Framework)

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

Important Stack Methods

Method	Meaning
`push(x)`	Add an element
`pop()`	Remove + return the top element
`peek()`	Return top element without removing
`isEmpty()`	Check if stack is empty
`size()`	Returns count of elements
`search(x)`	Finds 1-based position from top

Sample Code – Basic Stack Operations

```
import java.util.Stack;

public class StackBasics {
    public static void main(String[] args) {

        Stack<Integer> st = new Stack<>();

        st.push(10);
        st.push(20);
        st.push(30);

        System.out.println("Stack = " + st);
        System.out.println("Top = " + st.peek());
        System.out.println("Popped = " + st.pop());
        System.out.println("Size = " + st.size());
        System.out.println("Is Empty = " + st.isEmpty());
    }
}
```

IMPORTANT STACK PROBLEMS

Problem 1: Push Numbers in Stack & Print All

Input: push 1, 2, 3
Output: 3 2 1

```
Stack<Integer> st = new Stack<>();
st.push(1);
st.push(2);
st.push(3);

while(!st.isEmpty())
    System.out.print(st.pop() + " ");
```

Problem 2: Reverse a String

```
String s = "hello";
Stack<Character> st = new Stack<>();

for(char c : s.toCharArray()) st.push(c);

String rev = "";
while(!st.isEmpty()) rev += st.pop();

System.out.println(rev);
```

Problem 3: Find Minimum Element in Stack

```
Stack<Integer> st = new Stack<>();
st.push(5);
st.push(1);
st.push(3);

int min = Integer.MAX_VALUE;

for(int x : st)
    min = Math.min(min, x);

System.out.println("Min = " + min);
```

Problem 4: Check if Stack is Sorted (top → bottom)

```
static boolean isSorted(Stack<Integer> st) {
    int prev = Integer.MAX_VALUE;

    for(int x : st) {
        if(x > prev) return false;
        prev = x;
    }
    return true;
}
```

Problem 5: Count Elements in Stack

```
System.out.println("Count = " + st.size());
```

Problem 6: Remove All Even Numbers

```
Stack<Integer> st = new Stack<>();
Stack<Integer> temp = new Stack<>();

for(int i : st)
    if(i % 2 != 0)
        temp.push(i);
```

```
st = temp;
```

Problem 7: Display Middle Element of Stack

```
int mid = st.size() / 2;
System.out.println("Middle = " + st.get(mid));
```

Problem 8: Sum of All Elements in Stack

```
int sum = 0;
for(int x : st) sum += x;

System.out.println(sum);
```

Problem 9: Duplicate the Entire Stack

Example: `[1,2,3] → [1,2,3,1,2,3]`

```
Stack<Integer> newStack = new Stack<>();

newStack.addAll(st);
newStack.addAll(st);
```

Problem 10: Print Stack from Bottom to Top (without removing)

```
for(int i = 0; i < st.size(); i++)
    System.out.print(st.get(i) + " ");
```

STACK IMPLEMENTATION IN JAVA

Stack Using Array

```
class ArrayStack {

    int top;
    int[] arr;

    ArrayStack(int size) {
        arr = new int[size];
        top = -1;
    }

    void push(int x) {
        if(top == arr.length - 1) {
            System.out.println("Overflow");
            return;
        }
        arr[++top] = x;
    }

    int pop() {
        if(top == -1) {
            System.out.println("Underflow");
            return -1;
        }
        return arr[top--];
    }
```

```
    int peek() {
        return (top == -1) ? -1 : arr[top];
    }

    boolean isEmpty() {
        return top == -1;
    }
}
```

Stack Using Linked List

```
class Node {
    int data;
    Node next;
    Node(int d) { data = d; }
}

class LinkedStack {

    Node top;

    void push(int x) {
        Node n = new Node(x);
        n.next = top;
        top = n;
    }

    int pop() {
        if(top == null) {
            System.out.println("Underflow");
            return -1;
        }
        int val = top.data;
        top = top.next;
        return val;
    }

    int peek() {
        return (top == null) ? -1 : top.data;
    }

    boolean isEmpty() {
        return top == null;
    }
}
```

What is a Queue?

A **Queue** is a linear data structure that follows the
FIFO (First In First Out) principle.
The element inserted first will be removed first.

Real-life example:
People standing in a line → the person at the front goes first.

Queue in Java (Collection Framework)

Java does not have a direct `Queue` class, but it provides the `Queue` interface.

Mostly used implementations:

```
Queue<Integer> q = new LinkedList<>();
Queue<Integer> pq = new PriorityQueue<>(); // min-heap queue
```

For normal FIFO operations → **LinkedList** is used as Queue.

3. Important Queue Operations

Operation	Meaning
`add(x)` / `offer(x)`	Insert element at rear
`remove()` / `poll()`	Remove + return front element
`peek()`	Return front element without removing
`isEmpty()`	Check if queue is empty
`size()`	Count of elements
`contains(x)`	Check if element exists

Sample Code – Basic Queue Operations

```
import java.util.*;

public class QueueBasics {
    public static void main(String[] args) {

        Queue<Integer> q = new LinkedList<>();

        q.add(10);
        q.add(20);
        q.add(30);

        System.out.println("Queue = " + q);
        System.out.println("Front = " + q.peek());
        System.out.println("Removed = " + q.poll());
        System.out.println("Size = " + q.size());
        System.out.println("Is Empty = " + q.isEmpty());
    }
}
```

IMPORTANT QUEUE PROBLEMS

Problem 1: Print Queue Elements

```
Queue<Integer> q = new LinkedList<>();
q.add(1);
q.add(2);
```

```
q.add(3);

for(int x : q)
    System.out.print(x + " ");
```

Problem 2: Reverse a Queue (using Stack)

```
Queue<Integer> q = new LinkedList<>();
Stack<Integer> st = new Stack<>();

while(!q.isEmpty()) st.push(q.poll());
while(!st.isEmpty()) q.add(st.pop());
```

Problem 3: Find Maximum Element in Queue

```
int max = Integer.MIN_VALUE;
for(int x : q)
    max = Math.max(max, x);

System.out.println("Maximum = " + max);
```

Problem 4: Check if Queue is Sorted

(sorted from front to rear)

```
static boolean isSorted(Queue<Integer> q) {
    int prev = Integer.MIN_VALUE;

    for(int x : q) {
        if(x < prev) return false;
        prev = x;
    }
    return true;
}
```

Problem 5: Copy Queue to Another (Same Order)

```
Queue<Integer> q2 = new LinkedList<>();
q2.addAll(q);
```

Problem 6: Sum of Elements in Queue

```
int sum = 0;
for(int x : q) sum += x;

System.out.println(sum);
```

Problem 7: Find Middle Element of Queue

```
int mid = q.size() / 2;
int idx = 0;

for(int x : q) {
    if(idx == mid) {
        System.out.println("Middle = " + x);
        break;
    }
    idx++;
}
```

Problem 8: Remove All Even Numbers

```
Queue<Integer> temp = new LinkedList<>();

for(int x : q)
    if(x % 2 != 0)
        temp.add(x);

q = temp;
```

Problem 9: Rotate Queue by K positions

Move front elements to rear `k` times.

```
int k = 2;
while(k-- > 0) {
    q.add(q.poll());
}
```

Problem 10: Display Queue Without Removing Elements

```
for(int x : q)
    System.out.print(x + " ");
```

QUEUE IMPLEMENTATION USING ARRAY

This is called **Linear Queue using Array**.

```
class ArrayQueue {

    int front, rear;
    int[] arr;

    ArrayQueue(int size) {
        arr = new int[size];
        front = 0;
        rear = -1;
    }

    void enqueue(int x) {
        if(rear == arr.length - 1) {
            System.out.println("Overflow");
            return;
        }
        arr[++rear] = x;
    }

    int dequeue() {
        if(front > rear) {
            System.out.println("Underflow");
            return -1;
        }
        return arr[front++];
    }

    int peek() {
        if(front > rear) return -1;
        return arr[front];
    }

    boolean isEmpty() {
```

```
        return front > rear;
    }
}
```

7. CIRCULAR QUEUE USING ARRAY

Better performance (no wasted space).

```
class CircularQueue {

    int front, rear, size;
    int[] arr;

    CircularQueue(int n) {
        arr = new int[n];
        front = rear = -1;
        size = n;
    }

    void enqueue(int x) {
        if((rear + 1) % size == front) {
            System.out.println("Overflow");
            return;
        }

        if(front == -1) front = 0;

        rear = (rear + 1) % size;
        arr[rear] = x;
    }

    int dequeue() {
        if(front == -1) {
            System.out.println("Underflow");
            return -1;
        }

        int val = arr[front];

        if(front == rear) front = rear = -1;
        else front = (front + 1) % size;

        return val;
    }

    int peek() {
        return (front == -1) ? -1 : arr[front];
    }

    boolean isEmpty() {
        return front == -1;
    }
}
```

QUEUE IMPLEMENTATION USING LINKED LIST

```
class Node {
    int data;
    Node next;
    Node(int d) { data = d; }
```

```
}

class LinkedQueue {

    Node front, rear;

    void enqueue(int x) {
        Node n = new Node(x);

        if(rear == null) {
            front = rear = n;
            return;
        }

        rear.next = n;
        rear = n;
    }

    int dequeue() {
        if(front == null) {
            System.out.println("Underflow");
            return -1;
        }

        int val = front.data;
        front = front.next;

        if(front == null) rear = null;

        return val;
    }

    int peek() {
        return (front == null) ? -1 : front.data;
    }

    boolean isEmpty() {
        return front == null;
    }
}
```

What is Recursion?

Recursion is a programming technique where a method **calls itself** to solve a problem.

A recursive method always has:

1. **Base Case** → condition to stop recursion
2. **Recursive Case** → method calling itself again

Structure of a Recursive Function

```
void recursiveFunction() {
    // Base case: stop condition
    if(conditionIsTrue) {
        return;
    }

    // Recursive case: function calling itself
```

```
    recursiveFunction();
}
```

Why Do We Use Recursion?

Because some problems are naturally recursive and easier to solve using recursion:

- Solving mathematical functions (factorial, power)
- Tree traversals
- Graph traversals
- Searching and sorting algorithms (Merge Sort, Quick Sort)
- Backtracking problems
  (Sudoku, N-Queens, Maze solving)

How Recursion Works Internally

Recursion uses the **call stack**:

- Each recursive call is stored in the stack
- When the base case is hit, functions start returning back (unwinding)

If base case is missing → **StackOverflowError**

Examples of Recursion in Java

Factorial of a Number

$n! = n \times (n - 1)!$

Good example of base + recursive case.

```
int factorial(int n) {
    if(n == 0) {          // base case
        return 1;
    }
    return n * factorial(n - 1);   // recursive call
}
```

Sum of First N Natural Numbers

```
int sum(int n) {
    if(n == 1) {     // base case
        return 1;
    }
    return n + sum(n - 1);   // recursive call
}
```

Fibonacci Series Using Recursion

```
int fibonacci(int n) {
    if(n == 0 || n == 1) {          // base case
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Power Function (x^n)

```
int power(int x, int n) {
    if(n == 0) {     // base case
        return 1;
    }
    return x * power(x, n - 1);
}
```

Applications of Recursion

- ✓ Factorial, Fibonacci, Power, GCD, Sum of numbers
- ✓ Inorder, Preorder, Postorder, DFS
- ✓ Merge sort
- ✓ Quick sort
- ✓ Binary search

Advantages of Recursion

- ✓ Code becomes short and clean
- ✓ Best for tree and graph problems
- ✓ Natural fit for divide-and-conquer

Disadvantages of Recursion

- ✓ More memory usage (stack)
- ✓ Slow if too many calls (e.g., Fibonacci)
- ✓ Risk of StackOverflow if base case missing

When to Use Recursion

Use recursion when:

- Problem is naturally recursive
- Subproblems are smaller versions of the same problem
- Code becomes simpler than iterative method

Avoid recursion when:

- Performance is very critical
- Problem can be solved easily using loops