Actual Output

--- 🩺 Starting Hospital Patient Queue System Simulation ---

✅ CHECK-IN: 'John' (Severity: 3) has been added to the queue.

✅ CHECK-IN: 'Mary' (Severity: 1) has been added to the queue.

✅ CHECK-IN: 'Peter' (Severity: 3) has been added to the queue.

✅ CHECK-IN: 'Sarah' (Severity: 2) has been added to the queue.

🏥 CURRENT PATIENT QUEUE:

1. Mary (Severity: 1)

2. Sarah (Severity: 2)

3. John (Severity: 3)

4. Peter (Severity: 3)

🩺 CHECK-OUT: Doctor is now seeing 'Mary' (Severity: 1).

🏥 CURRENT PATIENT QUEUE:

1. Sarah (Severity: 2)

2. John (Severity: 3)

3. Peter (Severity: 3)

**✍️ Explanation**

The provided Java code implements a **priority queue** to solve the hospital patient queue problem. Here's a breakdown of the key components and how they work together:

1. **Patient Class**: This class represents a patient with three attributes: name, severity, and arrivalTime.
   * severity: A numerical value indicating how critical a patient's condition is. A lower number means higher priority.
   * arrivalTime: A timestamp to handle tie-breaking. This ensures that if two patients have the same severity level, the one who arrived earlier is seen first, maintaining a **First-In, First-Out (FIFO)** order for equal-priority patients.
2. **PatientComparator Class**: This is the core of the priority queue's custom logic. The compare method defines the priority rules.
   * It first compares the severity of two patients. The patient with the smaller severity value is given higher priority.
   * If the severities are identical, it then compares the arrivalTime to ensure that the patient who arrived earlier is prioritized. This two-part comparison is crucial for meeting all the requirements of the problem statement.
3. **HospitalQueueSystem Class**: This class contains the main logic for managing the patient queue.
   * **PriorityQueue<Patient> patientQueue**: This is the data structure used. Instead of a simple Queue, a PriorityQueue is chosen because it automatically orders elements based on a specified priority. By passing the PatientComparator to its constructor, we tell the PriorityQueue how to sort our Patient objects.
   * **checkIn()**: When a patient checks in, a new Patient object is created and added to the patientQueue using the add() method. The PriorityQueue automatically places the new patient in the correct position based on our custom comparator.
   * **checkOut()**: When a doctor is ready, the poll() method is called. This method retrieves and removes the highest-priority patient (the one at the head of the queue). The beauty of the PriorityQueue is that this operation is highly efficient.

* + **displayQueue()**: To display the queue in its current prioritized order without changing it, we create a temporary copy of the main patientQueue. This allows us to poll() from the temporary copy and print the patients in order, leaving the original queue intact for future operations.

**🎯 Conclusion**

This system effectively models a real-world hospital triage scenario using a **priority queue**. The use of a custom Comparator allows the system to prioritize patients based on a dual-criteria logic: **severity** first, then **arrival time** as a tie-breaker. This combination ensures that the most critical patients are always seen first, while also maintaining fairness among patients with the same level of urgency. This approach is highly efficient for managing dynamic queues where priorities can change and is a perfect example of how an appropriate data structure can elegantly solve a complex logical problem.