Implementing Queue in Java (Using Internal Collections)

🧠1️⃣What is a Queue?

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.
Think of it like a ticket line — the person who comes first gets served first.

Common operations:

Operation	Description
`add()` / `offer()`	Insert element at the end
`remove()` / `poll()`	Remove element from the front
`peek()` / `element()`	View front element without removing

⚙️2️⃣Queue Implementations in Java (Internal Collections)

Java provides several built-in classes in `java.util` package that implement the `Queue` interface.

Let's explore the main ones 👇

🧩 A. LinkedList (Implements Queue + Deque)

- Internally uses **doubly linked list**.
- Can be used as **Queue, Stack, or Deque**.
- Not thread-safe.

Key Methods:

Method	Description
`add(E e)`	Inserts element at tail
`remove()`	Removes head element

`peek()`	Retrieves but does not remove head

Example:

```
import java.util.*;

public class LinkedListQueue {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        queue.add("Python");
        queue.add("Java");
        queue.add("C++");

        System.out.println("Queue: " + queue);
        System.out.println("Removed: " + queue.remove());
        System.out.println("Front: " + queue.peek());
        System.out.println("Queue after removal: " + queue);
    }
}
```

🏆 B. PriorityQueue

- Elements are ordered based on **natural order** or a **custom comparator**.
- Does **not** maintain FIFO — it's based on **priority**.
- Internally uses a **heap**.

Key Methods:

Method	Description
`offer(E e)`	Inserts element in priority order
`poll()`	Removes and returns the highest-priority element
`peek()`	Returns (but does not remove) the highest-priority element

Example:

```java
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<Integer> pq = new PriorityQueue<>();

        pq.offer(30);
        pq.offer(10);
        pq.offer(20);

        System.out.println("PriorityQueue: " + pq);
        System.out.println("Removed: " + pq.poll());
        System.out.println("Front: " + pq.peek());
        System.out.println("Queue after removal: " + pq);
    }
}
```

💬 *Note:* Output order may not look sorted because PriorityQueue's internal structure (heap) doesn't guarantee ordered iteration — only removal order is based on priority.

🚀 C. ArrayDeque (Most Efficient for Normal Queues)

- Implemented using **resizable array** (like ArrayList but double-ended).
- **Faster** than LinkedList for both Queue and Stack operations.
- Can act as both **Queue** and **Deque**.

Key Methods:

Method	Description
`offer(E e)`	Adds element at tail
`poll()`	Removes element from head
`peek()`	Retrieves head element
`offerFirst()` / `offerLast()`	Adds at front or end
`pollFirst()` / `pollLast()`	Removes from front or end

Example:

```
import java.util.*;

public class ArrayDequeQueue {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<>();

        queue.offer("Python");
        queue.offer("C");
        queue.offer("Java");

        System.out.println("Queue: " + queue);
        System.out.println("Removed: " + queue.poll());
        System.out.println("Front: " + queue.peek());
        System.out.println("Queue after removal: " + queue);
    }
}
```

⚖️3️⃣ Performance Comparison

Implementation	Internal Structure	Ordering	Nulls Allowed	Thread-Safe	Best For
LinkedList	Doubly Linked List	FIFO	✅	❌	Simple queues
PriorityQueue	Heap	Priority-based	❌	❌	Task scheduling
ArrayDeque	Resizable Array	FIFO (or Deque)	❌	❌	High-performance queues

---

🗒️ Summary

- ✅ Use **ArrayDeque** for normal queue operations (best performance).
- 🔝 Use **PriorityQueue** when ordering by priority is needed.
- 🔁 Use **LinkedList** if you need a flexible structure that works as both queue and list.

```
package Queue_Examples;

import java.util.ArrayDeque;
import java.util.Scanner;
import java.util.Queue;

public class Binary_sequence
{
    static void binary_sequence_printer(int n)
    {
        //use ArrayDeque of String
        Queue<String> bin=new ArrayDeque<>();
        //add 1 to front
        bin.add("1");
        //run this
        for(int i=1;i<n;i++)
        {
            //print front and remove it
            String current=bin.poll();
            System.out.print("\n"+i+"-----"+current);
            //add element at front+"0" to ArrayDeque
            bin.add(current+"0");
            //add element at front+"1" to ArrayDeque
            bin.add(current+"1");
        }
```

```
    }
    public static void main(String[] args) {
        Scanner in =new Scanner(System.in);
        System.out.println("Enter n:");
        int n=in.nextInt();
        Binary_sequence.binary_sequence_printer(n);

    }
}
```

Question: Best Time to Buy and Sell a Share

(Using Queue - 15 Days Price Window)

Problem Statement:

You are given the daily prices of a company's share over **15 days**.
Your task is to find the **best day to buy** and **best day to sell** to achieve the **maximum profit**.

You must use a **Queue** to process the 15-day prices in the same order as they arrive (FIFO).

If no profit is possible (prices always decrease), print `0` as the maximum profit and show that no valid transaction can be made.

Detailed Explanation:

- You'll be given a **Queue of integers**, each representing the share price of a day.
- You need to determine:
    - The **buy day (index starting from 1)**
    - The **sell day**
    - And the **maximum profit = sell_price - buy_price**

Example Input:

```
Enter 15 days share prices:
100 180 260 310 40 535 695 30 50 70 20 40 60 90 10
```

Expected Output:

```
Best Day to Buy: Day 5  (Price = 40)
Best Day to Sell: Day 7 (Price = 695)
Maximum Profit: 655
```

```
package Queue_Examples;


public class Max_Profit
{
   static void Profit_Maker(int n[])
   {
      int Max_Profit=0,Buy_on=0,Sell_on=0;
      for(int buy=0;buy<n.length-1;buy++)
      {
         for(int sell=n.length-1;sell>buy;sell--)
         {
            int profit=n[sell]-n[buy];
            if(profit>Max_Profit)
            {
               Max_Profit=profit;
               Buy_on=buy;
               Sell_on=sell;
            }
         }
      }
      System.out.println("Max Profit "+Max_Profit+" by buy on "+Buy_on+" and sell
on"+Sell_on);
```

```
    }


    public static void main(String[] args)
    {
        int a[]={100,180, 260, 310 ,40 ,535, 695, 30, 50, 70 ,20 ,40 ,60 ,90, 10
        };
        Max_Profit.Profit_Maker(a);


    }
}
```

Simulate Bidding Process for an Item (Only Bid Amounts)

Problem Statement:

You are building a program to simulate the bidding process for an item.

- Buyers enter **bid amounts** one by one.
- The process continues until the user enters a **blank line** (press Enter without any number).
- Your task is to:
    - Display all the bids entered.
    - Display the **highest bid amount** using a collection such as `ArrayList` and `Collections.max()`.

Input Format:

Enter bid amounts one per line.
Press Enter on a blank line to stop.

Output Format:

- List of all bids entered
- Highest bid amount

Sample Input:

```
Enter bid amount (blank to stop): 4500
Enter bid amount (blank to stop): 6800
Enter bid amount (blank to stop): 7200
Enter bid amount (blank to stop): 6900
Enter bid amount (blank to stop):
```

Expected Output:

```
All Bids: [4500, 6800, 7200, 6900]
Highest Bid: 7200
```

\

```
package Stack_Examples;

public class Two_Stacks_In_One
{
    int stack[];
    int tos1,tos2,MaxSize;
    void create_Stack(int size)
    {
        MaxSize=size;
        tos1=-1;tos2=MaxSize;
        stack=new int[MaxSize];
    }
    //push:accepts an element on stack
    //tos+1
    void push1(int e)
    {
        tos1++;
        stack[tos1]=e;
        //stack[++tos]=e;
    }
    void push2(int e)
    {
        tos2--;
        stack[tos2]=e;
        //stack[++tos]=e;
    }
    boolean is_full()
    {
        if(tos1+1==tos2)
            return true;
        else
            return false;

        //return(tos==MaxSize-1);

    }
    int pop1()
    {
        int temp=stack[tos1];
        tos1--;
        return(temp);
        //return(stack[tos--]);
    }
    int pop2()
    {
        int temp=stack[tos2];
        tos2++;
        return(temp);
```

```
        //return(stack[tos--]);
    }



    boolean is_empty1()
    {
        if(tos1==-1)
            return true;
        else
            return false;

        //return(tos==-1);

    }
    boolean is_empty2()
    {
        if(tos2==MaxSize)
            return true;
        else
            return false;

        //return(tos==-1);

    }


    void print_stack1()//in LIFO
    {
        for(int i=tos1;i>=0;i--)
            System.out.println(stack[i]);
    }
    void print_stack2()//in LIFO
    {
        for(int i=tos2;i<MaxSize;i++)
            System.out.println(stack[i]);
    }
    int peek1()//only return element on top
    {
        return(stack[tos1]);
    }
    int peek2()//only return element on top
    {
        return(stack[tos2]);
    }
}
```

```
package Queue_Examples;

public class Two_Queue_in_an_Array
{
    int queue[];
    int front1,rear1,front2,rear2,MaxSize;
    void create_Queue(int size)
    {
        rear1=-1;
        front1=0;
        rear2=MaxSize;
        front2=MaxSize-1;
        MaxSize=size;
        queue=new int[MaxSize];
    }
    //enqueue:accepts an element in queue
//rear+1
    void enqueue1(int e)
    {
        queue[++rear1]=e;
    }
    void enqueue2(int e)
    {
        queue[--rear2]=e;
    }
    boolean is_full()
    {
        return((rear2-1)==rear1);
        //((rear2==(rear1+1));
    }



    int dequeue1()
    {
        int temp=queue[front1];
        front1++;
        return(temp);
        //return(queue[front++);
    }
    int dequeue2()
    {
        int temp=queue[front2];
        front2--;
        return(temp);
        //return(queue[front++);
    }
```

```
    boolean is_empty1()
    {
        if(front1>rear1)
            return true;
        else
            return false;
        //return(front>rear);
    }
    boolean is_empty2()
    {
        if(front2<rear2)
            return true;
        else
            return false;
        //return(front>rear);
    }

    void print_queue1()//in FIFO-->front to rear
    {
        for(int i=front1;i<=rear1;i++)
            System.out.print(queue[i]+" - ");
    }
    void print_queue2()//in FIFO-->front to rear
    {
        for(int i=front2;i>=rear2;i--)
            System.out.print(queue[i]+" - ");
    }


}
```