Lab 1: Trainspotting

This page is available live at Lab 1: Trainspotting. I highly recommend visiting it.

Table of contents

Choice of critical sections

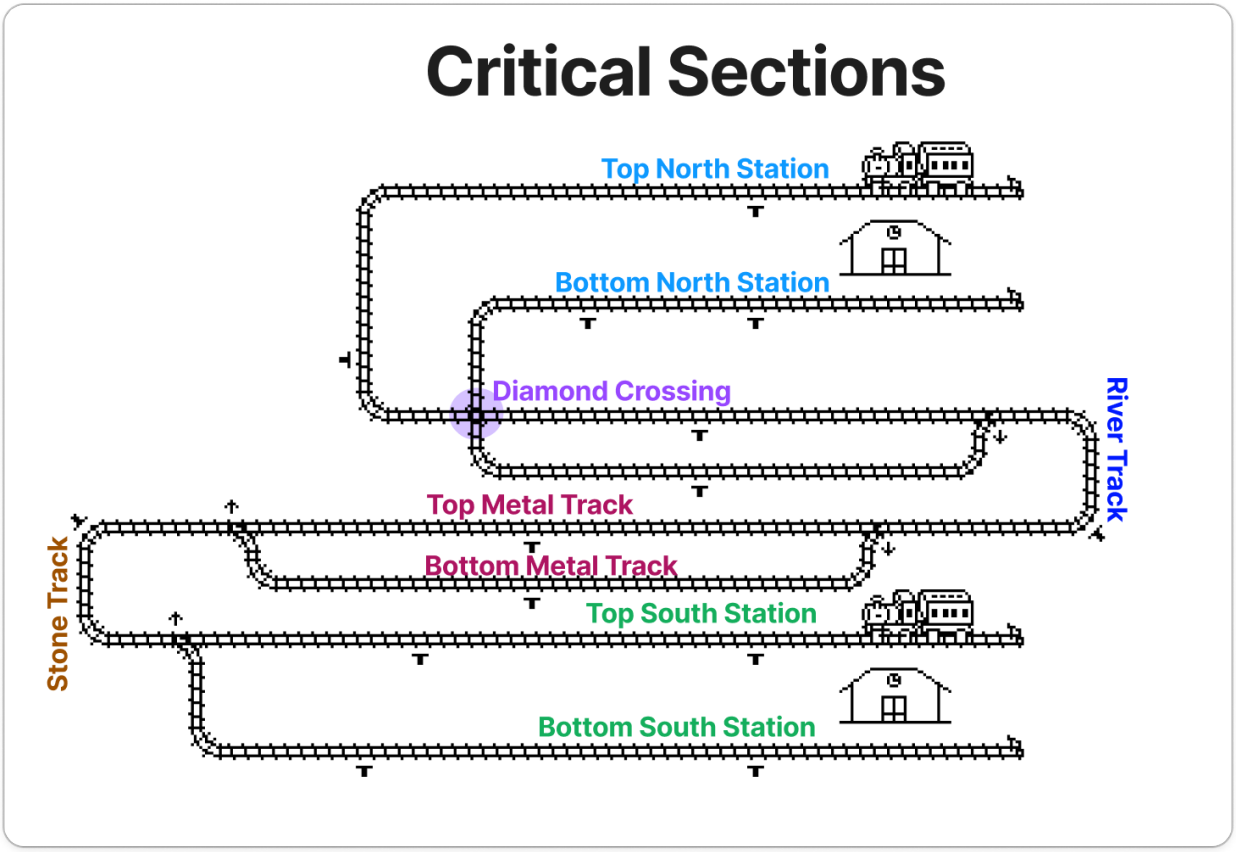

Fig 1. Critical sections.

Critical sections refer to regions where only one train can occupy the area at any given time. With this in mind, the following figure (Fig. 1) highlights the key areas of concern.

Stations

Each station — both North and South — has two tracks. Upon approaching a station, a fork determines whether the train enters the top or bottom track. If one of the tracks is already occupied, the approaching train automatically selects the available track. From a broader perspective, trains never face delays when entering a station because, with only two trains operating, at least one track is always available. This ensures smooth transitions and uninterrupted operations at the stations.

Stone and River Tracks

The Stone and River Tracks are clearly critical sections, where only one train is allowed to pass through at a time. If a train is currently on the Stone Track, any approaching train must wait until it clears the section before entering. The same principle applies to the River Track: a train must wait if another train is already on it, ensuring no collisions occur within these narrow areas.

Metal Track

The Metal Track consists of two parallel lines, which can be accessed from either the left or right forks. These forks are located at the Stone and River Tracks, respectively. Despite the two-track system,

proper coordination ensures no blockages occur at these junction points.

Diamond Crossing

The Diamond Crossing, situated between the Top and Bottom North Stations, is a critical point where the tracks intersect. Naturally, only one train can pass through this crossing at a time to avoid collisions. If a train is currently crossing, any other train must wait until it is clear.

Managing Critical Sections Programmatically

In the code, we handle these critical sections using semaphores to ensure that only one train can occupy a section at any given time. The enum `Track` defines each critical section in the system, including tracks, stations, and the diamond crossing. We initialize an array of semaphores corresponding to each track to control access to these sections.

```
enum Track {
    DIAMOND_CROSSING,
    RIVER_TRACK,
    STONE_TRACK,
    TOP_METAL_TRACK,
    BOTTOM_METAL_TRACK,
    TOP_SOUTH_STATION,
    BOTTOM_SOUTH_STATION,
    TOP_NORTH_STATION,
    BOTTOM_NORTH_STATION
};

Semaphore[] semaphores = new Semaphore[Track.values().length];
```

Placement of the sensors

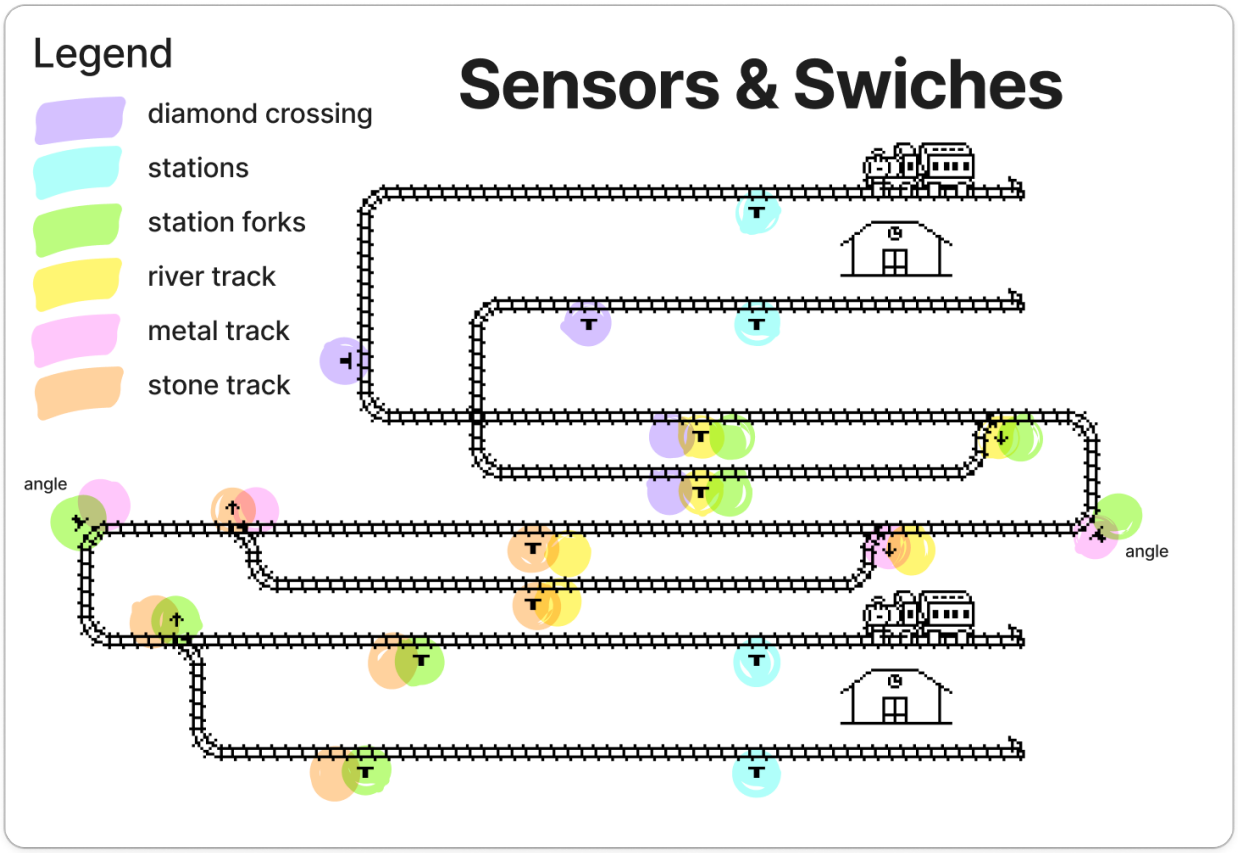

Fig 2. Sensors and switches.

The placement of sensors is critical for the smooth operation of the train system, ensuring that each train can navigate the tracks while following correct directions and safely interacting with switches. The train's interaction with the sensor is monitored using `tsi.getSensor(trainId)`. This method **blocks** the train's thread until it passes over a sensor, giving the train's current position and state (either active or inactive).

Direction Enum

Each train also maintains knowledge of its direction, which is key to understanding its interaction with sensors. For instance, a negative speed for Train 1 indicates it's moving upward (north), and positive speed for Train 2 indicates it's moving upward as well.

```
boolean getDirection() {
  return (id == 1 && speed < 0) || (id == 2 && speed > 0);
}
```

Sensor Case Enum

To further enhance code readability, an `enum` called `SensorCase` is introduced. The `SensorCase` has four states: `UP_BEFORE`, `UP_AFTER`, `DOWN_BEFORE`, and `DOWN_AFTER`. This classification helps determine the train's relative position concerning a sensor and its movement.

```
enum SensorCase {
  UP_BEFORE, UP_AFTER, DOWN_BEFORE, DOWN_AFTER;

  public static SensorCase get(boolean direction, boolean active) {
    if (direction && active) {
      return UP_BEFORE;
    } else if (direction && !active) {
      return UP_AFTER;
    } else if (!direction && active) {
      return DOWN_BEFORE;
    } else {
      return DOWN_AFTER;
    }
  }
}
```

This enum simplifies the logic for determining if the train is moving up and stepping on the sensor (i.e., **before**), or if it has just passed it (**after**). Similarly, it helps to track if a train is moving down and either approaching or passing a sensor.

Teal

The sensors marked with **teal** are located at the stations. They control when a train should stop at a station, pause for a moment, and reverse its direction.
At the
north station, the logic is triggered when a train is moving **up** and the sensor has just been activated (`UP_BEFORE`). At the **south station**, the logic is triggered when the train is moving **down** and the sensor is activated (`DOWN_BEFORE`).

```
// stations
else if (x == 13 && (y == 5 || y == 3)) {
  switch (sensorCase) {
    case UP_BEFORE:
      stopTrain();
      break;
  }
} else if (x == 13 && (y == 11 || y == 13)) {
  switch (sensorCase) {
    case DOWN_BEFORE:
      stopTrain();
      break;
  }
}
```

```
void stopTrain() throws CommandException, InterruptedException {
  tsi.setSpeed(id, 0);
  sleep(1000 + (20 * Math.abs(speed)));
  speed = -speed;
  tsi.setSpeed(id, speed);
}
```

Green

Sensors highlighted in **green** represent station forks. These are responsible for managing the switch settings based on the train's movement and direction. For example, as a train moves down toward the **south station**, it will encounter a sensor at a fork. Here, the train must decide which track to take, and the switch is set accordingly: left for the **top south station** and right for the **bottom south station**.

```
// angles
...
else if (x == 1 && y == 9) {
  switch (sensorCase) {
    ...
    case DOWN_BEFORE:
      if (semaphores[Track.TOP_SOUTH_STATION.ordinal()].tryAcquire()) {
        tsi.setSwitch(3, 11, TSimInterface.SWITCH_LEFT);
      } else {
        semaphores[Track.BOTTOM_SOUTH_STATION.ordinal()].acquire();
        tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT);
      }
            release(metalTrack);
      break;
  }
}
```

When a train leaves one of the station tracks, the south station fork sensors releases the station track:

```
// south station fork
else if (x == 7 && y == 11) {
  switch (sensorCase) {
    case UP_BEFORE:
      acquire(Track.STONE_TRACK);
      release(Track.TOP_SOUTH_STATION);
      tsi.setSwitch(3, 11, TSimInterface.SWITCH_LEFT);
      break;
    ...
  }
} else if (x == 6 && y == 13) {
```

```
  switch (sensorCase) {
    case UP_BEFORE:
      acquire(Track.STONE_TRACK);
      release(Track.BOTTOM_SOUTH_STATION);
      tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT);
      break;
    ...
  }
}
```

Orange

The code snippet above shows the south station fork sensors are also used for acquiring the stone track. Sensors marked in **orange** monitor access to the **stone track**, a critical section that only one train can occupy at a time. The train must acquire the stone track, and if it's unavailable, the train stops.

```
void acquire(Track track) throws CommandException, InterruptedException {
  tsi.setSpeed(id, 0); // get ready to be blocked
  semaphores[track.ordinal()].acquire();
  tsi.setSpeed(id, speed);
}
```

The same logic applies when the train moves from the **metal track** to the **stone track**:

```
// metal track
else if (x == 9 && y == 10) {
  switch (sensorCase) {
    ...
    case DOWN_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.STONE_TRACK);
      tsi.setSwitch(4, 9, TSimInterface.SWITCH_RIGHT);
      break;
  }
} else if (x == 9 && y == 9) {
  switch (sensorCase) {
    ...
    case DOWN_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.STONE_TRACK);
      tsi.setSwitch(4, 9, TSimInterface.SWITCH_LEFT);
      break;
  }
}
```

Stone track is released either on south station fork sensors or metal track sensors:

```
// south station fork
else if (x == 7 && y == 11) {
  switch (sensorCase) {
    ...
    case DOWN_AFTER:
      release(Track.STONE_TRACK);
      break;
  }
} else if (x == 6 && y == 13) {
  switch (sensorCase) {
    ...
    case DOWN_AFTER:
      release(Track.STONE_TRACK);
      break;
  }
}
```

```
// metal track
else if (x == 9 && y == 10) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.STONE_TRACK);
      acquire(Track.RIVER_TRACK);
      tsi.setSwitch(15, 9, TSimInterface.SWITCH_LEFT);
      break;
    ...
  }
} else if (x == 9 && y == 9) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.STONE_TRACK);
      acquire(Track.RIVER_TRACK);
      tsi.setSwitch(15, 9, TSimInterface.SWITCH_RIGHT);
      break;
   ...
  }
}
```

Pink

Sensors and switches marked in **pink** relate to the **metal track**, which can be entered from both directions. Like the stone track, only one train can occupy a metal track at a time. Since there are two

tracks, a train never waits trying to acquire one of the metal tracks. Angle sensors are responsible for acquiring and releasing. The train also saves state `metalTrack` variable to know which of the metal tracks it was when releasing at angle sensors.

```
// angles
else if (x == 19 && y == 9) {
  switch (sensorCase) {
    case UP_BEFORE:
      if (semaphores[Track.TOP_NORTH_STATION.ordinal()].tryAcquire()) {
        tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT);
      } else {
        semaphores[Track.BOTTOM_NORTH_STATION.ordinal()].acquire();
        tsi.setSwitch(17, 7, TSimInterface.SWITCH_LEFT);
      }
      release(metalTrack);
      break;
    case DOWN_BEFORE:
      if (semaphores[Track.TOP_METAL_TRACK.ordinal()].tryAcquire()) {
        tsi.setSwitch(15, 9, TSimInterface.SWITCH_RIGHT);
        metalTrack = Track.TOP_METAL_TRACK;
      } else {
        semaphores[Track.BOTTOM_METAL_TRACK.ordinal()].acquire();
        tsi.setSwitch(15, 9, TSimInterface.SWITCH_LEFT);
        metalTrack = Track.BOTTOM_METAL_TRACK;
      }
      break;
  }
} else if (x == 1 && y == 9) {
  switch (sensorCase) {
    case UP_BEFORE:
      if (semaphores[Track.TOP_METAL_TRACK.ordinal()].tryAcquire()) {
        tsi.setSwitch(4, 9, TSimInterface.SWITCH_LEFT);
        metalTrack = Track.TOP_METAL_TRACK;
      } else {
        semaphores[Track.BOTTOM_METAL_TRACK.ordinal()].acquire();
        tsi.setSwitch(4, 9, TSimInterface.SWITCH_RIGHT);
        metalTrack = Track.BOTTOM_METAL_TRACK;
      }
      break;
    case DOWN_BEFORE:
      if (semaphores[Track.TOP_SOUTH_STATION.ordinal()].tryAcquire()) {
        tsi.setSwitch(3, 11, TSimInterface.SWITCH_LEFT);
      } else {
        semaphores[Track.BOTTOM_SOUTH_STATION.ordinal()].acquire();
        tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT);
```

```
      }
      release(metalTrack);
      break;
  }
}
```

Yellow

Highlighted in **yellow** are sensors associated with the **river track**, functioning similarly to the stone track. River track is acquired and released either on metal track sensors or diamond crossing & north station fork sensors:

```
 // metal track
else if (x == 9 && y == 10) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.STONE_TRACK);
      acquire(Track.RIVER_TRACK);
      tsi.setSwitch(15, 9, TSimInterface.SWITCH_LEFT);
      break;
    case DOWN_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.STONE_TRACK);
      tsi.setSwitch(4, 9, TSimInterface.SWITCH_RIGHT);
      break;
  }
} else if (x == 9 && y == 9) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.STONE_TRACK);
      acquire(Track.RIVER_TRACK);
      tsi.setSwitch(15, 9, TSimInterface.SWITCH_RIGHT);
      break;
    case DOWN_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.STONE_TRACK);
      tsi.setSwitch(4, 9, TSimInterface.SWITCH_LEFT);
      break;
  }
}
```

```
// diamond crossing and north station fork
else if (x == 12 && y == 7) {
  switch (sensorCase) {
```

```
    ...
    case UP_AFTER:
      release(Track.RIVER_TRACK);
      break;
    case DOWN_BEFORE:
      release(Track.DIAMOND_CROSSING);
      acquire(Track.RIVER_TRACK);
      release(Track.TOP_NORTH_STATION);
      tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT);
      break;
  }
} else if (x == 12 && y == 8) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.DIAMOND_CROSSING);
      break;
    ...
    case DOWN_BEFORE:
      acquire(Track.RIVER_TRACK);
      release(Track.BOTTOM_NORTH_STATION);
      tsi.setSwitch(17, 7, TSimInterface.SWITCH_LEFT);
      break;
  }
}
```

Purple

Purple sensors monitor the **diamond crossing**, ensuring smooth passage through this section. The train must acquire and release the crossing properly, allowing only one train through at a time.

```
// diamond crossing
if (x == 6 && y == 6 || x == 10 && y == 5) {
  switch (sensorCase) {
    case UP_AFTER:
      release(Track.DIAMOND_CROSSING);
      break;
    case DOWN_BEFORE:
      acquire(Track.DIAMOND_CROSSING);
      break;
  }
}
// diamond crossing and north station fork
else if (x == 12 && y == 7) {
  switch (sensorCase) {
```

```
    case UP_BEFORE:
      acquire(Track.DIAMOND_CROSSING);
      break;
    ...
    case DOWN_BEFORE:
      release(Track.DIAMOND_CROSSING);
      acquire(Track.RIVER_TRACK);
      release(Track.TOP_NORTH_STATION);
      tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT);
      break;
  }
} else if (x == 12 && y == 8) {
  switch (sensorCase) {
    case UP_BEFORE:
      release(Track.RIVER_TRACK);
      acquire(Track.DIAMOND_CROSSING);
      break;
    case DOWN_AFTER:
      release(Track.DIAMOND_CROSSING);
      break;
    ...
  }
}
```

Note

An analogous situation can be observed for the **north station** as seen with the **south station**, where the sensors marked in **teal** stop the train, allow it to wait, and then reverse its direction when reaching the station.

Similarly, the **north station fork**, marked in **green**, operates in the same way as the **south station fork**. When the train approaches the fork, the sensors help decide which track to acquire, adjusting the switch accordingly (left or right) based on track availability.

Maximum train speed

The maximum train speed was tested and found to be around **25**. The upper speed limit is primarily dictated by the fact that trains do not come to an immediate stop, which can lead to overshooting critical points such as switches or sections where they could potentially collide with another train.

The sensors are placed carefully to make sure trains run smoothly and safely, even when they're going fast. These sensors work well for different train speeds - whether trains are moving slowly, at top speed, or somewhere in between. This smart placement, on one hand, helps high-speed trains slow down in time before they reach important parts of the track. On the other hand, sensors keep performance even for slow trains.

How the solution was tested

The solution was tested manually with the help of tsim simulator and run by the following command:

`make all` && `java -cp bin Main Lab1.map 20 25 15`,

where 20 represents the speed of the first train, 25 represents the speed of the second train, 15 depict the simulation speed.

The simulation was left running for ~1 min at a simulation speed of 2 (not recommended).

The following cases were covered:

speed #1	speed #2
25	25
20	25
25	20
15	20
20	15
15	10
10	15
10	10
10	5
5	10
5	5
25	5
5	25
25	10
10	25

Listing of developed programs

Lab1.java

```
import TSim.*;

import java.util.concurrent.Semaphore;
import static java.lang.Thread.sleep;

public class Lab1 {
  enum Track {
    DIAMOND_CROSSING,
    RIVER_TRACK,
    STONE_TRACK,
```

```
    TOP_METAL_TRACK,
    BOTTOM_METAL_TRACK,
    TOP_SOUTH_STATION,
    BOTTOM_SOUTH_STATION,
    TOP_NORTH_STATION,
    BOTTOM_NORTH_STATION
  };

  Semaphore[] semaphores = new Semaphore[Track.values().length];

  enum SensorCase {
    UP_BEFORE, UP_AFTER, DOWN_BEFORE, DOWN_AFTER;

    public static SensorCase get(boolean direction, boolean active) {
      if (direction && active) {
        return UP_BEFORE;
      } else if (direction && !active) {
        return UP_AFTER;
      } else if (!direction && active) {
        return DOWN_BEFORE;
      } else {
        return DOWN_AFTER;
      }
    }
  }

  public Lab1(int speed1, int speed2) {
    TSimInterface tsi = TSimInterface.getInstance();
    tsi.setDebug(false);

    for (int i = 0; i < Track.values().length; i++) {
      semaphores[i] = new Semaphore(1);
    }

    try {
      tsi.setSpeed(1, speed1);
      tsi.setSpeed(2, speed2);
    } catch (CommandException e) {
      e.printStackTrace();
      System.exit(1);
    }

    class Train implements Runnable {
      int id;
      int speed;
```

```
    Track metalTrack;

    Train(int id, int speed, Track track) {
      this.id = id;
      this.speed = speed;

      // acquire default semaphore
      semaphores[track.ordinal()] = new Semaphore(0);
    }

    boolean getDirection() {
      return (id == 1 && speed < 0) || (id == 2 && speed > 0);
    }

    void acquire(Track track) throws CommandException, InterruptedException {
      tsi.setSpeed(id, 0); // get ready to be blocked
      semaphores[track.ordinal()].acquire();
      tsi.setSpeed(id, speed);
    }

    void release(Track track) {
      semaphores[track.ordinal()].release();
    }

    void stopTrain() throws CommandException, InterruptedException {
      tsi.setSpeed(id, 0);
      sleep(1000 + (20 * Math.abs(speed)));
      speed = -speed;
      tsi.setSpeed(id, speed);
    }

    @SuppressWarnings("incomplete-switch")
    @Override
    public void run() {
      try {
        while (true) {
          SensorEvent sensor = tsi.getSensor(id);

          int x = sensor.getXpos();
          int y = sensor.getYpos();
          SensorCase sensorCase = SensorCase.get(getDirection(), sensor.getSta

          // diamond crossing
          if (x == 6 && y == 6 || x == 10 && y == 5) {
            switch (sensorCase) {
```

```
        case UP_AFTER:
          release(Track.DIAMOND_CROSSING);
          break;
        case DOWN_BEFORE:
          acquire(Track.DIAMOND_CROSSING);
          break;
      }
    }
    // diamond crossing and north station fork
    else if (x == 12 && y == 7) {
      switch (sensorCase) {
        case UP_BEFORE:
          acquire(Track.DIAMOND_CROSSING);
          break;
        case UP_AFTER:
          release(Track.RIVER_TRACK);
          break;
        case DOWN_BEFORE:
          release(Track.DIAMOND_CROSSING);
          acquire(Track.RIVER_TRACK);
          release(Track.TOP_NORTH_STATION);
          tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT);
          break;
      }
    } else if (x == 12 && y == 8) {
      switch (sensorCase) {
        case UP_BEFORE:
          release(Track.RIVER_TRACK);
          acquire(Track.DIAMOND_CROSSING);
          break;
        case DOWN_AFTER:
          release(Track.DIAMOND_CROSSING);
          break;
        case DOWN_BEFORE:
          acquire(Track.RIVER_TRACK);
          release(Track.BOTTOM_NORTH_STATION);
          tsi.setSwitch(17, 7, TSimInterface.SWITCH_LEFT);
          break;
      }
    }
    // south station fork
    else if (x == 7 && y == 11) {
      switch (sensorCase) {
        case UP_BEFORE:
          acquire(Track.STONE_TRACK);
```

```
                release(Track.TOP_SOUTH_STATION);
                tsi.setSwitch(3, 11, TSimInterface.SWITCH_LEFT);
                break;
              case DOWN_AFTER:
                release(Track.STONE_TRACK);
                break;
            }
          } else if (x == 6 && y == 13) {
            switch (sensorCase) {
              case UP_BEFORE:
                acquire(Track.STONE_TRACK);
                release(Track.BOTTOM_SOUTH_STATION);
                tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT);
                break;
              case DOWN_AFTER:
                release(Track.STONE_TRACK);
                break;
            }
          }
          // metal track
          else if (x == 9 && y == 10) {
            switch (sensorCase) {
              case UP_BEFORE:
                release(Track.STONE_TRACK);
                acquire(Track.RIVER_TRACK);
                tsi.setSwitch(15, 9, TSimInterface.SWITCH_LEFT);
                break;
              case DOWN_BEFORE:
                release(Track.RIVER_TRACK);
                acquire(Track.STONE_TRACK);
                tsi.setSwitch(4, 9, TSimInterface.SWITCH_RIGHT);
                break;
            }
          } else if (x == 9 && y == 9) {
            switch (sensorCase) {
              case UP_BEFORE:
                release(Track.STONE_TRACK);
                acquire(Track.RIVER_TRACK);
                tsi.setSwitch(15, 9, TSimInterface.SWITCH_RIGHT);
                break;
              case DOWN_BEFORE:
                release(Track.RIVER_TRACK);
                acquire(Track.STONE_TRACK);
                tsi.setSwitch(4, 9, TSimInterface.SWITCH_LEFT);
                break;
```

```
        }
      }
      // angles
      else if (x == 19 && y == 9) {
        switch (sensorCase) {
          case UP_BEFORE:
            if (semaphores[Track.TOP_NORTH_STATION.ordinal()].tryAcquire()
              tsi.setSwitch(17, 7, TSimInterface.SWITCH_RIGHT);
            } else {
              semaphores[Track.BOTTOM_NORTH_STATION.ordinal()].acquire();
              tsi.setSwitch(17, 7, TSimInterface.SWITCH_LEFT);
            }
            release(metalTrack);
            break;
          case DOWN_BEFORE:
            if (semaphores[Track.TOP_METAL_TRACK.ordinal()].tryAcquire())
              tsi.setSwitch(15, 9, TSimInterface.SWITCH_RIGHT);
              metalTrack = Track.TOP_METAL_TRACK;
            } else {
              semaphores[Track.BOTTOM_METAL_TRACK.ordinal()].acquire();
              tsi.setSwitch(15, 9, TSimInterface.SWITCH_LEFT);
              metalTrack = Track.BOTTOM_METAL_TRACK;
            }
            break;
        }
      } else if (x == 1 && y == 9) {
        switch (sensorCase) {
          case UP_BEFORE:
            if (semaphores[Track.TOP_METAL_TRACK.ordinal()].tryAcquire())
              tsi.setSwitch(4, 9, TSimInterface.SWITCH_LEFT);
              metalTrack = Track.TOP_METAL_TRACK;
            } else {
              semaphores[Track.BOTTOM_METAL_TRACK.ordinal()].acquire();
              tsi.setSwitch(4, 9, TSimInterface.SWITCH_RIGHT);
              metalTrack = Track.BOTTOM_METAL_TRACK;
            }
            break;
          case DOWN_BEFORE:
            if (semaphores[Track.TOP_SOUTH_STATION.ordinal()].tryAcquire()
              tsi.setSwitch(3, 11, TSimInterface.SWITCH_LEFT);
            } else {
              semaphores[Track.BOTTOM_SOUTH_STATION.ordinal()].acquire();
              tsi.setSwitch(3, 11, TSimInterface.SWITCH_RIGHT);
            }
            release(metalTrack);
```

```
                  break;
              }
            }
            // stations
            else if (x == 13 && (y == 5 || y == 3)) {
              switch (sensorCase) {
                case UP_BEFORE:
                  stopTrain();
                  break;
              }
            } else if (x == 13 && (y == 11 || y == 13)) {
              switch (sensorCase) {
                case DOWN_BEFORE:
                  stopTrain();
                  break;
              }
            }
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }

    new Thread(new Train(1, speed1, Track.TOP_NORTH_STATION)).start();
    new Thread(new Train(2, speed2, Track.TOP_SOUTH_STATION)).start();
  }
}
```