**Stack and Queue in Java**

**Introduction**

A **Stack** is a data structure that follows the **Last In, First Out (LIFO)** principle. The element that is added last is the one to be removed first. In Java, the Stack class is part of the java.util package and extends the Vector class.

**Key Characteristics of a Stack**

1. **LIFO Behavior**: Last element added is the first to be removed.
2. **Operations**:
   * **Push**: Adds an element to the top of the stack.
   * **Pop**: Removes and returns the top element of the stack.
   * **Peek**: Returns the top element without removing it.
   * **Search**: Returns the 1-based position of an element from the top of the stack.
   * **isEmpty**: Checks if the stack is empty.
3. **Dynamic Size**: The size of a stack in Java can grow or shrink dynamically.

**Stack Class in Java**

The Stack class in Java is a generic class, meaning it can hold objects of any type.

**Declaration:**

Stack<Type> stack = new Stack<>();

**Example:**

import java.util.Stack;

public class Main {

public static void main(String[] args) {

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

// Push elements

stack.push(10);

stack.push(20);

stack.push(30);

// Peek top element

System.out.println("Top element: " + stack.peek());

// Pop element

System.out.println("Popped element: " + stack.pop());

// Search for an element

System.out.println("Position of 10: " + stack.search(10));

// Check if stack is empty

System.out.println("Is stack empty? " + stack.isEmpty());

}

}

**Methods of the Stack Class**

1. **push(E item)**
   * Adds an element to the top of the stack.
   * Returns the element added.
   * Throws NullPointerException if the element is null (in earlier versions).
2. **pop()**
   * Removes and returns the top element.
   * Throws EmptyStackException if the stack is empty.
3. **peek()**
   * Returns the top element without removing it.
   * Throws EmptyStackException if the stack is empty.
4. **isEmpty()**
   * Checks if the stack contains no elements.
   * Returns a boolean.
5. **search(Object o)**
   * Searches for the specified object in the stack.
   * Returns the 1-based position of the element from the top.
   * Returns -1 if the element is not found.

**Applications of Stack**

1. **Expression Evaluation**: Used in evaluating postfix or prefix expressions.
2. **Expression Conversion**: Converts infix expressions to postfix or prefix.
3. **Backtracking**: Used in algorithms like maze solving, navigating browser history.
4. **Function Call Stack**: Stores information about active subroutines in programs.
5. **Undo Mechanisms**: Implemented in text editors and other applications.
6. **Parsing**: Used in syntax parsing for compilers.

**Stack Implementation in Java**

Although the Stack class is part of the Java Collections Framework, its usage is less common due to alternatives like Deque. A stack can also be implemented manually using Deque or arrays.

**Implementation Using Deque:**

import java.util.Deque;

import java.util.ArrayDeque;

public class Main {

public static void main(String[] args) {

Deque<Integer> stack = new ArrayDeque<>();

// Push elements

stack.push(10);

stack.push(20);

stack.push(30);

// Peek top element

System.out.println("Top element: " + stack.peek());

// Pop element

System.out.println("Popped element: " + stack.pop());

// Check if stack is empty

System.out.println("Is stack empty? " + stack.isEmpty());

}

}

**Stack Limitations**

1. **Thread Safety**: Stack is synchronized, which may lead to performance issues.
2. **Legacy**: Considered a legacy class, newer classes like Deque are preferred for stack implementations.

**Best Practices**

1. Use Deque for stack implementations in modern applications.
2. Avoid using Stack for high-performance or multi-threaded applications.
3. Understand the underlying operations to optimize stack usage for specific applications.

**Comparison with Other Data Structures**

1. **Queue**: Operates on First In, First Out (FIFO) principle, opposite to LIFO.
2. **Deque**: Supports both stack and queue operations, making it more versatile.
3. **Vector**: Stack extends Vector, but is less commonly used for general data storage due to its LIFO nature.

**Frequently Asked Questions**

1. **Why use a stack over other data structures?**
   * For problems requiring LIFO behavior, such as recursion simulation or backtracking.
2. **Is Stack class outdated?**
   * Yes, it is considered legacy. Use Deque for modern applications.
3. **Can a stack overflow in Java?**
   * Yes, if recursion is too deep or memory limits are reached.

**Syntaxes for defining a stack or a queue in Java:**

**Stack**

1. **Using Array: int[] stack = new int[capacity];**
2. **Using Linked List: Node top = null; // where Node is a custom class**
3. **Using Built-in Stack Class: Stack<Integer> stack = new Stack<>();**
4. **Using Deque: Deque<Integer> stack = new ArrayDeque<>();**
5. **Using Queues (Two Queues): Queue<Integer> q1 = new LinkedList<>(), q2 = new LinkedList<>();**

**Queue**

1. **Using Array: int[] queue = new int[capacity];**
2. **Using Linked List: Node front = null, rear = null; // where Node is a custom class**
3. **Using Built-in Queue Interface: Queue<Integer> queue = new LinkedList<>();**
4. **Using Deque: Deque<Integer> queue = new ArrayDeque<>();**
5. **Using Stacks (Two Stacks): Stack<Integer> s1 = new Stack<>(), s2 = new Stack<>();**

**Detailed Notes on Patterns and Applications (Java Version)**

**I. Monotonic Stack Pattern (Detailed)**

* **Core Idea:** A monotonic stack maintains elements in either strictly increasing or strictly decreasing order. This allows for efficient processing of problems involving finding the next greater/smaller element.
* **Basic Template (Java):**

Java

import java.util.Stack;

import java.util.ArrayList;

import java.util.List;

List<Integer> stackPattern(int[] arr, int startIndex, java.util.function.BiPredicate<Integer, Integer> comparison) {

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

List<Integer> result = new ArrayList<>();

for (int i = startIndex; i >= 0 && i < arr.length; i = (startIndex > i ? i - 1 : i + 1)) {

while (!st.isEmpty() && comparison.test(st.peek(), arr[i])) {

st.pop();

}

result.add(st.isEmpty() ? -1 : st.peek());

st.push(arr[i]);

}

return result; // Reverse if needed

}

* **Key Variations (with detailed explanations, comparison logic, and Java implementations):**
  1. **Next Greater Element Right (NGER):**
     + **Direction:** Right-to-left traversal of the input array.
     + **Stack Order:** Decreasing.
     + **Comparison:** (stTop, current) -> stTop <= current
     + **Java Code:**

Java

List<Integer> nger(int[] arr) {

return stackPattern(arr, arr.length - 1, (stTop, current) -> stTop <= current);

}

* 1. **Next Greater Element Left (NGEL):**
     + **Direction:** Left-to-right traversal.
     + **Stack Order:** Decreasing.
     + **Comparison:** (stTop, current) -> stTop <= current
     + **Java Code:**

Java

List<Integer> ngel(int[] arr) {

return stackPattern(arr, 0, (stTop, current) -> stTop <= current);

}

* 1. **Next Smaller Element Right (NSER):**
     + **Direction:** Right-to-left traversal.
     + **Stack Order:** Increasing.
     + **Comparison:** (stTop, current) -> stTop >= current
     + **Java Code:**

Java

List<Integer> nser(int[] arr) {

return stackPattern(arr, arr.length - 1, (stTop, current) -> stTop >= current);

}

* 1. **Next Smaller Element Left (NSEL):**
     + **Direction:** Left-to-right traversal.
     + **Stack Order:** Increasing.
     + **Comparison:** (stTop, current) -> stTop >= current
     + **Java Code:**

List<Integer> nsel(int[] arr) { return stackPattern(arr, 0, (stTop, current) -> stTop >= current); }

**II. Parentheses Pattern (Detailed)**

* **Purpose:** Validating whether a string of parentheses is balanced (correctly nested).
* **Implementation (Java):**

Java

import java.util.Stack;

boolean parenthesesValidator(String s) {

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

for (char c : s.toCharArray()) {

if (c == '(' || c == '{' || c == '[') {

st.push(c);

} else {

if (st.isEmpty()) return false;

char top = st.pop();

if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {

return false;

}

}

}

return st.isEmpty();

}

* **Key Idea:** Use a stack to keep track of opening brackets. When a closing bracket is encountered, check if it matches the top of the stack.

**III. Advanced Stack Problems (Detailed)**

* **Min Stack Implementation:**
  + **Challenge:** Implement a stack that supports push, pop, top, and getMin in O(1) time.
  + **Solution (using a single stack and a variable for minElement):**

Java

class MinStack {

Stack<Long> st;

long minElement;

public MinStack() {

st = new Stack<>();

}

public void push(int val) {

if (st.isEmpty()) {

minElement = val;

st.push((long)val);

} else {

if (val < minElement) {

st.push(2L \* val - minElement);

minElement = val;

} else {

st.push((long)val);

}

}

}

public void pop() {

if (st.isEmpty()) return;

long val = st.pop();

if (val < minElement) {

minElement = 2 \* minElement - val;

}

}

public int top() {

if (st.isEmpty()) return -1;

long val = st.peek();

if (val < minElement) {

return (int)minElement;

}

return (int)val;

}

public int getMin() {

return (int)minElement;

}

}

* **Advanced Applications:** (Conceptual explanations remain the same, adapting to Java syntax would be straightforward)

**IV. Common Applications (Detailed)**

* **Stock Span Problem:**

Java

import java.util.Stack;

public int[] calculateSpan(int[] prices) {

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

int[] spans = new int[prices.length];

for (int i = 0; i < prices.length; i++) {

while (!st.isEmpty() && prices[st.peek()] <= prices[i]) {

st.pop();

}

spans[i] = st.isEmpty() ? i + 1 : i - st.peek();

st.push(i);

}

return spans;

}

* **Maximum Rectangle in Histogram:** (Implementation would follow the same logic as C++, using NSEL and NSER, now in Java)

I have focused on providing complete and working Java code examples for the core patterns and applications. If you have any further questions or need clarification on any specific part, feel free to ask!

**Stack Implementations**

1. **Using Array**:
   * A simple integer array is used to store stack elements.
   * A pointer top keeps track of the index of the last added element.
   * Operations:
     + **Push**: Increment top and place the element at stack[top].
     + **Pop**: Retrieve the element at stack[top] and decrement top.
     + **Peek**: Directly access stack[top].
   * **Limitations**: Fixed size, requiring resizing if capacity is exceeded.
2. **Using Linked List**:
   * A linked list node has two parts: data and a pointer to the next node.
   * The top node represents the stack's head.
   * Operations:
     + **Push**: Create a new node, set its next to the current top, and update top.
     + **Pop**: Remove the current top by pointing top to top.next.
     + **Peek**: Access the data of the top node.
   * **Advantages**: Dynamically resizable; no predefined size needed.
3. **Using Built-in Stack Class**:
   * Internally uses a Vector (dynamic array).
   * Operations like push and pop manipulate the end of the vector.
   * **Push**: Appends the element to the end of the underlying array.
   * **Pop**: Removes and returns the last element.
   * **Peek**: Retrieves the last element without removing it.
   * **Advantages**: Thread-safe but slightly slower due to synchronization.
4. **Using Deque**:
   * Internally uses a double-ended queue (like ArrayDeque).
   * Operations are restricted to one end (treated as a stack).
   * **Push**: Adds an element at the front using addFirst().
   * **Pop**: Removes and returns the front element using removeFirst().
   * **Peek**: Accesses the front element without removing it using peekFirst().
5. **Using Queues (Two Queues)**:
   * Two queues (q1 and q2) simulate stack behavior.
   * **Push**: Add the new element to q2. Move all elements from q1 to q2 to maintain stack order, then swap q1 and q2.
   * **Pop**: Remove the front element of q1.
   * **Peek**: Access the front element of q1.
   * **Drawback**: Push operation is slower (O(n)) due to element transfer.

**Queue Implementations**

1. **Using Array**:
   * A circular array is used to implement a queue.
   * Two pointers, front and rear, track the start and end of the queue.
   * Operations:
     + **Enqueue**: Add an element at queue[rear] and increment rear (circularly).
     + **Dequeue**: Access queue[front], increment front (circularly).
     + **Peek**: Directly access queue[front].
   * **Limitations**: Fixed size, resizing needed for dynamic behavior.
2. **Using Linked List**:
   * Each node in the linked list represents an element in the queue.
   * Two pointers, front and rear, manage the queue.
   * Operations:
     + **Enqueue**: Create a new node and link it to the rear; update rear.
     + **Dequeue**: Remove the node at front and update front to front.next.
     + **Peek**: Access the data of front.
   * **Advantages**: Dynamically resizable.
3. **Using Built-in Queue Interface**:
   * A LinkedList is used internally to implement the queue.
   * Operations:
     + **Enqueue**: Add elements using add() or offer() (at the tail).
     + **Dequeue**: Remove elements using remove() or poll() (from the head).
     + **Peek**: Retrieve the head element without removing it using peek().
   * **Advantages**: Easy to use with predefined methods.
4. **Using Deque**:
   * A Deque is treated as a regular queue by restricting operations to both ends.
   * Operations:
     + **Enqueue**: Add elements to the tail using addLast().
     + **Dequeue**: Remove elements from the head using removeFirst().
     + **Peek**: Access the head element without removing it using peekFirst().
   * **Advantages**: Efficient and versatile, supporting both stack and queue behavior.
5. **Using Stacks (Two Stacks)**:
   * Two stacks (s1 and s2) simulate queue behavior.
   * **Enqueue**: Push the element onto s1.
   * **Dequeue**:
     + If s2 is empty, transfer all elements from s1 to s2 (reversing the order).
     + Pop an element from s2.
   * **Peek**: Access the top element of s2 (or transfer and then access if s2 is empty).
   * **Drawback**: Dequeue operation is slower (O(n)) if s2 is empty.