Explanation of Shortest Job First (SJF) scheduling algorithm implementation:

1. NumOfProcess Class:

```java
class NumOfProcess {
    String name;
    int burstTime;

    NumOfProcess(String name, int burstTime) {
        this.name = name;
        this.burstTime = burstTime;
    }
}
```

This class represents a process with a name and burst time. It encapsulates the basic attributes needed for SJF scheduling.

2. Main Method and Process Generation:

```java
public static void main(String[] args) {
    int numProcesses = 79;
    int transientEventTime = 5;
    List<NumOfProcess> numOfProcesses = generateProcesses(numProcesses);
    sjfScheduling(numOfProcesses, transientEventTime);
}

private static List<NumOfProcess> generateProcesses(int count) {
    List<NumOfProcess> numOfProcesses = new ArrayList<>(count);
    for (int i = 1; i <= count; i++) {
        numOfProcesses.add(new NumOfProcess("P" + i, RANDOM.nextInt(10) +
1));
    }
    return numOfProcesses;
}
```

The main method sets up the number of processes (79) and the time for the transient event (5). It then generates the processes using a separate method, which creates processes with random burst times between 1 and 10.

3. SJF Scheduling Method:

```java
```

```
public static void sjfScheduling(List<NumOfProcess> numOfProcesses, int
transientEventTime) {
    PriorityQueue<NumOfProcess> processQueue = new
PriorityQueue<>(Comparator.comparingInt(p -> p.burstTime));
    processQueue.addAll(numOfProcesses);

    int currentTime = 0;
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;
    boolean transientEventOccurred = false;

    StringBuilder output = new StringBuilder("Process\tBurst Time\tWaiting
Time\tTurnaround Time\n");

    while (!processQueue.isEmpty()) {
        NumOfProcess numOfProcess = processQueue.poll();

        // Transient event handling
        if (currentTime >= transientEventTime && !transientEventOccurred) {
            numOfProcess.burstTime += 3;
            output.append(String.format("Transient event: Process %s burst
time adjusted by +3 units at time %d\n",
                    numOfProcess.name, currentTime));
            transientEventOccurred = true;
            processQueue.add(numOfProcess);
            continue;
        }

        int waitingTime = currentTime;
        int turnaroundTime = waitingTime + numOfProcess.burstTime;

        totalWaitingTime += waitingTime;
        totalTurnaroundTime += turnaroundTime;
        currentTime += numOfProcess.burstTime;

        output.append(String.format("%s\t\t%d\t\t\t%d\t\t\t\t%d\n",
                numOfProcess.name, numOfProcess.burstTime, waitingTime,
turnaroundTime));
    }

    // Print results and averages
    System.out.print(output);
    System.out.printf("\nAverage waiting time = %.2f\n", (double)
```

```
totalWaitingTime / numOfProcesses.size());
    System.out.printf("Average turnaround time = %.2f\n", (double)
totalTurnaroundTime / numOfProcesses.size());
}
```

This is the core of the SJF algorithm:
- It uses a PriorityQueue to automatically sort processes by burst time.
- It processes each job in order of shortest burst time.
- It handles a transient event at a specified time, which increases the burst time of the current process.
- It calculates waiting time and turnaround time for each process.
- Finally, it calculates and prints average waiting and turnaround times.

Key Features of this Implementation:
1. Use of PriorityQueue: This ensures that processes are always sorted by burst time, implementing the core idea of SJF scheduling efficiently.
2. Transient Event Handling: The implementation simulates a real-world scenario where a process's burst time can unexpectedly increase.
3. Non-preemptive: Once a process starts executing, it runs to completion.
4. Performance Metrics: It calculates and displays key performance metrics like waiting time and turnaround time for each process, as well as averages for the entire set of processes.
5. Efficient Sorting: The PriorityQueue handles the sorting of processes, which is more efficient than sorting the entire list repeatedly.

This implementation demonstrates how SJF works, showing how processes with shorter burst times are given preference in execution. The inclusion of a transient event adds an element of dynamism to the simulation, reflecting real-world scenarios where process execution times might be adjusted during runtime.