Explanation of the optimized First Come First Serve (FCFS) scheduling algorithm codet:

1. Class Structure and Constants:

```
public class FirstComeFirstServe {
    private static final int NUM_PROCESSES = 79;
    private static final int MAX_TRANSIENT_EVENTS = 10;
    private static final Random RANDOM = new Random();

    static class Process {
        String name;
        int burstTime;
        int waitingTime;
        int turnaroundTime;

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

- The class defines constants for the number of processes and maximum transient events.
- A static nested `Process` class encapsulates all process-related data, improving organization and readability.

2. Main Method:

```java
public static void main(String[] args) {
    Process[] processes = generateProcesses();
    introduceTransientEvents(processes);
    calculateTimes(processes);
    printResults(processes);
}
```

- The main method orchestrates the overall flow of the program.
- It calls methods to generate processes, introduce transient events, calculate times, and print results.

3. Process Generation:

```java
```

```
private static Process[] generateProcesses() {
    Process[] processes = new Process[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++) {
        processes[i] = new Process("P" + (i + 1), (i % 10) + 1);
    }
    return processes;
}
```

- This method creates an array of `Process` objects.
- Each process is given a name (P1, P2, etc.) and a burst time (cyclically assigned values from 1 to 10).

4. Transient Events:

```java
private static void introduceTransientEvents(Process[] processes) {
    int numTransientEvents = RANDOM.nextInt(MAX_TRANSIENT_EVENTS) + 1;
    System.out.println("\nNumber of transient events: " +
numTransientEvents);
    for (int i = 0; i < numTransientEvents; i++) {
        int processIndex = RANDOM.nextInt(NUM_PROCESSES);
        int adjustment = RANDOM.nextInt(5) + 1;
        processes[processIndex].burstTime += adjustment;
        System.out.println("Transient event: Process " +
processes[processIndex].name +
                           " burst time adjusted by " + adjustment + "
units");
    }
}
```

- This method simulates transient events by randomly adjusting burst times of processes.
- The number of events and their effects are randomized, adding unpredictability to the simulation.

5. Time Calculation:

```
private static void calculateTimes(Process[] processes) {
    int currentTime = 0;
    for (Process process : processes) {
        process.waitingTime = currentTime;
        currentTime += process.burstTime;
        process.turnaroundTime = currentTime;
    }
```

```
}
```

- This method calculates waiting and turnaround times for each process.
- It uses a single pass through the processes, maintaining a `currentTime` variable to track the system time.
- Waiting time for a process is the current time when the process starts.
- Turnaround time is the current time when the process finishes.

6. Results Printing:

```
private static void printResults(Process[] processes) {
    System.out.println("\nProcess\tBurst Time\tWaiting Time\tTurnaround
Time");
    double totalWaitingTime = 0;
    double totalTurnaroundTime = 0;

    for (Process process : processes) {
        System.out.printf("%s\t\t%d\t\t\t%d\t\t\t%d\n",
                          process.name, process.burstTime,
process.waitingTime, process.turnaroundTime);
        totalWaitingTime += process.waitingTime;
        totalTurnaroundTime += process.turnaroundTime;
    }

    double avgWaitingTime = totalWaitingTime / NUM_PROCESSES;
    double avgTurnaroundTime = totalTurnaroundTime / NUM_PROCESSES;

    System.out.printf("\nAverage waiting time = %.2f\n", avgWaitingTime);
    System.out.printf("Average turnaround time = %.2f\n",
avgTurnaroundTime);
}
```

- This method prints the results of the scheduling simulation.
- It displays details for each process and calculates average waiting and turnaround times.

Key Optimizations:

1. Object-Oriented Approach: Using a `Process` class improves code organization and readability.
2. Single-Pass Calculation: Waiting and turnaround times are calculated in one pass, improving efficiency.
3. Modular Design: Separate methods for different functionalities improve maintainability.

4. Efficient Random Number Generation: A single `Random` instance is used throughout the program.
5. Formatted Output: `System.out.printf()` is used for cleaner, formatted output.

This implementation efficiently simulates the FCFS scheduling algorithm with the added complexity of random transient events, providing a realistic scenario for process scheduling in operating systems.