Java PriorityQueue

In Java, a **`PriorityQueue`** is a class from the **`java.util`** package that implements the **`Queue`** interface. It holds elements in a way that the **head of the queue is always the least element**, according to the **natural ordering** or a **custom comparator**.

🔸 Key Points

- **Ordered based on priority** (natural order or custom).
- **Not thread-safe** (use **`PriorityBlockingQueue`** for concurrent use).
- **Does not allow null elements.**
- **Implements a min-heap** internally.

✅ Syntax

```java
PriorityQueue<Type> pq = new PriorityQueue<>();
```

✅ Common Constructors

```java
PriorityQueue()                                // Natural ordering
PriorityQueue(int initialCapacity)
PriorityQueue(Comparator<? super E> comparator) // Custom ordering
```

🔹 Example: Natural Ordering (Min-Heap)

```java
import java.util.PriorityQueue;

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

        pq.add(10);
        pq.add(5);
        pq.add(20);

        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // prints 5, 10, 20
        }
    }
}
```

🔹 Example: Custom Comparator (Max-Heap)

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        maxHeap.add(10);
        maxHeap.add(5);
        maxHeap.add(20);

        while (!maxHeap.isEmpty()) {
            System.out.println(maxHeap.poll()); // prints 20, 10, 5
        }
    }
}
```

🔹 Useful Methods

Method	Description
`add(e)`	Inserts element
`offer(e)`	Same as add, but returns false if failed
`poll()`	Retrieves and removes head
`peek()`	Retrieves head without removing
`remove(e)`	Removes a specific element
`isEmpty()`	Checks if the queue is empty
`size()`	Returns the number of elements

🔹 Use Case

- **Task scheduling**
- **Dijkstra's algorithm**
- **Huffman encoding**
- **Real-time system priority queues**

🔸 Notes

- Internally uses a **binary heap**.
- Default ordering is **ascending** (min-heap).
- Elements must be **comparable** (implement **`Comparable`** or use a **`Comparator`**).

Would you like an example using custom objects (e.g., jobs with priority)?

Here are **real-time use cases** of **`PriorityQueue`** in **testing or QA automation**, especially relevant if you're working with **Java + Selenium + TestNG/JUnit** or backend systems:

✅ 1. Test Execution Prioritization

You can use **`PriorityQueue`** to **dynamically decide the order of test execution** based on priority.

```java
class TestCase implements Comparable<TestCase> {
    String name;
    int priority;

    TestCase(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    @Override
    public int compareTo(TestCase other) {
        return this.priority - other.priority; // lower value = higher priority
    }
}
```

```java
PriorityQueue<TestCase> queue = new PriorityQueue<>();
queue.add(new TestCase("LoginTest", 1));
queue.add(new TestCase("ProfileUpdateTest", 3));
queue.add(new TestCase("PaymentTest", 2));

while (!queue.isEmpty()) {
    TestCase tc = queue.poll();
    System.out.println("Running: " + tc.name);
}
```

✅ 2. Bug Triage System

Simulate **bug priority processing** in automation, helping test scripts to validate that **high-priority bugs get addressed first**.

```java
class BugReport implements Comparable<BugReport> {
    String id;
    int severity; // 1: Critical, 2: High, 3: Medium, 4: Low

    public BugReport(String id, int severity) {
        this.id = id;
        this.severity = severity;
    }

    public int compareTo(BugReport o) {
        return Integer.compare(this.severity, o.severity);
    }
}
```

✅ 3. Job Queue Simulation in Load Testing

Automate test cases that simulate **job queues**, like background processes or batch jobs.

```java
class Job implements Comparable<Job> {
    String name;
    int executionTime;

    Job(String name, int executionTime) {
        this.name = name;
        this.executionTime = executionTime;
    }

    public int compareTo(Job other) {
        return this.executionTime - other.executionTime;
    }
}
```

✅ Useful in **backend performance testing** where jobs need to be handled in order of load or time sensitivity.

✅ 4. Retry Failed Tests by Priority

Maintain a queue of **failed test cases** based on their criticality for retry.

✅ 5. Dynamic Test Suite Prioritization

In CI/CD pipelines, use a **`PriorityQueue`** to **sort and execute tests** based on:

- Severity of the feature
- Past test failures
- Execution duration

✅ 6. API Rate Limiting Simulation

Simulate systems where **requests are processed by importance**.

- Critical API tests go first.
- Less critical ones wait.

✅ 7. Resource Management Simulation

Automate tests that verify system handles **tasks with different priorities** when **limited resources** are available.

Would you like a full Java + TestNG example of any of these?