

Chapter 3

■■ How It Works Internally

Memory, pointers, and what really happens under the hood

■■ Array-Based Stack Implementation

The simplest stack uses an array with a **top pointer** (an integer index) tracking the topmost element. Initially `top = -1` (empty).

Stack using Array

JAVA

```
class Stack {
    private int[] arr;
    private int top;
    private int capacity;

    Stack(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        top = -1; // -1 means empty
    }

    // Push: increment top, store value
    void push(int x) {
        if (top == capacity - 1) throw new RuntimeException("Stack Overflow");
        arr[++top] = x; // pre-increment top, then store
    }

    // Pop: return value, decrement top
    int pop() {
        if (top == -1) throw new RuntimeException("Stack Underflow");
        return arr[top--]; // return value, then post-decrement
    }

    int peek() { return arr[top]; }
    boolean isEmpty() { return top == -1; }
    int size() { return top + 1; }
```

```
}
```

■ Linked List-Based Stack

Dynamic size, no overflow (until heap is full). Each node holds data + pointer to the next node. The **head** acts as the top.

Stack using Linked List

JAVA

```
class StackLL {  
    static class Node {  
        int data;  
        Node next;  
        Node(int d) { data = d; }  
    }  
    private Node head; // head = top  
  
    void push(int x) {  
        Node node = new Node(x);  
        node.next = head; // new node points to old head  
        head = node; // head moves up  
    }  
  
    int pop() {  
        int val = head.data;  
        head = head.next; // head moves down  
        return val;  
    }  
}
```

■ ■ Array-Based Queue Implementation

Circular Queue using Array

JAVA

```
class Queue {  
    private int[] arr;  
    private int front, rear, size, capacity;
```

```
Queue(int cap) {
    capacity = cap;
    arr = new int[cap];
    front = 0; rear = -1; size = 0;
}

void enqueue(int x) {
    rear = (rear + 1) % capacity; // circular wrap
    arr[rear] = x;
    size++;
}

int dequeue() {
    int val = arr[front];
    front = (front + 1) % capacity; // circular wrap
    size--;
    return val;
}

boolean isEmpty() { return size == 0; }
int size() { return size; }
}
```

■■ Why circular? Without the circular wrap (% capacity), both front and rear keep moving right. You'd run out of space even when earlier slots are empty. The modulo operation wraps them back to index 0.

■ Memory Perspective

- Array Stack: Contiguous memory. top is just an index. Push = `arr[++top] = x`. Pop = return `arr[top--]`.
- LinkedList Stack: Heap-allocated nodes. Push = allocate node, update head. Pop = move head, GC frees.
- Java's Stack: Uses `java.util.Stack` (extends `Vector`, thread-safe but slower). Use `Deque` instead.
- Java's Queue: Use `LinkedList` or `ArrayDeque`. `ArrayDeque` is faster (no synchronization overhead).