Stack

```
class Stack {
    private int[] elements;
    private int top;
    private static final int DEFAULT_CAPACITY = 10;

    public Stack() {
        elements = new int[DEFAULT_CAPACITY];
        top = -1; // Stack is initially empty
    }

    // Method to push an element to the stack
    public void push(int element) {
        if (top == elements.length - 1) {
            expandCapacity(); // Expand stack capacity if needed
        }
        elements[++top] = element; // Place element at the next available position and increment 'top'
    }

    // Method to pop the top element from the stack
    public int pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Pop operation attempted on an empty stack.");
        }
        int result = elements[top--]; // Retrieve the top element and Decrement 'top'
        return result;
    }
```

```
// Method to peek at the top element of the stack without removing it
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Peek operation attempted on an empty stack.");
    }
        • return elements[top]; // Return the top element
}

// Helper method to check if the stack is empty
public boolean isEmpty() {
    return top == -1;
}

// Helper method to expand the capacity of the stack
private void expandCapacity() {
    int[] newStack = new int[elements.length * 2];
    System.arraycopy(elements, 0, newStack, 0, elements.length);
    elements = newStack;
}

public static void main(String[] args) {
    Stack stack = new Stack();

    // Demonstrating pushing elements
    stack.push(10);
    stack.push(20);
    stack.push(30);
    System.out.println("Pushed elements: 10, 20, 30");
```

```
        // Demonstrating peeking the top element
        System.out.println("Peeked element: " + stack.peek()); // Should show 30

        // Demonstrating popping elements
        System.out.println("Popped element: " + stack.pop()); // Should pop 30
        System.out.println("Popped element: " + stack.pop()); // Should pop 20

        // Checking the final top element
        System.out.println("Final top element after pops: " + stack.peek()); // Should show 10
    }
}
```

Output

Pushed elements: 10, 20, 30

Peeked element: 30

Popped element: 30

Popped element: 20

Final top element after pops: 10

Java.util.stack

```
import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        // Create a new stack
        Stack<Integer> stack = new Stack<>();
        // Check if the stack is empty
        System.out.println("Is the stack empty? " + stack.isEmpty());
```

```
    // Push elements onto the stack
    stack.push(10);
    System.out.println("Push 10");
    stack.push(20);
    System.out.println("Push 20");

    // Peek at the top element of the stack
    System.out.println("Top element (peek): " + stack.peek());

    // Pop elements from the stack
    System.out.println("Pop: " + stack.pop());
    System.out.println("Pop: " + stack.pop());

    // Check if the stack is empty after popping elements
    System.out.println("Is the stack empty now? " + stack.isEmpty());

    // Attempt to peek or pop when the stack is empty
    try {
        System.out.println("Trying to peek: " + stack.peek());
    } catch (Exception e) {
        System.out.println("Caught exception on peek: " + e.getMessage());
    }

    try {
        System.out.println("Trying to pop: " + stack.pop());
    } catch (Exception e) {
        System.out.println("Caught exception on pop: " + e.getMessage());
    }
}
```

```
}
```

Output

```
Is the stack empty? true
Push 10
Push 20
Top element (peek): 20
Pop: 20
Pop: 10
Is the stack empty now? true
Caught exception on peek: java.util.EmptyStackException
Caught exception on pop: java.util.EmptyStackException
```

Balanced Parenthesis

```
import java.util.ArrayDeque;
import java.util.Deque;

public class BalancedParentheses {
    public static boolean isBalanced(String expression) {
        Deque<Character> stack = new ArrayDeque<>();
        for (char ch : expression.toCharArray()) {
            switch (ch) {
                case '(', '{', '[':
                    stack.push(ch);
                    break;
                case ')', '}', ']':
                    if (stack.isEmpty() || !matches(stack.pop(), ch)) {
                        return false;
                    }
```

```
                break;
            }
        }
        return stack.isEmpty();
    }

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

    public static void main(String[] args) {
        String expression = "{[(2+3)*(1+1)]}";
        System.out.println("Expression is balanced: " + isBalanced(expression));
    }
}
```

Output

Expression is balanced: true

Evaluating PostFix Expression

```
import java.util.Stack;
public class PostfixEvaluation {

    public static int evaluatePostfix(String[] postfix) {
        Stack<Integer> stack = new Stack<>();
        for (String token : postfix) {
            if (Character.isDigit(token.charAt(0))) {  // Assuming single-digit operands for simplicity
                stack.push(Integer.parseInt(token));
```

```
            } else {
                int rightOperand = stack.pop();
                int leftOperand = stack.pop();
                int result = switch (token) {
                    case "+" -> leftOperand + rightOperand;
                    case "-" -> leftOperand - rightOperand;
                    case "*" -> leftOperand * rightOperand;
                    case "/" -> leftOperand / rightOperand;
                    default -> throw new IllegalArgumentException("Unexpected operator: " + token);
                };
                stack.push(result);
            }
        }
        return stack.pop();
    }
    public static void main(String[] args) {
        String[] postfix = {"3", "4", "2", "*", "1", "5", "-", "2", "3", "^", "^", "/", "+"};
        System.out.println("Result of the expression: " + evaluatePostfix(postfix));
    }
}
```

Output

Result of the expression: 3

Function Call Management

```
import java.util.ArrayDeque;
import java.util.Deque;

public class FunctionCallSimulator {
    private static void functionA() {
```

```
        System.out.println("Entering function A");
        callStack.push("functionA");
        functionB();
        callStack.pop();
        System.out.println("Exiting function A");
    }

    private static void functionB() {
        System.out.println("Entering function B");
        callStack.push("functionB");
        // Simulating some operations
        callStack.pop();
        System.out.println("Exiting function B");
    }

    private static Deque<String> callStack = new ArrayDeque<>();
    public static void main(String[] args) {
        callStack.push("main");
        System.out.println("Start of program");
        functionA();
        callStack.pop();
        System.out.println("End of program");
    }
}
```

Output

```
Start of program
Entering function A
Entering function B
```

```
Exiting function B
Exiting function A
End of program
```

Java.util.Vector

```
import java.util.Vector;


public class VectorStack {
    private Vector<Integer> stack;


    public VectorStack() {
        // Initialize the Vector with a default capacity
        stack = new Vector<>(2);  // Start with a small capacity to demonstrate resizing
    }


    public void push(int element) {
        System.out.println("Pushing " + element + " onto the stack.");
        // Before adding, print the current capacity
        System.out.println("Current capacity: " + stack.capacity());
        stack.add(element);
        // After adding, check if the capacity has changed
        System.out.println("New capacity (if resized): " + stack.capacity());
    }


    public int pop() {
        if (stack.isEmpty()) {
            throw new RuntimeException("Stack underflow - cannot pop from an empty stack");
        }
        return stack.remove(stack.size() - 1);
```

```
    }
    public static void main(String[] args) {
        VectorStack vs = new VectorStack();
        // Push elements to force the vector to resize
        vs.push(1);
        vs.push(2);
        // The next push should trigger a resize if the initial capacity was reached
        vs.push(3);
        vs.push(4);
        vs.push(5);  // Continues to push more elements to potentially trigger another resize
    }
}
```

Output

```
Pushing 1 onto the stack.
Current capacity: 2
New capacity (if resized): 2
Pushing 2 onto the stack.
Current capacity: 2
New capacity (if resized): 2
Pushing 3 onto the stack.
Current capacity: 2
New capacity (if resized): 4
Pushing 4 onto the stack.
Current capacity: 4
New capacity (if resized): 4
Pushing 5 onto the stack.
Current capacity: 4
New capacity (if resized): 8
```