Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack will be the first to be removed. Imagine a stack of plates: the last plate you place on top is the first one you take off.

Key Points about Stack

1. **LIFO Principle**: Last In, First Out.
2. **Basic Operations**:
    - **Push**: Adds an element to the top of the stack.
    - **Pop**: Removes the element from the top of the stack.
    - **Peek**: Returns the top element without removing it.
    - **isEmpty**: Checks if the stack is empty.
3. **Applications**:
    - Expression evaluation and syntax parsing.
    - Backtracking algorithms (like maze solving).
    - Undo operations in applications.
    - Function call management in recursion.

Example of a Stack

Imagine you're reading through books:

1. You put Book 1 on the table, followed by Book 2, then Book 3.
2. Now, to read, you start with Book 3 (the last one placed).
3. You remove Book 3 (pop), then read Book 2, and finally Book 1.

Implementation of Stack in Java

Java has a built-in `Stack` class in `java.util` package, but we can also implement it manually using an array or linked list.

Stack Implementation Using Array

```
java
Copy code
class Stack {
    private int maxSize;
    private int[] stackArray;
    private int top;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1; // No elements initially
    }

    // Push operation
    public void push(int value) {
        if (top == maxSize - 1) {
            System.out.println("Stack overflow!");
        } else {
            stackArray[++top] = value;
        }
    }

    // Pop operation
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack underflow!");
            return -1;
        } else {
            return stackArray[top--];
        }
    }
```

```
    // Peek operation
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return -1;
        } else {
            return stackArray[top];
        }
    }

    // Check if the stack is empty
    public boolean isEmpty() {
        return top == -1;
    }
}

// Using the stack
public class StackExample {
    public static void main(String[] args) {
        Stack stack = new Stack(5);
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Top element: " + stack.peek());
        System.out.println("Popped: " + stack.pop());
        System.out.println("Is stack empty? " + stack.isEmpty
());
    }
}
```

Interview Questions

1. Explain the LIFO principle with a real-life example.

Answer:

The LIFO (Last In, First Out) principle means that the last element added to a stack is the first one to be removed. A real-life example of LIFO is a stack of plates. When you add plates to the stack, each new plate goes on top. To take a plate, you start from the top, removing the last plate you added first.

---

2. Describe the differences between a stack and a queue.

Answer:

- **Stack** follows the **LIFO** (Last In, First Out) principle, where the last element added is the first to be removed.
- **Queue** follows the **FIFO** (First In, First Out) principle, where the first element added is the first to be removed.
- Common uses:
  - **Stack**: Used in backtracking, function call management, and expression evaluation.
  - **Queue**: Used in scheduling processes, handling requests in order, and breadth-first search in graphs.

---

3. Implement a stack using a linked list in Java.

Answer:

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class LinkedListStack {
    private Node top;

    public LinkedListStack() {
        top = null;
    }

    // Push operation
    public void push(int value) {
        Node newNode = new Node(value);
        newNode.next = top;
        top = newNode;
    }

    // Pop operation
    public int pop() {
        if (top == null) {
            System.out.println("Stack underflow!");
            return -1;
        }
        int value = top.data;
        top = top.next;
        return value;
    }

    // Peek operation
    public int peek() {
        if (top == null) {
            System.out.println("Stack is empty!");
            return -1;
        }
        return top.data;
    }

    // Check if the stack is empty
    public boolean isEmpty() {
```

```
        return top == null;
    }
}

// Using the stack
public class LinkedListStackExample {
    public static void main(String[] args) {
        LinkedListStack stack = new LinkedListStack();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Top element: " + stack.peek());
        System.out.println("Popped: " + stack.pop());
        System.out.println("Is stack empty? " + stack.isEmpty
());
    }
}
```

4. How would you reverse a string using a stack?

Answer:
To reverse a string using a stack:

1. Push each character of the string onto the stack.
2. Pop each character from the stack and append it to a new string.

Here's how it would look in Java:

```
java
Copy code
import java.util.Stack;

public class ReverseString {
    public static String reverse(String str) {
        Stack<Character> stack = new Stack<>();
```

```
        for (char ch : str.toCharArray()) {
            stack.push(ch);
        }

        StringBuilder reversed = new StringBuilder();
        while (!stack.isEmpty()) {
            reversed.append(stack.pop());
        }

        return reversed.toString();
    }

    public static void main(String[] args) {
        String original = "hello";
        String reversed = reverse(original);
        System.out.println("Original: " + original);
        System.out.println("Reversed: " + reversed);
    }
}
```

5. How can you check for balanced parentheses in an expression using a stack?

Answer:
Balanced parentheses mean every opening bracket has a matching closing bracket in the correct order. Use a stack to:

1. Traverse each character in the string.
2. Push opening brackets (`(`, `{`, `[`) onto the stack.
3. When encountering a closing bracket (`)`, `}`, `]`), check if the top of the stack matches it. If it matches, pop the stack; if it doesn't, or the stack is empty, the parentheses are unbalanced.
4. At the end, if the stack is empty, the parentheses are balanced.

```
java
Copy code
import java.util.Stack;

public class BalancedParentheses {
    public static boolean isBalanced(String expression) {
        Stack<Character> stack = new Stack<>();

        for (char ch : expression.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty()) {
                    return false;
                }
                char top = stack.pop();
                if (!isMatchingPair(top, ch)) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }

    private static boolean isMatchingPair(char open, char clo
se) {
        return (open == '(' && close == ')') ||
               (open == '{' && close == '}') ||
               (open == '[' && close == ']');
    }

    public static void main(String[] args) {
        String expression = "{[()]}";
        System.out.println("Is balanced: " + isBalanced(expre
ssion));
```

```
    }
}
```

6. Explain the function of the call stack in recursion.

Answer:

In recursion, each function call is placed on the call stack. The **call stack** helps to:

1. **Store the state of each function** call, including variables and the point to return after the call completes.
2. **Track the order of calls** in a LIFO manner, so each function call is executed and returned in the reverse order of how they were called.

For example, in calculating a factorial, each call (e.g., `factorial(3)`) depends on a previous call (e.g., `factorial(2)`). The call stack stores these until it reaches the base case and then unwinds to return results in the reverse order.