DELFT UNIVERSITY OF TECHNOLOGY
Faculty of Electrical Engineering, Mathematics and Computer Science

TU Delft

CSE1100 Object-Oriented Programming (computer exam)
Mock exam week 1.6, 2 hours (2:00)

Created by A. Zaidman and F. Mulder

STEP 1: Log into the computer with your personal netID/password combination.

STEP 2: Start up Eclipse (this might take a few minutes) → STEP 3.

STEP 3: Carefully read the assignment for this mock exam.

You can make use of the following during this exam:

- One book on Java, e.g., Java in Two Semesters (Charatan & Kans) or Introduction to Java Programming and Data Structures (Lang).
- The slides are available in PDF format via Brightspace
- The Java API document (javadoc) is available through a link from Brightspace

This exam contains **1 assignment (10 points)** (total exam: 5 pages).

HINT 1:	**Read the entire assignment and only then start implementing**
HINT 2:	**Copy the file trains.txt from Brightspace into your Eclipse project folder. Put it in the root of your project (so not /src).**
HINT 3:	**Look at the last page to get an overview of how your score is built up for this exam. We will not score this mock exam for you, but this overview provides perspective on how well you are doing.**
HINT 4:	**It might be that JUnit 4.0 (or 5.0) is not in your Eclipse build path by default. If you add a unit test case through the "New" JUnit wizard, then Eclipse will notice that JUnit is not in the build path and Eclipse will direct you to the build path menu. Go to the "Libraries" tab, click "Add Library" and Eclipse will suggest to add JUnit.**
HINT 5:	**At the end of the exam, you can take your solution with you (e.g., email it to yourself)**
HINT 6:	**While Google is available to you during this mock exam, it will not be during the actual exam… (so if you want to experience "the real deal", don't use any internet!)**

A few extra hints:

- Your program must **compile** (fail to compile == fail this exam)
- As part of the mock exam exercise, when your program is finished, use the 7Zip progam to zip your files. Specifically, make a **zip file of your src folder** (the folder that contains your .java files) when your assignment is ready. Give the zip file the following name <studentnumber>.zip, so for example 12121212.zip. Please also put the inputfile into this zip.
  → upload your zip file to Brightspace: Content > Mock Exam > Upload your assignment
- Please, do not use specific packages but rather use the **default package** (this makes correcting the exam that much easier for us) → if you do use a specific package, you will lose 1 point.

ProRail is the company that is responsible for the railway infrastructure in the Netherlands. They are in dire need of a new software system. They want to keep track of which trains are on the tracks on a certain day to get more insight into the availability and usage of the railway tracks. ProRail has contacted you to make a first prototype of this application. ProRail is well aware that you can only produce a prototype in such a short amount of time, but they will still need some time to finish some of the extra requirements.

Specifically, ProRail wants you to develop an application that reads in a file that specifies which trains are present on the tracks.

An example of such a file looks as follows:

```
TRAIN: INTERCITY
DEPARTS: 14:20
ARRIVES: 15:00
IC-STATION: Rotterdam Centraal
IC-STATION: Schiedam Centrum
STATION: Delft Zuid
IC-STATION: Delft
STATION: Rijswijk
STATION: Den Haag Moerwijk
IC-STATION: Den Haag HS
END
TRAIN: SPRINTER
DEPARTS: 14:30
ARRIVES: 15:20
IC-STATION: Rotterdam Centraal
IC-STATION: Schiedam Centrum
STATION: Delft Zuid
IC-STATION: Delft
STATION: Rijswijk
STATION: Den Haag Moerwijk
IC-STATION: Den Haag HS
END
```

The complete file **trains.txt** is available on Brightspace.

The order of the lines of a train specification is fixed. You will always first get the departure and arrival times and then a sequence of stations along the route of the train. Also important to know: each piece of information is always on a new line.

A **Sprinter** is characterised by:
- The time of departure
- The time of arrival
- A list of stations that are part of the route of this train. The train stops at each station on this route.

An **Intercity** is characterised by:
- The time of departure
- The time of arrival
- A list of stations that are part of the route of this train. There are IC-stations in this list and regular stations. This type of train stops at the begin/end station and all intermediate IC-stations, while normal stations on the route are just there to show the route (the train doesn't stop at these).

ProRail asks you to design and implement a program that:
- **Reads in** the file trains.txt

  Hint:

  ```
  Scanner input = new Scanner(new File("trains.txt"));
  ```
  → this works if the file trains.txt is in the root folder of your Eclipse project

  ```
  String line = input.nextLine();  // read line
  String word = input.next();      // read next token or word
  int number = input.nextInt();    // read next int
  ```

  What if you want to "scan" a line that you've just read in?
  ```
  String line = input.nextLine();
  Scanner scanline = new Scanner(line).useDelimiter(":");
  ```
  → now you scan the line, reading words separated by a ":"

- Allows to **list** all trains that have been read
- Allows to **list** all trains that ride from station A to station B (irrespective of the intermediate stations on the route).
- Write an **equals()** method for each class (except for the class that contains the main() method). Two trains should be equal only if their type of train, route followed and departure and arrival times match.
- To enable user interaction, please provide a **command line interface** (reading from `System.in` and writing to `System.out`). This interface should look like:

```
Please make your choice:
   1 – Show all trains that are in the in-memory database on screen
   2 – Show all trains from station A to station B
   3 – Stop the program
```

Option 1:
All trains are shown on screen in the following format:

```
Intercity from Den Haag HS to Rotterdam Centraal
   Departure: 13:20
   Arrival: 14:00
   ---Den Haag HS
   ----(Den Haag Moerwijk)
   ----(Rijswijk)
   ---Delft
   ----(Delft Zuid)
   ---Schiedam Centrum
   ---Rotterdam
```

This output format makes a distinction between stations that are on the route (e.g. Rijswijk) and a station where the train stops (e.g. Delft). Please take into consideration:

- An Intercity stops at the start and stop station and at IC-stations
- A Sprinter stops at all stations on the route

Option 2

- You ask the user to provide a departure and arrival station. Use System.in here!
- Show the result of all trains that satisfy the conditions (and show it in a similar way to Option 1). The condition being the same start and end station, so the two trains in the example file listed above would be returned in case the user enters "Rotterdam Centraal" and "Den Haag HS".

Option 3
The application stops.

Some important things to consider for this assignment:

- The textual information should be read into a class containing the right attributes to store the data (so storing all information in a single `String` is not allowed, because this would hinder further development). With regard to the type of attributes used (`int`, `double`, `String`, etc.): you decide!
- Think about the usefulness of applying inheritance.
- The **filename trains.txt should not be hardcoded** in your Java program. Please make sure to let the user provide it when starting the program (either as an explicit question to the user or as a program argument)
- Write unit tests!

Other things to consider:

- The program should **compile**
- For a good grade, your program should also work well, without exceptions. Take care to have a nice **programming style**, in other words make use of code indentation, whitespaces, logical identifier names, etc. Also provide javadoc comments.

What key things will determine your grade? [1]

- 3 for compilation; if it doesn't compile → final score = 1
- 1.3 for a well-thought-out application of inheritance
    - Are you using inheritance and polymorphism in the right way, and is the functionality correctly distributed over the inheritance hierarchy?
- 0.8 for correctly implementing equals() methods
- 1.4 for implementing reading in a file (functioning code that leads to exception still gives part of the score)
- 0.6 for nicely styled code. Aspects being considered:
    - Length and complexity of methods
    - Length of parameter lists
    - Well-chosen identifier names
    - Whitespace, indentation, …
- 0.8 for a well-working textual interface (including option 1, which prints all trains to screen)
- 0.6 for not hardcoding the filename
- 1.5 for JUnit tests
    - 0.75 for testing the class Train (depending on how well you test, you get a score between 0.0 and 0.75)
    - 0.75 for testing all other classes (except the class that contains main(), you also get a score between 0.0 and 0.75 depending on how well you test)
    - Do not use files in your tests! (You do not have to test the method with which you read the file trains.txt.)

There is a 1 point deduction if you do not work in the default package. So work in the default package!

[1] Sub-scores add up to 10 and have been scaled to be realistic with regard to a real exam. Mind you, the real exam will require extra functionality to be created and will also have additional grading criteria. See a previous old exam for details.