Lab1: Trainspotting Documentation

Tesfu Tekleab Hailu
Oskar Heijkenskjöld

Group 5

September 15, 2025

1 Introduction

This document describes the `Lab1.java` solution for a trainspotting on a track defined by `Lab1.map`. The program controls two trains (ID 1 and 2) moving between north and south stations, ensuring no collisions through semaphore-based synchronization. The track, a 22x15 grid, includes sensors, switches, and critical sections (protected by semaphores) to manage train paths. This documentation provides a high-level overview, details essential code components, and discusses sensor placement, critical sections, train speed, and testing, as required for correctness and clarity.

2 High-Level Description

The `Lab1.java` program simulates two trains (ID 1 and 2) navigating a track with parallel paths across multiple zones, covering the entire active map (rows 3–13, columns 1–19). Trains start at opposite ends (train 1 southbound, train 2 northbound) with user-defined speeds. The track is divided into:

- **North Zone (rows 3–5, cols 13–18)**: Includes north stations and parallel tracks.
- **T-Crossing Zone (rows 7–9, cols 14–19)**: Features parallel upper (row 8) and lower (row 7) T-paths merging at a switch.
- **Middle Zone (rows 9–10, cols 6–13)**: Parallel tracks connecting east and west merges.
- **South Zone (rows 11–13, cols 2–5)**: Parallel south loops to stations.
- **Central Crossing (rows 6–8, cols 6–10)**: Intersection of vertical and horizontal tracks.
- **West Merge (rows 9–10, cols 2–6)**: Convergence of mid-paths toward south paths.

The track has stations at (15,4) (north) and (15,12) (south), switches at (3,11), (4,9), (15,9), (17,7), and sensors to trigger actions. The program uses the `TSim` library, with each train running in a thread (`Runnable`), reacting to sensor events in an infinite loop. Nine semaphores ensure exclusive access to shared track sections, prioritizing the upper T-path (row 8) initially to sequence train movements safely.

3 Track Layout

The track, defined in `Lab1.map` including sensors and marked critical sections, is visualized in the annotated screenshot below.

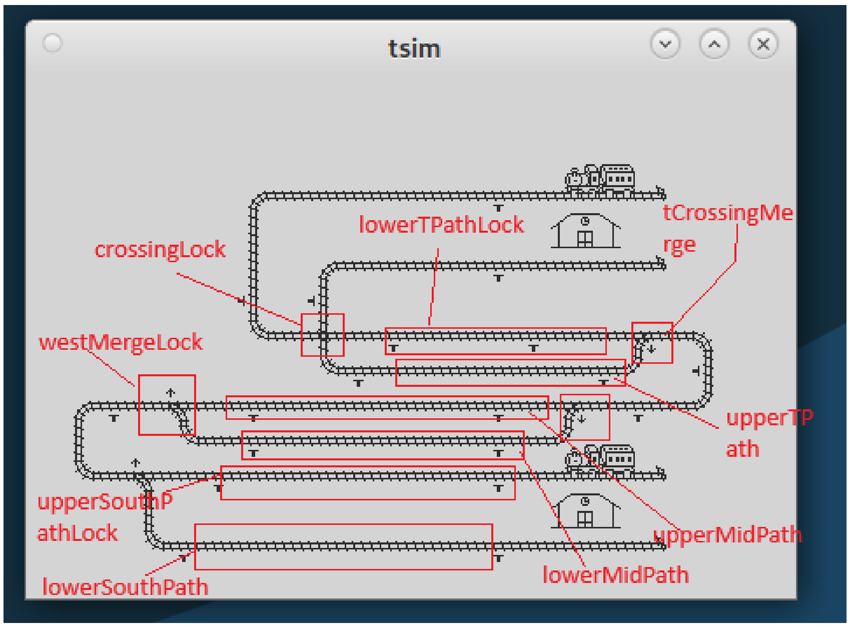

Figure 1: Train track setup with the sensors and marked critical areas.

Note: Please ignore the area marked by rectangle at coordinate (15, 9) which also a switch as it was by mistake.
Key features:

- **North Zone**: Rows 3–5, columns 13–18; stations at (15,4), sensors at (13,3), (13,5).
- **T-Crossing Merge**: Rows 7–9, columns 14–19; switch at (17,7), sensors at (14,7), (16,8), (19,8), (17,9).
- **Middle Zone**: Rows 9–10, columns 6–13; switch at (15,9), sensors at (13,9), (13,10), (6,9), (6,10).
- **West Merge**: Rows 9–10, columns 2–6; switch at (4,9), sensor at (2,9).
- **South Zone**: Rows 11–13, columns 2–5; switch at (3,11), sensors at (5,11), (4,13).
- **Central Crossing**: Rows 6–8, columns 6–10; sensors at (6,6), (8,6), (9,8), (10,7).

The screenshot marks critical sections (e.g., `crossingLock`, `tCrossingMerge`, `westMergeLock`) and sensors/switches with code-consistent labels.

4 Critical Code Components

The program's correctness relies on proper initialization, semaphore management, and sensor-driven logic. Below, we detail essential code sections.

4.1 Constructor and Initialization

```java
public Lab1(int speed1, int speed2) {
    TSimInterface tsi = TSimInterface.getInstance();
    tsi.setDebug(false);
    try {
        tsi.setSpeed(1, speed1);
        tsi.setSpeed(2, speed2);
        tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT); // Set
           default to row 11 (RIGHT)
        tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT); // Set
           default to row 7 (RIGHT)
    } catch (CommandException e) {
        e.printStackTrace();
        System.exit(1);
    }
```

- **Purpose**: Initializes the simulation with train speeds and switch settings.
- **Correctness**: Sets initial speeds for trains 1 and 2. The switch at (3,11) to `SWITCH_RIGHT` directs to the lower south path (row 11). The switch at (17,7) to `SWITCH_RIGHT` sets the default T-merge path to row 7 (lower), aligning with `upperTPath` priority.

4.2 Semaphore Declarations

```java
Semaphore crossingLock = new Semaphore(1);
Semaphore lowerTPathLock = new Semaphore(0);
Semaphore upperTPath = new Semaphore(1);
Semaphore tCrossingMerge = new Semaphore(1);
Semaphore upperMidPath = new Semaphore(1);
Semaphore lowerMidPath = new Semaphore(1);
Semaphore westMergeLock = new Semaphore(1);
Semaphore upperSouthPathLock = new Semaphore(0);
Semaphore lowerSouthPath = new Semaphore(1);
```

- **Purpose**: Defines semaphores to protect critical sections.
- **Correctness**:
  - `crossingLock` (1): Allows one train in the central crossing (rows 6–8, cols 6–10).
  - `lowerTPathLock` (0): Locks lower T-path (row 7, cols 6–17) initially, forcing upper path use.
  - `upperTPath` (1): Available for upper T-path (row 8, cols 8–17) at startup.

  - `tCrossingMerge` (1): Protects T-merge (rows 7–9, cols 14–19).
  - `upperMidPath`, `lowerMidPath` (1): Exclusive access to mid-path lanes (rows 9, 10).
  - `westMergeLock` (1): Guards west merge (rows 9–10, cols 2–6).
  - `upperSouthPathLock` (0), `lowerSouthPath` (1): Prioritize lower south path (row 13).

4.3 Train Logic (run Method)

```
class Train implements Runnable {
    int id;
    int speed;
    boolean direction; // True = north
    // Constructor omitted
    @Override
    public void run() {
        try {
            while (true) {
                SensorEvent se = tsi.getSensor(id);
                // Sensor handlers follow
```

- **Purpose**: Each train runs in a thread, reacting to sensor events to manage semaphores and switches.
- **Correctness**: The infinite loop processes sensor events, ensuring trains stop, acquire/release semaphores, and set switches before moving. The `direction` flag (`true` = northbound, `false` = southbound) guides logic.

4.4 Key Sensor Handlers

- **Crossing Handlers (e.g., (6,6), (8,6), (9,8), (10,7))**:

```
// Lower T-path past crossing
if (se.getXpos() == 6 && se.getYpos() == 6 && se.getStatus()
    == SensorEvent.ACTIVE) {
    if (!direction) { // to South
        tsi.setSpeed(id, 0);
        System.out.println("crossingLock acquired! - " +
            crossingLock.availablePermits());
        crossingLock.acquire();
        tsi.setSpeed(id, speed);
    } else { // to North
        System.out.println("crossingLock released! - " +
            crossingLock.availablePermits());
        crossingLock.release();
    }
}
```

  - **Correctness**: Acquires `crossingLock` when entering the crossing southbound, releases northbound or southbound on exit. Ensures one train in the crossing. In detail: Southbound, the train stops (`tsi.setSpeed(id, 0)`), prints the current permits for debugging, acquires the lock (blocking if occupied, reducing permits to 0), then resumes speed. Northbound, it releases the lock (increasing permits to 1), allowing others to enter, without stopping. This prevents collisions at the intersection (rows 6–8, cols 6–10).

- **T-Crossing Merge Handlers (e.g., (14,7), (16,8), (19,8))**:

```
// T-crossing merge east
if (se.getXpos() == 19 && se.getYpos() == 8 && se.getStatus()
    == SensorEvent.ACTIVE && direction) {
    tsi.setSpeed(id, 0);
    if (lowerTPathLock.tryAcquire()) {
        System.out.println("lowerTPathLock acquired! - " +
           lowerTPathLock.availablePermits());
        tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT); //
           row 8
    } else {
        System.out.println("upperTPath acquired! - " +
           upperTPath.availablePermits());
        upperTPath.acquire();
        tsi.setSwitch(17, 7, TSimInterface.SWITCH_LEFT); //
           row 7
    }
    tsi.setSpeed(id, speed);
}
```

  - **Correctness**: Northbound at (19,8), tries lower path (locked initially), falls back to upper (available), setting switch accordingly. Southbound handlers (14,7), (16,8) manage `tCrossingMerge` and path releases.

5 Sensor Placement

Sensors are placed at critical points to detect train positions and trigger semaphore/switch actions (see screenshot). The 19 sensors ensure timely synchronization across all zones:

- **Crossing**: Sensors at (6,6), (8,6), (9,8), (10,7) (4 sensors) surround the crossing, detecting entry/exit for `crossingLock` to prevent collisions at the intersection.
- **T-Merge**: Sensors at (14,7), (16,8), (19,8), (17,9) (4 sensors) manage path selection and merge at (17,7), ensuring safe switch settings and semaphore handling.
- **Mid-Path**: Sensors at (13,9), (13,10), (6,9), (6,10) (4 sensors) control entry/exit to upper and lower mid lanes, coordinating with adjacent merges.
- **West Merge**: Sensor at (2,9) (1 sensor) detects trains entering south paths or returning to mid-paths, triggering `westMergeLock` actions.
- **South Paths**: Sensors at (5,11), (4,13) (2 sensors) manage access to upper and lower south loops for semaphore release.

- **Stations**: Sensors at (13,3), (13,5), (13,11), (13,13) (4 sensors) trigger direction reversals at north and south stations.

Correctness: Sensors are positioned before/after switches and critical sections, ensuring precise synchronization.

6 Choice of Critical Sections

The nine semaphores protect:

1. `crossingLock`: Central crossing (rows 6–8, cols 6–10).
2. `tCrossingMerge`: T-merge (rows 7–9, cols 14–19).
3. `lowerTPathLock`, `upperTPath`: Lower/upper T-path arms (rows 7, 8).
4. `upperMidPath`, `lowerMidPath`: Central lanes (rows 9, 10).
5. `westMergeLock`: West merge (rows 9–10, cols 2–6).
6. `upperSouthPathLock`, `lowerSouthPath`: South loops (rows 11, 13).

Correctness: Each section is a shared resource. Semaphores ensure exclusivity, with initial values (e.g., `upperTPath=1`, `lowerTPathLock=0`) prioritizing upper T-path to avoid deadlocks. The switch at (15,9) is not a critical section because it diverges southbound (protected by `tCrossingMerge` and mid-path semaphores) and is covered by `tCrossingMerge` northbound, unlike converging switches (17,7), (4,9).

7 Maximum Train Speed

The program accepts arbitrary speeds via `speed1`, `speed2`. Testing revealed a maximum safe speed of 17 (TSim units); at 18 or higher, collisions occur due to simulation timing issues where trains overshoot sensors before semaphore logic executes, potentially entering critical sections without proper acquisition. Tested speed pairs (e.g., (16,17), (5,17), (5,16), (13,15), (5,15), (10,15)) worked without collisions, confirming robustness up to 17. The station sleep (`1000 + 20 * Math.abs(speed)`) scales with speed (e.g., 1200 ms at speed 10, 1340 ms at 17) to simulate realistic stops and ensure simulation stability by allowing sufficient time for sensor events and semaphore handling, preventing issues like those seen at speed 18.

8 Testing the Solution

Testing involved:

- **Varied Speeds**: Ran with speeds like (16,17), (5,17), (5,16), (13,15), (5,15), (10,15) to observe overtaking and waiting without collisions.
- **Deadlock Testing**: Used equal speeds (15,15), (10,10), (17,17) to check for deadlocks at merges; no deadlocks occurred due to prioritized paths.
- **Debug Prints**: Monitored semaphore permits (e.g., `crossingLock.availablePermits()`) to verify acquire/release balance.

- **Collision Checks**: Confirmed no trains occupy the same critical section (via TSim visualization and prints).

Correctness: The solution prevents collisions, sequences trains via prioritized upper T-path, and handles reversals correctly.

9 Conclusion

The `Lab1.java` solution is correct due to precise semaphore synchronization, strategic sensor placement, and robust switch handling. The annotated map and debug prints provide clear insights for demonstration.