Assignment 1

Trains

Romano Asciutto (500801794)
Jesse van Bree (500801418)
Class IS201

26th September, 2019

Table of contents

Table of contents **1**

1 Introduction **2**

2 Development **3**

2.1 Classes **3**

2.1.1 Locomotive 3

2.1.2 Train 3

2.1.3 Wagon 4

2.1.4 PassengerWagon 4

2.1.5 FreightWagon 4

2.1.6 Shunter 4

2.2 Most important code **5**

2.2.1 Attaching wagon to wagon 5

2.2.2 Attaching wagon to train rear 6

2.2.3 Number of wagons attached 7

2.2.4 Last wagon attached 8

2.3 Output **9**

2.3.1 TrainLauncher 9

2.3.2 Testing 10

3 Conclusion **11**

1 Introduction

In this assignment, we have built an application in which you can build a train consisting of equally typed wagons. Actions can be performed on the train and its wagons by a shunter. This shunter has different functionalities implemented to attach, detach and move wagons around. By building this application, we have been able to implement a custom linked list and recursive methods to iterate through it.

The purpose of this report is to go over our Train assignment and present our solution. By doing so, we can reflect on how we came to this solution. We can then use this knowledge to improve our efficiency and problem solving skills in future programming projects.

We will begin by giving a short description of each class, what its purpose is and how they work together. This will give an overall view of the structure of our application. With this discussed, we will go over the most important code in the project and describe its functionality along with code snippets. We will then show the output of our code and unit tests.

At the end of this report is a conclusion in which we will reflect on this assignment and discuss the methods we used to solve the problem. This will give an idea of how we can use this project as a learning experience for further assignments.

2 Development

In this chapter we will dive into the code of our project. We will discuss the following items:

- The core classes of the assignment with its descriptions.
- The most important code of this project.
- The output of the program.

2.1 Classes

In this section we will describe the classes that have been used in this assignment and how they work together.

2.1.1 Locomotive

The locomotive class stores the maximum amount of wagons that the train can pull. This number can be used in the part of the application where the amount of space left on the train is checked to make sure attaching another wagon will not exceed the maximum capacity of the train.

2.1.2 Train

The train class stores the engine, the first wagon, strings of the origin and destination and the current number of wagons attached. WIth this information, it is possible to iterate through all the wagons and update the number of wagons if changes are applied by the shunter. This class allows for different actions on finding specific wagons and is mainly used to get the first wagon, which is needed by most methods in the application.

2.1.3 Wagon

A single wagon instance is a specific link in a chain of wagons, it only stores which wagon is previous and next to it. With this information, it is possible to recursively iterate through the chain of wagons. We have implemented a way to find the end of the chain and also to find the total number of wagons attached. This is all the information needed by the rest of the application to perform actions on the chain of wagons.

2.1.4 PassengerWagon

The PassengerWagon is a concrete extension of the abstract Wagon class. The way this type of wagon deviates is in its number of seats, this is specific to this type of wagon alone. A train may only consist of one type of wagons and this specific implementation allows for type checking in other parts of the code where wagons are attached.

2.1.5 FreightWagon

The FreightWagon is another concrete extension of the abstract Wagon class. This time, it stores the maximum weight of the wagon. As soon as this type of wagon is attached, the rest may only be of this type. This specific instance allows for type checking in other parts of the application.

2.1.6 Shunter

The shunter is a helper class to perform different actions on a train. It consists of static methods so an instance doesn’t have to be created because it also doesn’t have any properties. Examples of the actions are attaching and detaching wagons from a train, hooking wagons on wagons and checking if a wagon is suitable to be attached.

2.2 Most important code

In this chapter we will disclose the most important code in the train application. With some code snippets and an explanation we will explain why this is. The methods we find the most important are: hookWagonOnWagon, hookWagonOnTrainRear, getNumberOfWagonsAttached and getLastWagonAttached.

2.2.1 Attaching wagon to wagon

The method hookWagonOnWagon is important to be able to attach wagons directly to a wagon without bothering with the train. This method is also really helpful in inserting a wagon within any position in the list of wagons.

The method first checks if the type of both wagons are the same(FreightWagon or PassengerWagon). Then if the second wagon is already connected to previous wagon it will disconnect them. After the method will check if the wagon has a next wagon attached. It will then check if the first wagon also has more wagons attached. If so the method will attach the wagons from second onto first and attach the wagons attached to first to the back of the last wagon attached to second. If this is not the case it will simply attach second to first. Based on the first if statement the method will return true or false.

```java
/**
* Hooks a wagon to another wagon if they are of the same type
* @param first wagon to attach to
* @param second wagon to attach
* @return if the method is successful
*/
public static boolean hookWagonOnWagon(Wagon first, Wagon second) {
   if(isSuitableWagon(first, second)){
      if(second.hasPreviousWagon()){
          second.getPreviousWagon().setNextWagon(null);
      }
      if(second.hasNextWagon()){
          if(first.hasNextWagon()) {
              Wagon lastWagon = second.getLastWagonAttached();
              lastWagon.setNextWagon(first.getNextWagon());
              first.getNextWagon().setPreviousWagon(lastWagon);
              first.getLastWagonAttached().setNextWagon(null);
          }
      } else if(first.hasNextWagon()){
          Wagon oldSecond = first.getNextWagon();
          oldSecond.setPreviousWagon(second);
          second.setNextWagon(oldSecond);
      }
      first.setNextWagon(second);
      second.setPreviousWagon(first);
```

```
        return true;
    }else {
        return false;
    }
}
```

2.2.2 Attaching wagon to train rear

Having a method that attaches a wagon to the rear of a train is also very nice to have. This makes it very easy to attach a wagon or a series of wagons to the train.

The method first checks if there is space on the train and checks if the type of wagon is the same as the type of wagons on the train. Then it checks if there are wagons attached to the train. If not then its sets the wagon as the first wagon in the train. If there are wagons attached to the train, the wagon will be attached to the last wagon of the train.

```
/**
* Hooks the wagon to the back of the train
* @param train train to attach to
* @param wagon wagon to attach
* @return if the method is successful
*/
public static boolean hookWagonOnTrainRear(Train train, Wagon wagon) {
    if (hasPlaceForWagons(train, wagon) && isSuitableWagon(train, wagon)) {
        Wagon currentWagon = train.getFirstWagon();
        if(currentWagon == null){
            train.setFirstWagon(wagon);
            wagon.setPreviousWagon(null);
        } else {
            currentWagon.getLastWagonAttached().setNextWagon(wagon);
        }
        train.resetNumberOfWagons();
        return true;
    }
    return false;
}
```

2.2.3 Number of wagons attached

It's very important to know how many wagons are attached to a wagon. This is because a locomotive has a max pull limit regarding the amount of wagons.

The method is written recursively. This means that the method calls itself and sets its value onto a stack. This stack grows until the wagon parameter has no wagons attached to the back. It then counts how many wagons are attached based on the count parameter and returns this value.

```java
/**
* Recursive method that gets the number of wagons attached to the given wagon
* @param count Amount of wagons
* @param wagon wagon to count with
* @return int the number of wagons attached to wagon
*/
public int getNumberOfWagonsAttached(int count, Wagon wagon) {
   if (wagon.hasNextWagon()) {
       return getNumberOfWagonsAttached(++count, wagon.getNextWagon());
   } else {
       return count;
   }
}
```

2.2.4 Last wagon attached

This method is important to be able to attach wagons onto the rear of the train, or to find out what the last wagon attached is.

This method is like getNumberOfWagonsAttached also a recursive method. It checks if the wagon parameter has a next wagon attached. If so it will call itself again with the next wagon in the chain. Until the last wagon in the chain is reached. It will then empty the stack by eventually returning the last wagon to the original call.

```
/**
* Gets the last wagon in the chain of wagons
* @param wagon the wagon that starts in the chain
* @return Wagon  the last wagon in the chain
*/
public Wagon getLastWagonAttached(Wagon wagon)
   if (wagon.hasNextWagon()) {
       return getLastWagonAttached(wagon.nextWagon);
   } else {
       return wagon;
   }
}
```

2.3 Output

In this chapter we will show the output of the program with an explanation, including the pre written tests and the extra tests we added.

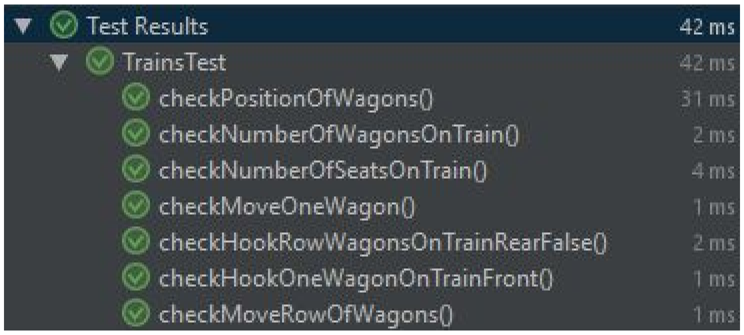
2.3.1 TrainLauncher

This is the output of the TrainLauncher, in which wagons are attached to the front and back of the train and then moved around to mainly test the functionality of the helper methods in the Shunter class. This is convenient for generating an output of the tests with console prints and also helped us in quickly testing certain parts of the application without the need to build a unit test right away.

```
{Loc 2453}[Wagon 3][Wagon 24][Wagon 17][Wagon 32][Wagon 38] with 5 wagons and 630 seats from Amsterdam to Haarlem

{Loc 2453}[Wagon 21][Wagon 24][Wagon 17][Wagon 32][Wagon 38][Wagon 3] with 6 wagons and 770 seats from Amsterdam to Haarlem

{Loc 2453}[Wagon 21][Wagon 24][Wagon 17][Wagon 32][Wagon 38][Wagon 3] with 6 wagons and 770 seats from Amsterdam to Haarlem

Position of Wagon 21 is: 1

Wagon on position 5 is: [Wagon 38]

----------------------

{Loc 2453}[Wagon 21][Wagon 24][Wagon 17][Wagon 38][Wagon 3] with 5 wagons and 620 seats from Amsterdam to Haarlem

{Loc 5277}[Wagon 32] with 1 wagons and 150 seats from Cape Town to Joburg

----------------------

{Loc 2453}[Wagon 21][Wagon 24] with 2 wagons and 240 seats from Amsterdam to Haarlem

{Loc 5277}[Wagon 32][Wagon 17][Wagon 38][Wagon 3] with 4 wagons and 530 seats from Cape Town to Joburg
```

2.3.2 Testing

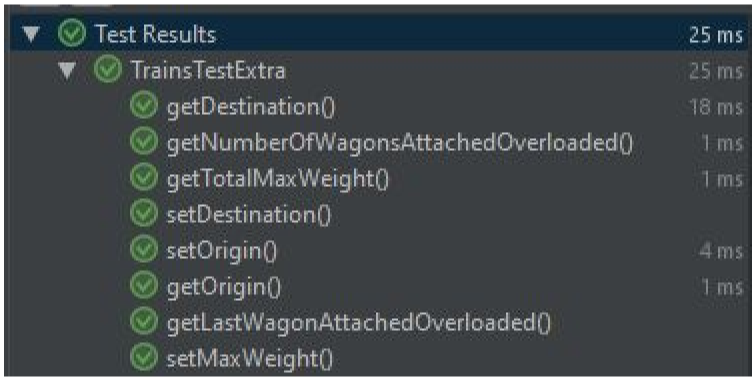
To test our application, JUnit was used to build unit tests. Below is the given test class for the assignment in which the methods of the Shunter class are tested. We were able to pass all given unit tests.



```
▼ Test Results                              42 ms
  ▼ TrainsTest                              42 ms
      checkPositionOfWagons()               31 ms
      checkNumberOfWagonsOnTrain()           2 ms
      checkNumberOfSeatsOnTrain()            4 ms
      checkMoveOneWagon()                    1 ms
      checkHookRowWagonsOnTrainRearFalse()   2 ms
      checkHookOneWagonOnTrainFront()        1 ms
      checkMoveRowOfWagons()                 1 ms
```

Below is a test class we have added ourselves, separate from the given class of the assignment. The two recursive methods: getNumberOfWagonsAttached and getLastWagonAttached were already tested above when attaching wagons to the train. What is different here is that we test our implementation of method overloading.

These two methods require parameters to recursively call itself with new data, but outside of this method these parameters are not required. We therefore use method overloading to allow for the methods to be called without any given parameters. To make sure this functions correctly, we have written tests for it as well as can be seen below along with new methods we created in the Train class.



```
▼ Test Results                                25 ms
  ▼ TrainsTestExtra                           25 ms
      getDestination()                        18 ms
      getNumberOfWagonsAttachedOverloaded()    1 ms
      getTotalMaxWeight()                      1 ms
      setDestination()
      setOrigin()                              4 ms
      getOrigin()                              1 ms
      getLastWagonAttachedOverloaded()
      setMaxWeight()
```

3 Conclusion

What we learned by making this assignment, is how to implement a linked list and what the advantages and disadvantages are compared to an array or arraylist. Next we learned how to implement the Iterable interface on a linked list which is a bit different than how we implement it on an arraylist. To iterate over the wagons from within the wagon class itself, we have used recursion for a method to recursively call itself until the end of the linked list is reached.

This concludes the report about the trains assignment 1.