CHALMERS

# Lab 1: Trainspotting

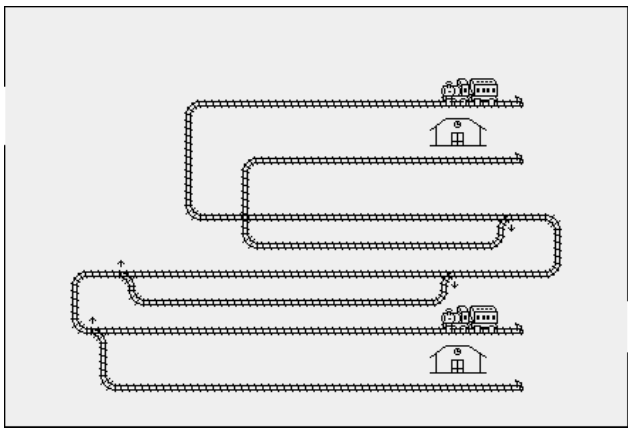
## Deadlines

- See labs page

- To pass this lab you have to give a short demo of your submission to the TAs on 26 September 2019; to do that, you have to sign up here by 25 September 2019, indicating your group number from Fire, to select a time slot. Please try to split evenly among the available rooms. If it is important to you to pick a specific time slot, make sure to sign up early.

## Lab description

In this assignment you will write a program to control two trains. The purpose is to write the program so that the trains move relatively independently of each other, but without colliding. Instead of a real railway, we have a simulator, where one can control the speeds of trains and change rail switches. The simulator gives a signal when a train passes sensors suitably placed on the tracks. It is part of the assignment to decide where sensors should be placed to give sufficient information for your program. The map without sensors looks as follows.



## Programming language and environment

This lab assignment must be developed in Java.

In order to make sure you are using the right versions of Java and the train simulator, you should type the command

```
setup_course tda381
```

on Chalmers GNU/Linux machines.

If you wish to work on this assignment on your own computer, then you will need to build and run the simulator locally. We do

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


not recommend this, since it may not be straightforward, especially if you are using Windows.

See the computing resources page for further instructions.

## Assignment

Your assignment is to write a program that controls the trains. The trains should run simultaneously between the two stations (*note that each station has two tracks*). It is important that the trains are able to move at the same time, and be as independent from each other as possible, without crashing. At the stations the trains should stop and then wait 1 to 2 seconds. The trains should run forever between the stations.

In the beginning there are no sensors on the tracks. You must decide where to place sensors so that the control program gets the necessary information.

The program that you deliver should work as follows. Each train should be controlled by an independent *train program*, which runs in its own Java thread. Independently running train programs are only allowed to communicate with each other by acquiring and releasing semaphores. In particular, no shared variables should be read or changed by the train programs. Also, both trains must use the same *train program*, and the behaviour of a train cannot depend on its train id. For example, the track that a train chooses cannot not depend on its train id.

## Requirements

- Waiting at stations. The trains must wait 1-2 seconds at each station after coming to a full stop. Avoid randomness. You can use a formula such as: `1000 + (20 * |train_speed|)`

- Maximum train speed. You need to find out the maximum train speed `maxspeed` that is usable with your particular solution. This must be at least `15`.

- No minimal speed. There should be no assumption about minimal train speed. All speeds in the range `[1,maxspeed]` will be tried.

- No excessive semaphores. Solutions that use `10` or more semaphores will be rejected.

- No polling. Your solutions must not use *busy-waiting loops* or their close relative, *polling*.

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


- Dynamic behaviour. You must not statically encode how the trains choose track at the switches (for example: train 1 always takes the upper path in the middle, while train 2 always takes the lower; same for stations). You must let the trains have one track as default (for example: the track with shortest distance). You must select the other track if the default track is already occupied by the other train. Note that you should not try to find a solution which works for any track layout or any number of trains, as this will be too complicated for all of us! It is enough to solve the problem for the given track layout.

- No map modifications. You may not alter the map in any way, other than by adding sensors.

- No randomization. You may not use randomization in any way that affects the simulation, as this makes testing more difficult.

- Two trains — two threads. When the system is running there should be only two processes (threads): one for each train. This is in addition to the main thread and the thread that handles communication with the simulator. This latter thread is created in the interface code we provide you with.

- Two trains — one implementation. The solution must use the same implementation for both trains. The only thing separating the processes of the different trains is the train id, starting position, and speed. The fact that the two trains start in different positions means that it is permitted (and necessary) to have different initial behaviour for the two processes, but after that the behaviour of the processes *must not be dependent* on which train it is acting for.

- Use binary semaphores. You must use Java's semaphores (class `java.util.concurrent.Semaphore`) for mutual exclusion; even though those are general semaphores, you must use them as *binary semaphores*. You will have the opportunity to use other synchronization mechanisms in the other assignments.

- Trains mind their own business. Trains should operate independently. A train should not make plans/decisions for the other train. Further, a train should not make plans/decisions for itself based on the information (speed, position, direction) of the other one.

- Good train flow. The parallel section in the middle of the map must allow a fast train to overtake a slow train going in the same direction.

- Documentation. In addition to working code you need to provide a convincing argument why your solution is correct. You must document your solution *with a PDF or plain text file*. Give a high-level description of your program and then go into detail about all pieces of code

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


that are essential for correctness. The documentation should include everything needed for easy understanding of the solution. In particular we demand that your documentation contains discussion of:

- Placement of the sensors
- Choice of critical sections
- Maximum train speed and the reason for it
- How you tested your solution

*Hint:* start by taking a screenshot of your train track setup with the sensors included, then mark the critical sections and other areas you refer to in your code in that screenshot; use the same names to label the entities (critical sections, areas, sensors, and switches) that you use in your code, for easy comparison. You may want to include a copy of this annotated map in the documentation you submit to illustrate your solution.

- Code quality. Reasonable code quality is expected; your assignment might be rejected if we cannot follow your code or if it is too poorly written (use of magic numbers, poor use of types, cut-and-paste coding style, no comments) — even if it works.

- Submission. When you submit your solution, you must provide the following files:
  - `Lab1.java` (your program to control the trains)
  - `Lab1.map` (the placement of your sensors)
  - `documentation.pdf` or `documentation.txt` (your documentation)

## Distinction assignment

In order to be able to get the two distinction points on the first lab, you should modify your solution to work with monitors instead. A detailed description of the distinction assignment can be found here.

## Train simulator

The simulator is called `tsim`; use it as follows:

- `tsim` Make a new, empty map, of default size.

- `tsim filename` Read in an old map from file `filename`.

- `tsim --speed 50 filename` Read in an old map from file `filename`, and also set the simulation speed to `50`. This value is the number of milliseconds that `tsim` waits between updates of its state (i.e moving the trains). `100` ms is the default, so `50` makes the simulation go at double speed.

- `tsim --help` See all options of the simulator.

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises

## Map editor

The train simulator `tsim` also works as a map editor.

You need to get your own copy of the map in order to be able to add the sensors. The map you will be working on is that shown above. The map file is included in the archive in the downloads section.

You modify the map with `tsim Lab1.map`. The figure below shows how to use `tsim`. You should *only* select the sensor button in the tools window (the small T-like symbol) and then use the mouse to place sensors on the track.

Notice that editing the map file manually may lead to unexpected results. In particular, sensors added manually to switch locations do not work.

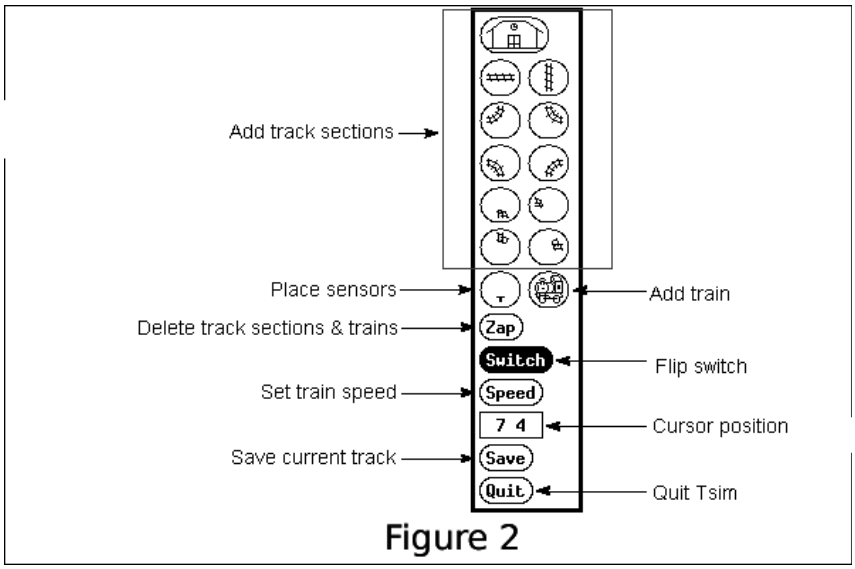


Figure 2

## Simulator

You may start `tsim` and interact with it using commands on standard input. Try for example the commands

```
SetSpeed 1 20
SetSwitch 17 7 RightSwitch
```

and see what happens. (The second command is necessary for the train to survive the first switch which initially is set in the wrong direction). We provide the Java library to handle the communication with `tsim`, which lets you change the speed of the trains and the state of the switches in a more convenient way (see next section).

Here are some things to note about the simulator:

- When you give a new speed for a train, it takes a while for the train to increase or decrease from its old to its new speed. This means that the train's braking distance puts limits on how fast they could drive.

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises

- The trains can not turn around, but can be given a negative speed to travel backwards.

- You are not able to give a speed which implies that a train changes direction. To change direction the train must first be stopped (by giving it speed 0). There is no signal from the simulator to tell you when the train has stopped, so you will need to wait a suitable time before changing direction.

- You must be careful when you change the switches. If a train comes to a switch which is in the wrong position, then the train will derail. See the figure below.

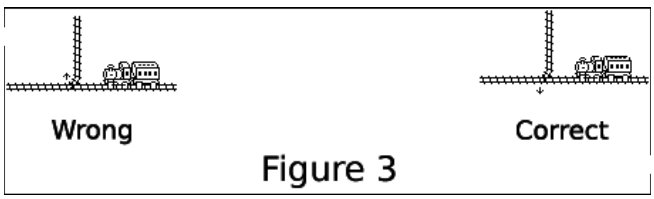


Figure 3

- Switches can be set to either left or right. This direction is relative the fixed end of the switch. I.e., if you were standing at the fixed end of the switch facing the moving end, the switch's left and right would be the same as your left and right. E.g., in the figure above, in the "wrong" configuration the switch is in its left position, and in the "correct" configuration it is in its right position.

## Program/Simulator interaction

The simulator and your program run as separate OS processes. They use pipes to communicate. The `Main` class provided in the skeleton code takes care of joining together the pipes of your solution in Java with the `tsim` simulator.

All commands between the your program and the simulator are sent line-by-line as text strings, as you saw above. To aid you in your assignment we provide you with a Java library (`TSim`) that implements the protocol `tsim` is using. You must use this Java library. The package provides you with an interface to control the behaviour of the trains and switches. You can view the JavaDoc documentation of this library.

### Usage

You cannot create instances of `TSimInterface` using its constructor, since it is private. Instead you use the static method `getInstance()` that creates an instance of the class if that has not already been done (singleton pattern).

```
TSimInterface tsi = TSimInterface.getInstance() ;
```

### Useful classes

- `TSimInterface` is the class used to control the trains. It

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


contains methods to control the speed of the train and to switch the positions of the switches. All these methods throw a `CommandException` if some tsim-related error occurs. In addition there is a method to turn on debugging, which displays the communication between your program and `tsim`. Read the Javadoc documentation of this class carefully!

- SensorEvent represents the event of a train passing over a sensor. Every time a train passes over a sensor first an active event is created followed by an inactive event when the train leaves the sensor. This class contains the status of the sensor, the position of the sensor and the identity of the train causing the event.
- `CommandException` represents a failure of one of the methods of `TSimInterface`. Information on the error is stored in the exception and can be acquired by the `getMessage` method. You should *never* catch an exception leaving the exception handler empty! For your own and our best *always* print the error message contained in the exception object. Otherwise, your program will be much harder to debug.

```
try {
  ...
}
catch (CommandException e) {
  System.err.println(e.getMessage());

  /* If a command failed then something bad has
     happened and we probably don't want to
     continue executing */

  System.exit(1);
}
```

## Download material for the lab

To get started, you need the skeleton code package.

## Getting started

You should implement your solution in the `Lab1.java` file. We provide a `Main` class which takes care of loading the simulator and connecting it to your solution. You shouldn't need to change this file. This entry point expects at least 3 command line arguments:

1. map file
2. speed of train 1
3. speed of train 2
4. simulator speed [optional]

Using the command line

To build your solution, you can simply run the following command:


```
make all
```

from the root directory of the project. This will place the class files in the `bin` folder.

You can then run your solution with:

```
java -cp bin Main Lab1.map 5 15
```

Using Eclipse

You can also open the files as an Eclipse project:

1. Extract the `trainspotting` folder into your workspace directory.
2. In Eclipse, choose New → Java Project. Give it the exact name `trainspotting` to use the existing sources.

When running or debugging the `Main` class in Eclipse, you will need to specify the command-line arguments via the Run Configurations dialog.

Eclipse should be installed on all the StuDAT-linux workstations in the lab rooms but may not be available in the startup menu. To start it, open a terminal window and run

```
eclipse &
```

## Tips and tricks

- Make sure you understand which parts of the tracks are critical resources.
- There are two common ways of writing the train programs. One is that the train remembers in which place on the track it is and performs its actions based on that (stateful solution). The other is that the train decides what to do based on the coordinates of the sensor it steps on (stateless solution). In the stateless solution, you might still have to remember some things, for example which semaphors you are holding.
- Do not try to solve the general problem. Create a solution that works for this particular exercise. For example, it is OK to assume that there will be only 2 trains on the map. If you are interested in solving the general problem, write a short description of how to solve it in the documentation file.
- If you lock two critical sections at the same time, make sure that this does not negatively affect the train flow.
- *Hint:* using around 16 sensors is a suitable number to guarantee both correctness and good flow; while there is some flexibility, if you end up needing *many* more or

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


*many* less sensors you'd probably better revise your solution.

- It is not necessary for trains to stop exactly at the end of the track at a station (before changing direction).
- Do not overoptimize modularity. The assignment is simple enough to have the solution in one file.
- For testing, run multiple instances of the simulator (at a high simulation speed), with different train speeds for a long time. e.g. Test your trains with the first going very fast and the second going very slow, and vice versa. Don't set the simulation speed too high (`--speed` switch should be about 15 or higher), since the trains might not get commands from your program on time.
- Note that `get_sensor(TId)` blocks until the given train (`TId`) next passes a sensor.
- In previous courses, the best solutions had usually less than 300 lines of code. If your solution is much bigger than that then you are probably making it too complicated. Try making it simpler. It is very important that your code is understandable.

## Important

It was stated above but may be stated again: reasonable code quality is expected. You are programmers. Programming should be an art to you. The quality of your code will be vital for its reuse and maintenance.

## Common reasons for rejection

Using *polling/busy waiting* for synchronization is a common mistake that leads to submissions being *rejected*.

Here are some examples of polling/busy waiting in pseudo code. Loops that behave similarly to the situations below (where the dots do not include any *blocking wait*) are considered as polling.

```
while (e) { // POLLING!
   ...
   sleep(t);
   ...
}
```

Using a blocking operation within a loop is not considered as polling

```
while (e) { // NO POLLING!
   ...
   wait(o);
   ...
}
```

provided that it is not the case that that the waiting process is

Principles of Concurrent Programming HT19

TDA384/DIT391

Canvas room (CTH)
Canvas room (GU)
Home
Information
Lectures
Labs
Reading material
Computing resources
Exam
Exercises


woken up from its wait at regular intervals. Thus, the following example is also an instance of polling:

```
obj o;

process a {
  while (true) {
    sleep(t);
    notify(o);
  }
}

process b {  // POLLING!
  while (e) {
    wait(o);
  }
}
```