Explanation of the Round Robin scheduling algorithm implementation:

1. Process Class:

~~~
```java
private static class Process {
    String name;
    int burstTime;
    int remainingTime;

    Process(String name, int burstTime) {
        this.name = name;
        this.burstTime = burstTime;
        this.remainingTime = burstTime;
    }
}
```
~~~

This inner class represents a process with a name, burst time, and remaining time. The remaining time is initially set to the burst time and decreases as the process is executed.

2. Main Method and Initialization:

~~~
```java
public static void main(String[] args) {
    final int NUM_PROCESSES = 79;
    final int TIME_QUANTUM = 2;
    final int MAX_TRANSIENT_EVENTS = 10;

    Queue<Process> processQueue = new LinkedList<>();
    Random random = new Random();
    for (int i = 0; i < NUM_PROCESSES; i++) {
        processQueue.add(new Process("P" + (i + 1), (i % 10) + 1));
    }
```
~~~

The main method sets up constants and initializes a queue of processes. Each process is given a name (P1, P2, etc.) and a burst time (cyclically assigned values from 1 to 10).

3. Transient Events:

~~~java

```java
int numTransientEvents = random.nextInt(MAX_TRANSIENT_EVENTS) + 1;
System.out.println("Number of transient events: " + numTransientEvents);
for (int i = 0; i < numTransientEvents; i++) {
    int index = random.nextInt(NUM_PROCESSES);
~~~

~~~
    int adjustment = random.nextInt(5) + 1;
    List<Process> processes = new ArrayList<>(processQueue);
    Process transientProcess = processes.get(index);
    transientProcess.burstTime += adjustment;
    transientProcess.remainingTime += adjustment;
    System.out.println("Transient event: Process " + transientProcess.name
+ " burst time adjusted by " + adjustment + " units");
}
```
~~~

This section simulates transient events by randomly adjusting burst times of processes. The number of events and their effects are randomized.

4. Process Index Mapping:

~~~
```java
Map<String, Integer> processIndexMap = new HashMap<>();
int index = 0;
for (Process p : processQueue) {
    processIndexMap.put(p.name, index++);
}
```
~~~

This creates a mapping of process names to their original indices, which is used later to compute waiting times correctly.

5. Round Robin Scheduling:

~~~
```java
while (!processQueue.isEmpty()) {
    Process process = processQueue.poll();
    int timeToRun = Math.min(process.remainingTime, TIME_QUANTUM);
    process.remainingTime -= timeToRun;
    currentTime += timeToRun;

    if (process.remainingTime > 0) {
        processQueue.add(process);
    } else {
        int processIndex = processIndexMap.get(process.name);
        waitingTimes[processIndex] = currentTime - process.burstTime;
        turnaroundTimes[processIndex] = waitingTimes[processIndex] +
process.burstTime;
        processedCount++;
    }
}
~~~

~~~
```
~~~
This is the core of the Round Robin algorithm:
- It runs each process for a time quantum or until completion.
- If a process is not finished, it's added back to the queue.
- When a process completes, its waiting and turnaround times are calculated.

6. Results Calculation and Printing:

~~~
```java
System.out.println("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time");
for (int i = 0; i < NUM_PROCESSES; i++) {
    System.out.printf("P%d\t\t%d\t\t\t\t%d\t\t\t%d\n", i + 1,
waitingTimes[i] + turnaroundTimes[i] - waitingTimes[i], waitingTimes[i],
turnaroundTimes[i]);
}
~~~

~~~
double totalWaitingTime = Arrays.stream(waitingTimes).sum();
double totalTurnaroundTime = Arrays.stream(turnaroundTimes).sum();

System.out.printf("\nAverage waiting time = %.2f\n", totalWaitingTime /
NUM_PROCESSES);
System.out.printf("Average turnaround time = %.2f\n", totalTurnaroundTime /
NUM_PROCESSES);
```
~~~
This section prints the results for each process and calculates average waiting and turnaround times.

Key Features of this Implementation:
1. Use of Queue: Efficiently manages the order of process execution.
2. Transient Events: Simulates real-world scenarios where process requirements can change unexpectedly.
3. Time Quantum: Implements the core concept of Round Robin scheduling.
4. Process Tracking: Uses a map to keep track of original process order, ensuring correct calculation of waiting times.
5. Dynamic Execution: Processes are re-added to the queue if they're not finished, ensuring fair execution.

This implementation provides a realistic simulation of the Round Robin scheduling algorithm, incorporating the complexity of transient events. It demonstrates how processes are executed in a time-sliced manner, providing fairness in CPU allocation while handling unexpected changes in process requirements.