Alright, my coding companions! We've stacked (pun intended!) quite a bit of knowledge about organizing data so far. We've explored arrays, singly linked lists, and doubly linked lists. Now, let's look at another fundamental data structure: the **Stack**.

Imagine a stack of plates in a cafeteria, or a stack of books on your desk. When you add a new plate or book, where does it go? On *top* of the existing stack, right? And when you want to take a plate or book, where do you take it from? Also from the *top*!

This simple, intuitive behavior is the essence of a **Stack** in Data Structures and Algorithms.

Stacks: The Last-In, First-Out (LIFO) Principle

A **Stack** is a linear data structure that follows a particular order in which operations are performed. This order is known as **LIFO: Last-In, First-Out**.

Think of it like this: The last item you added to the stack is always the first one you can remove. It's similar to:

- A pile of clothes: You always take the top one off.
- A Pringles can: The last chip you put in (if you could!) would be the first one you'd pull out.
- A deck of cards: You draw from the top, and new cards are usually added to the top.

Key Characteristics of a Stack:

- **LIFO Principle:** This is the defining characteristic.
- **Single End Operations:** All additions and removals happen from one end, traditionally called the **"Top"** of the stack.
- **Fixed or Dynamic Capacity:** Depending on how it's implemented (using an array or a linked list), a stack might have a fixed maximum size or be able to grow dynamically.

The Basic Operations: What Can a Stack Do?

Stacks have a very limited, but very powerful, set of operations.

1. **push(item):** Adds an item to the top of the stack.
    - *Analogy:* Putting a new plate on top of the stack.
    - *Time Complexity:* O(1) (constant time).
2. **pop():** Removes and returns the item from the top of the stack.
    - *Analogy:* Taking the top plate off the stack.
    - *Time Complexity:* O(1) (constant time).

3. **peek() (or top()):** Returns the item at the top of the stack *without* removing it.
    - *Analogy:* Looking at the top plate without taking it off.
    - *Time Complexity:* O(1) (constant time).
4. **isEmpty():** Checks if the stack contains any items. Returns true if empty, false otherwise.
    - *Analogy:* Is there any plate left on the stack?
    - *Time Complexity:* O(1) (constant time).
5. **size():** Returns the number of items currently in the stack.
    - *Analogy:* How many plates are there in the stack?
    - *Time Complexity:* O(1) (constant time, if size is tracked).

Why Use Stacks? (Advantages)

- **Enforces LIFO Order:** Guarantees that elements are processed in a strict "last in, first out" manner, which is crucial for many algorithms.
- **Simple and Intuitive:** The concept is easy to grasp and implement.
- **Efficient Operations:** All primary operations (push, pop, peek, isEmpty, and size) are typically O(1) (constant time), making stacks very fast for their primary purpose.

The Trade-offs (Disadvantages)

- **Limited Access:** You can only access the top element. If you need to access an element in the middle or bottom, you usually have to pop all the elements above it first.
- **Fixed Size (if array-backed):** If implemented with a fixed-size array, you can run into a "Stack Overflow" error (when trying to push onto a full stack) or "Stack Underflow" error (when trying to pop from an empty stack).

How to Implement a Stack in Java

You can implement a Stack using two common underlying data structures:

Method 1: Array-Backed Stack

This approach uses a regular array to store the elements and an integer variable (often called top or pointer) to keep track of the index of the top element.

Key Idea:

- top will initially be -1 (indicating an empty stack).
- push: Increment top, then add the item at array[top]. We must check if the stack is full before pushing.
- pop: Get the item from array[top], then decrement top. We must check if the

stack is empty before popping.

- peek: Just return array[top]. Again, check for an empty stack.

Code:

```
public class ArrayStack {
    private int[] stackArray;
    private int top;        // Index of the top element (points to the last added item)
    private int capacity;    // Maximum size of the stack

    // Constructor: Creates a new ArrayStack with a specified maximum capacity.
    public ArrayStack(int capacity) {
        this.capacity = capacity;
        this.stackArray = new int[capacity]; // Initialize the underlying array
        this.top = -1; // -1 indicates the stack is empty (no elements yet)
        System.out.println("ArrayStack created with capacity: " + capacity);
    }

    // --- Basic Stack Operations ---

    // Adds an item to the top of the stack.
    // Time Complexity: O(1)
    public void push(int item) {
        if (isFull()) { // First, check if the stack has reached its maximum capacity.
            System.out.println("Stack Overflow! Cannot push " + item + ". Stack is full.");
            return; // Exit if full.
        }
        stackArray[++top] = item; // 1. Increment 'top' (move to the next available slot).
                                  // 2. Place the 'item' at the new 'top' index.
        System.out.println("Pushed: " + item + ". Top index: " + top);
    }

    // Removes and returns the item from the top of the stack.
    // Time Complexity: O(1)
    public int pop() {
        if (isEmpty()) { // First, check if the stack is empty.
            System.out.println("Stack Underflow! Cannot pop from an empty stack.");
            return -1; // Indicate failure (or you could throw an exception like
NoSuchElementException).
        }
```

```
        int poppedItem = stackArray[top--]; // 1. Retrieve the item at the current 'top' index.
                                            // 2. Decrement 'top' (conceptually removing the item).
        System.out.println("Popped: " + poppedItem + ". Top index: " + top);
        return poppedItem; // Return the removed item.
    }

    // Returns the item at the top of the stack without removing it.
    // Time Complexity: O(1)
    public int peek() {
        if (isEmpty()) { // Check if the stack is empty.
            System.out.println("Stack is empty. No item to peek.");
            return -1; // Indicate failure.
        }
        System.out.println("Peeked: " + stackArray[top]);
        return stackArray[top]; // Return the item at 'top'.
    }

    // Checks if the stack contains any items.
    // Time Complexity: O(1)
    public boolean isEmpty() {
        return top == -1; // If 'top' is -1, it means there are no elements.
    }

    // Checks if the stack is full (relevant for array-backed stacks).
    // Time Complexity: O(1)
    public boolean isFull() {
        return top == capacity - 1; // If 'top' is at the last valid index, the stack is full.
    }

    // Returns the current number of elements in the stack.
    // Time Complexity: O(1)
    public int size() {
        return top + 1; // Since 'top' is 0-indexed, size is top + 1.
    }

    // --- Utility Method for Visualization ---
    // Prints the elements of the stack from top to bottom.
    public void printStack() {
```

```
    if (isEmpty()) {
        System.out.println("Stack: (empty)");
        return;
    }
    System.out.print("Stack (Top to Bottom): ");
    for (int i = top; i >= 0; i--) { // Iterate from 'top' down to 0.
        System.out.print(stackArray[i] + " ");
    }
    System.out.println();
}

// --- Main Method for Testing ArrayStack ---
public static void main(String[] args) {
    ArrayStack stack = new ArrayStack(3); // Let's make a small stack with capacity 3

    System.out.println("--- Initial State ---");
    System.out.println("Is stack empty? " + stack.isEmpty()); // Expected: true
    System.out.println("Stack size: " + stack.size());       // Expected: 0
    stack.printStack();

    System.out.println("\n--- Pushing Elements ---");
    stack.push(10); // stack: [10]
    stack.printStack();
    stack.push(20); // stack: [20, 10]
    stack.printStack();
    stack.push(30); // stack: [30, 20, 10]
    stack.printStack();

    System.out.println("\n--- Stack Full Check ---");
    System.out.println("Is stack full? " + stack.isFull()); // Expected: true
    stack.push(40); // Expected: Stack Overflow!

    System.out.println("\n--- Peeking ---");
    stack.peek(); // Expected: 30

    System.out.println("\n--- Popping Elements ---");
    stack.pop(); // Expected: Popped 30. stack: [20, 10]
    stack.printStack();
    System.out.println("Current stack size: " + stack.size()); // Expected: 2
```

```
        stack.pop(); // Expected: Popped 20. stack: [10]
        stack.printStack();

        stack.pop(); // Expected: Popped 10. stack: []
        stack.printStack();
        System.out.println("Is stack empty? " + stack.isEmpty()); // Expected: true

        System.out.println("\n--- Stack Empty Check ---");
        stack.pop(); // Expected: Stack Underflow!
        stack.peek(); // Expected: Stack is empty.
    }
}
```

Method 2: Linked List-Backed Stack

This approach uses a Singly Linked List (or a Doubly Linked List) to implement the stack. The "top" of the stack simply corresponds to the head of the underlying linked list.

Key Idea:

- push: Add a new node at the head of the linked list. This is effectively the addAtHead operation we learned for SLLs.
- pop: Remove the head node of the linked list. This is effectively the deleteFromHead operation.
- peek: Return the data of the head node.

Code:

```
// We'll define the Node class directly within this file for convenience.
class StackNode {
    int data;     // The actual value stored in this node.
    StackNode next; // Reference to the next node in the stack.

    // Constructor for StackNode.
    public StackNode(int data) {
        this.data = data;
        this.next = null; // Initially, points to nothing.
    }
```

```
}

public class LinkedListStack {
    private StackNode top; // Reference to the top node of the stack.
                           // This is essentially the 'head' of our linked list.
    private int size;      // Current number of elements in the stack.

    // Constructor: Initializes an empty LinkedListStack.
    public LinkedListStack() {
        this.top = null; // Stack starts empty.
        this.size = 0;   // Size is zero.
        System.out.println("LinkedListStack created.");
    }

    // --- Basic Stack Operations ---

    // Adds an item to the top of the stack.
    // Time Complexity: O(1) - Because we just update the 'top' pointer.
    public void push(int item) {
        StackNode newNode = new StackNode(item); // 1. Create a new StackNode.
        newNode.next = top; // 2. Make the new node's 'next' point to the current 'top'.
        top = newNode;      // 3. The new node now becomes the new 'top'.
        size++;             // 4. Increment the size.
        System.out.println("Pushed: " + item + ". Current size: " + size);
    }

    // Removes and returns the item from the top of the stack.
    // Time Complexity: O(1) - Because we just update the 'top' pointer.
    public int pop() {
        if (isEmpty()) { // 1. Check if the stack is empty.
            System.out.println("Stack Underflow! Cannot pop from an empty stack.");
            return -1; // Indicate failure.
        }
        int poppedItem = top.data; // 2. Get the data from the current 'top' node.
        top = top.next;            // 3. Move 'top' to the next node in the sequence.
                                   //    The old 'top' node is now unreferenced and eligible for
garbage collection.
        size--;                    // 4. Decrement the size.
        System.out.println("Popped: " + poppedItem + ". Current size: " + size);
```

```
    return poppedItem; // Return the removed item.
}

// Returns the item at the top of the stack without removing it.
// Time Complexity: O(1)
public int peek() {
    if (isEmpty()) { // Check if the stack is empty.
        System.out.println("Stack is empty. No item to peek.");
        return -1; // Indicate failure.
    }
    System.out.println("Peeked: " + top.data);
    return top.data; // Return the data from the 'top' node.
}

// Checks if the stack contains any items.
// Time Complexity: O(1)
public boolean isEmpty() {
    return top == null; // If 'top' is null, the stack is empty.
}

// Returns the current number of elements in the stack.
// Time Complexity: O(1)
public int size() {
    return size; // Return the maintained 'size' count.
}

// --- Utility Method for Visualization ---
// Prints the elements of the stack from top to bottom.
public void printStack() {
    if (isEmpty()) {
        System.out.println("Stack: (empty)");
        return;
    }
    System.out.print("Stack (Top to Bottom): ");
    StackNode current = top; // Start from the 'top' of the stack.
    while (current != null) { // Traverse the linked list until the end (null).
        System.out.print(current.data + " ");
        current = current.next; // Move to the next node.
    }
```

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

// --- Main Method for Testing LinkedListStack ---
public static void main(String[] args) {
    LinkedListStack stack = new LinkedListStack();

    System.out.println("--- Initial State ---");
    System.out.println("Is stack empty? " + stack.isEmpty()); // Expected: true
    System.out.println("Stack size: " + stack.size());       // Expected: 0
    stack.printStack();

    System.out.println("\n--- Pushing Elements ---");
    stack.push(10); // stack: [10]
    stack.printStack();
    stack.push(20); // stack: [20, 10]
    stack.printStack();
    stack.push(30); // stack: [30, 20, 10]
    stack.printStack();

    System.out.println("\n--- Peeking ---");
    stack.peek(); // Expected: 30

    System.out.println("\n--- Popping Elements ---");
    stack.pop(); // Expected: Popped 30. stack: [20, 10]
    stack.printStack();
    System.out.println("Current stack size: " + stack.size()); // Expected: 2

    stack.push(40); // stack: [40, 20, 10]
    stack.printStack();
    stack.push(50); // stack: [50, 40, 20, 10]
    stack.printStack();

    stack.pop(); // Expected: Popped 50.
    stack.pop(); // Expected: Popped 40.
    stack.pop(); // Expected: Popped 20.
    stack.pop(); // Expected: Popped 10.
    stack.printStack();
    System.out.println("Is stack empty? " + stack.isEmpty()); // Expected: true
```

```
        System.out.println("\n--- Stack Empty Check ---");
        stack.pop(); // Expected: Stack Underflow!
        stack.peek(); // Expected: Stack is empty.
    }
}
```

Java's Built-in Stack Class (and Deque - Recommended!)

While Java provides a java.util.Stack class, it's generally **not recommended** for new code. It extends Vector (an old, synchronized, dynamic array), which makes it less efficient for many use cases.

For modern Java, it's highly recommended to use the java.util.Deque (Double-ended Queue) interface when you need Stack (or Queue) functionality. ArrayDeque is a good, efficient, non-synchronized implementation of Deque that works well as a stack.

- **For Stack behavior (LIFO) with Deque:**
  - push(item): Adds an element to the front (which acts as the top of the stack).
  - pop(): Removes and returns an element from the front (top).
  - peek(): Returns an element from the front (top) without removing.

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.NoSuchElementException; // For handling empty stack pop/peek gracefully

public class JavaBuiltInStackDemo {
    public static void main(String[] args) {
        // Using ArrayDeque as a Stack (Recommended way in modern Java)
        // You declare it as Deque, and instantiate as ArrayDeque or LinkedList.
        Deque<String> browserHistory = new ArrayDeque<>(); // LIFO behavior from the front/end

        System.out.println("Is browser history empty? " + browserHistory.isEmpty()); // true

        System.out.println("\n--- Navigating (Pushing) ---");
        browserHistory.push("google.com");    // push to top
```

```
        System.out.println("Pushed: google.com. Current history: " + browserHistory);
        browserHistory.push("youtube.com");
        System.out.println("Pushed: youtube.com. Current history: " + browserHistory);
        browserHistory.push("javatpoint.com");
        System.out.println("Pushed: javatpoint.com. Current history: " + browserHistory);

        System.out.println("\n--- Peeking ---");
        System.out.println("Current page: " + browserHistory.peek()); // javatpoint.com
(top element)
        System.out.println("History size: " + browserHistory.size()); // 3

        System.out.println("\n--- Going Back (Popping) ---");
        String currentPage = browserHistory.pop(); // Remove javatpoint.com
        System.out.println("Went back from: " + currentPage);
        System.out.println("Now on: " + browserHistory.peek()); // youtube.com
        System.out.println("Current history: " + browserHistory);

        browserHistory.pop(); // Removes youtube.com
        System.out.println("Current history: " + browserHistory);
        browserHistory.pop(); // Removes google.com
        System.out.println("Current history: " + browserHistory);

        System.out.println("\n--- Empty Check ---");
        System.out.println("Is browser history empty? " + browserHistory.isEmpty()); //
true

        System.out.println("\n--- Trying to Pop/Peek from Empty Stack ---");
        try {
            browserHistory.pop(); // This will throw NoSuchElementException if empty!
        } catch (NoSuchElementException e) {
            System.out.println("Error: " + e.getMessage() + " - Cannot pop from empty
history!");
        }

        try {
            browserHistory.peek(); // This will return null if empty!
            System.out.println("Peeked from empty: " + browserHistory.peek()); // Will be
null
        } catch (Exception e) {
```

```
            System.out.println("Error peeking from empty: " + e.getMessage());
        }
    }
}
```

Common Use Cases and Advanced Stack Problems:

Stacks are surprisingly versatile and appear in many algorithms and real-world applications:

1. **Function Call Stack (Recursion):** When you call a method in a program, information about that method call (local variables, where to return to) is implicitly pushed onto a call stack by the operating system/JVM. When the method finishes, its information is popped. Recursion heavily relies on this implicit stack.
2. **Expression Evaluation:**
    - **Infix to Postfix/Prefix Conversion:** Stacks are used to convert mathematical expressions from human-readable (infix like A + B * C) to forms easier for computers to parse (postfix like A B C * + or prefix like + A * B C).
    - **Postfix/Prefix Expression Evaluation:** Stacks are used to evaluate expressions in postfix (Reverse Polish Notation) or prefix forms.
3. **Undo/Redo Functionality:** Most editors (text editors, image editors like Photoshop) use stacks to keep track of changes for "undo" and "redo" operations. Each action is pushed onto an "undo" stack. When you undo, the action is popped from the undo stack and pushed onto a "redo" stack.
4. **Browser History:** Navigating back and forth in a web browser can be modeled using two stacks (one for 'back' history and one for 'forward' history).
5. **Parentheses/Bracket Balancing:** Checking if parentheses, brackets, and braces in an expression or code are correctly matched and nested (e.g., ({[]}) is balanced, ([)] is not). You push opening brackets and pop/match with closing ones.
6. **Backtracking Algorithms:** Used in algorithms that involve exploring choices, and then "backtracking" if a path doesn't lead to a solution (e.g., solving mazes, Sudoku, N-Queens problem). When you hit a dead end, you pop back to the last decision point.
7. **Depth-First Search (DFS):** In graph and tree traversal, DFS can be implemented using a stack (explicitly or implicitly via recursion, which uses the call stack).

When to Choose a Stack?

- When you specifically need **LIFO** (Last-In, First-Out) behavior for adding and

removing elements.
- When you need very **fast (**O(1)**) additions and removals** from one end.
- For problems involving **backtracking**, **expression evaluation**, or situations where you need to **reverse the order** of elements.

When NOT to Choose a Stack?

- When you need **random access** to elements by index (e.g., "get the 5th element") – use an ArrayList or Array instead (O(1)).
- When you need **FIFO** (First-In, First-Out) behavior – use a Queue instead (O(1) operations).
- When you need to **insert or delete elements from the middle** efficiently – use a LinkedList instead.

Conclusion: Stack It Up!

You've now got a solid understanding of Stacks – their LIFO principle, core operations, and how to implement them from scratch using arrays or linked lists. You've also seen Java's built-in options and, importantly, when and why to use (or not use) a Stack.

Stacks are a deceptively simple yet incredibly powerful data structure, fundamental to understanding how computer programs work and how to solve many algorithmic challenges efficiently. Keep practicing those push and pop operations, and you'll master this essential tool!