

Project 2: Linked List Murders

DUE: Sunday, Sept 27th at 11:59pm (Extra Credit for Early Submission!)

Setup

- Download the `project2.zip` and unzip it. This will create a folder `section-yourGMUUserName-p2`.
- Rename the folder replacing `section` with the `001, 003, 004, 005` based on the lecture section you are in.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address.
- After renaming, your folder should be named something like: `001-krusselc-p2`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your `readme.txt`
- Zip your user folder (not just the files) and name the zip `section-username-p2.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
 - The submitted file should look something like this:


```
001-krusselc-p2.zip --> 001-krusselc-p2 --> JavaFile1.java
                                     JavaFile2.java
                                     JavaFile3.java
                                     ...
```
- Submit to blackboard. **DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED IS THE RIGHT THING. Submitting the wrong files will result in a 0 on the assignment!**

Basic Procedures

You must:

- Have code that compiles with the command: `javac *.java` in your user directory WITHOUT errors or warnings.
- Have code that runs with the command:


```
java Simulator [TrainNotes] [PeopleNotes] [CarOfMurder] [TimeOfMurder]
```

You may:

- Add additional methods and variables, however these methods **must be private**.
- Add additional nested, local, or anonymous classes, but any nested classes must be private.

You may NOT:

- Add additional *public* methods, variables, or classes. You may have public methods in private nested classes.
- Use any built in Java Collections Framework classes in your program (e.g. no `LinkedList`, `ArrayList`, etc.).
- Add any additional import statements (or use the “fully qualified name” to get around adding import statements).
- Create any arrays anywhere in your program. Using arrays, in any way, will result in a 0 on this assignment (the goal is to learn linked lists!). The only exception to this is for handing a `String[]` in `setupInitialTrainPositions()` when splitting the line.
- Make your program part of a package.
- Alter provided classes or methods that are complete (e.g. most of `Simulator`).
- Alter any method signatures defined in this document of the template code. Note: “throws” is part of the method signature in Java, don’t add/remove these.
- Change any code in any file below the “DO NOT EDIT BELOW THIS LINE” line. You still must add comments to any methods and instance variables that do not have comments already.
- Add any additional libraries/packages which require downloading from the internet.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Where Should I Start?

This may seem like a very large project given the size of this document, but your part is actually very simple. Below is an ordered checklist for working on this project (after you've read this document in full):

- ☐ Read this document, in full. I know it's long, but Page 1 is a reference, Page 2 is this checklist, and Pages 5-7 and 10-13 are sample runs for testing. So that's really 4 pages of actual description.
- ☐ Trace all three scenarios by hand (make sure you understand what is happening). Pages 5-7 and 10-13.
- ☐ Comment ALL the code with JavaDocs (once you've done this, you'll know how all the parts fit together). If you leave this to the end you'll be *very* unhappy.
- ☐ Complete the **Car** class – This class can be written fairly quickly and should be more or less review from CS211 (each car is just a doubly-linked *node*).
- ☐ Run the **Car** class main method (and finish debugging this before you continue).
- ☐ **Person** class – This class is also fairly simple and only needs to know about the **Car** class to work.
- ☐ Run the **Person** class main method (and finish debugging this before you continue).
- ☐ **Train** class – This class requires the **Car** class and is a little more advanced (you'll need to make an iterator for the cars connected to the train). Remember: the cars are linked list nodes, so **Train** is essentially a linked list with a name (e.g. T0). Head and tail pointers would be useful here.
- ☐ Run the **Train** class main method (and finish debugging this before you continue).
- ☐ **UniqueList** – This class is a little more advanced and will require you to write a linked list with different properties than the train class. Take your time on this and debug it separately.
- ☐ Run the **UniqueList** class main method (and finish debugging this before you continue).
- ☐ **Pair** – This class is fairly easy, but it is only important for the **UniquePairList**, so you can leave it off until you're ready to work on the simulator.
- ☐ Run the **UniquePairList** class main method (and finish debugging this before you continue).
- ☐ Implement `setupInitialTrainPositions()` in **Simulator**.
- ☐ You *might* be done! Run the sample scenarios included with this assignment and compare your output to the sample output (the command used to generate the output is the first line at the top of each sample run file). If something looks wrong... debug, debug, debug.

TL;DR: Use the order above to write your project or you'll get very muddled

Overview: Murder Mystery Introduction

There has been a murder on a train! The police know what day it happened, but they are not quite sure when it happened. Additionally, they aren't quite sure they have correct statements from everyone. You and your professor have been asked to help the police by recreating the events on the day leading up to the murder. You will base your simulation of notes from the two detectives working on the case: Fox and Alleyn.

Your professor is an expert at reading in weird formatted notes and creating amazing ASCII art visualizations, but wants your help in "modeling" the trains and the people on them. Additionally, your professor would like you to provide helper data structures that can be used to quickly look up things by name.

To get you started, let's introduce you to the two detectives working the case and what they've done so far.

TL;DR: You're "helping" to simulate a bunch of people moving around on trains before a "murder".

Fox's Investigation: The Trains

Fox has contacted the train authority and took meticulous notes on how the trains moved on the day of the murder. It turns out that there were actually several trains which connected and disconnected cars throughout the day. His notes run something like the following:

Train 0 started with a single car (car 2).

Train 1 started with three cars (car 0, 1, and 3).

At 01:17 train 1 disconnected car 3, then disconnected car 1. At the same time train 0 disconnected car 2.

At 04:37 train 0 connected car 1, then car 3.

...
...

Your very wise professor has asked Fox to transcribe these notes so that they can be more easily read by a computer. Below are the same statements in shorthand:

T0 C2

T1 C0 C1 C3

01:17

T1 disconnect C3

T1 disconnect C1

T0 disconnect C2

04:37

T0 connect C1

T0 connect C3

...
...

TL;DR: There are files that describe the train movements.

Alleyn's Investigation: The People

While the trains were connecting and disconnecting cars, people were also moving around inside the cars. Alleyn, after extensive interviews, has compiled his notes. Originally they looked something like:

At the start of the day, car 0 contained person 3 only; car 1 contained person 1; car 2 contained person 0, 2, and 4; and finally car 3 contained person 5.

At 05:37 person 5 moved from car 3 to car 1 and person 1 moved from car 1 to car 3 (they passed each other).

At 05:49, security footage shows person 5 returned to car 3.

...

But after discussions with your professor, Alleyn also created a shorthand which is easier for the computer to read:

```
P0 C2
P1 C1
P2 C2
P3 C0
P4 C2
P5 C3

05:37
P5 C3 C1
P1 C1 C3

05:49
P5 C1 C3
...
```

TL;DR: There are files that describe the people movements.

The police have asked you to try running simulations for several scenarios to make sure your program works. Three sample scenarios have been provided for you in your user directory, along with a sample run of each scenario for testing purposes. These scenarios are written in the format shown above. There are also two additional files in each directory, one shows the final positions of the cars on the trains at midnight, the other shows the final positions of the people at midnight.

The next section will walk you through a partial run of scenario 2. It is highly recommended that you trace through all three scenarios *by hand* before you attempt to program anything.

TL;DR: There are sample scenarios which you can trace.

Simulation Introduction

Let's run through a quick scenario first to make sure you understand what is going to happen when you're done:

```
>java Simulator ./scenario2/trains-scenario2.txt ./scenario2/people-scenario2.txt C6 06:15
```

Detective's Shorthand Notes	Simulation Output
<p>T0 C3 C5 T1 C0 C1 C2 C4 C8 C9 T2 C6 C7</p> <p>P0 C2 P1 C7 P2 C6 P3 C7</p> <div> <p>Notes:</p> <p>These are the initial positions of the cars on the train and the people in the cars. They appear at the start of the files.</p> </div>	<p>00:00</p> <p>Trains:</p> <p>T0 o o _ _ _ o C3 C5 _ _ _ - _ _ _ - _ _ _ /0-0-0 o o o o</p> <p>T1 o o _ _ _ o C0 C1 C2 C4 C8 C9 _ _ _ - _ _ _ - _ _ _ - _ _ _ - _ _ _ - _ _ _ - _ _ _ /0-0-0 o o o o o o o o o o</p> <p>T2 o o _ _ _ o C6 C7 _ _ _ - _ _ _ - _ _ _ /0-0-0 o o o o</p> <p>Decoupled Cars:</p> <p>People: P0 on car C2 P1 on car C7 P2 on car C6 P3 on car C7</p>
<p>01:12</p> <p>T0 disconnect C5 T2 disconnect C6 T1 disconnect C9</p> <div> <p>Notes:</p> <p>When train 2 disconnected C6, car C7 was still attached.</p> </div>	<p>01:12</p> <p>Trains:</p> <p>T0 o o _ _ _ o C3 _ _ _ - _ _ _ /0-0-0 o o</p> <p>T1 o o _ _ _ o C0 C1 C2 C4 C8 _ _ _ - _ _ _ - _ _ _ - _ _ _ - _ _ _ - _ _ _ /0-0-0 o o o o o o o o</p> <p>T2 o o _ _ _ o _ _ _ /0-0-0</p> <p>Decoupled Cars:</p> <p> C5 - _ _ _ o o</p> <p> C6 C7 - _ _ _ - _ _ _ o o o o</p> <p> C9 - _ _ _ o o</p> <p>People: P0 on car C2 P1 on car C7 P2 on car C6 P3 on car C7</p>

01:39
P0 C2 C1

Notes:

The only change here is that person P0 moved from car C2 to car C1.

01:39

Trains:

T0 o o
 _] [_ | o | C3
 | | - | |
 /0-0-0 o o

T1 o o
 _] [_ | o | C0 C1 C2 C4 C8
 | | - | | - | | - | | - | |
 /0-0-0 o o o o o o o o

T2 o o
 _] [_ | o |
 | |
 /0-0-0

Decoupled Cars:

 | C5 |
 - | |
 o o

 | C6 | | C7 |
 - | | - | |
 o o o o

 | C9 |
 - | |
 o o

People:

P0 on car C1
P1 on car C7
P2 on car C6
P3 on car C7

05:51
P0 C1 C2
P2 C6 C7

Notes:

More people moved around...

05:51

Trains:

T0 o o
 _] [_ | o | C3
 | | - | |
 /0-0-0 o o

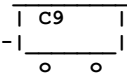
T1 o o
 _] [_ | o | C0 C1 C2 C4 C8
 | | - | | - | | - | | - | |
 /0-0-0 o o o o o o o o

T2 o o
 _] [_ | o |
 | |
 /0-0-0

Decoupled Cars:

 | C5 |
 - | |
 o o

 | C6 | | C7 |
 - | | - | |
 o o o o

	 <p>People: P0 on car C2 P1 on car C7 P2 on car C7 P3 on car C7</p>
<p>06:04 T0 connect C5 T2 connect C6 T2 connect C9</p> <div data-bbox="97 541 565 638" style="border: 1px solid black; padding: 5px;"> <p>Notes: Train cars have been reconnected. When train 2 connected C6, car C7 was connected as well.</p> </div>	<p>06:04</p> <p>Trains:</p> <p>T0 o o _] [_ o C3 C5 /0-0-0 o o o o</p> <p>T1 o o _] [_ o C0 C1 C2 C4 C8 /0-0-0 o o o o o o o o o o</p> <p>T2 o o _] [_ o C6 C7 C9 /0-0-0 o o o o o o</p> <p>Decoupled Cars:</p> <p>People: P0 on car C2 P1 on car C7 P2 on car C7 P3 on car C7</p>
<div data-bbox="97 1171 565 1554" style="border: 1px solid black; padding: 5px;"> <p>Notes: The command:</p> <pre>>java Simulator ./scenario2/trains-scenario2.txt ./scenario2/people-scenario2.txt C6 06:15</pre> <p>Asks for suspects for a murder on car "C6" at 6:15. So the simulator stops at this point and tries to find suspects. Car C6 is empty, so it broadens the search to people who could have gotten to the car without passing other people: those on C7.</p> </div>	<p>Suspects:</p> <p>P1 P2 P3</p>

Other items of note:

- While not shown here, people always wait for the trains to finish connecting/disconnecting cars before moving between cars, so people actions always take place after train actions (this is handed by the simulator your professor wrote).

TL;DR: The output of the simulator is above. You need to understand this. There are many “corner cases”.

How Can You Help?

Before you start coding, make sure you have a solid understanding of what the end program will produce and a good understanding of what code has and hasn't been written. Again, you are "working with your professor" for this project, so there is a lot more code already written. You are just writing the "support code" for the simulator, *not* the simulator itself.

TL;DR: You're not writing the simulator. That's been done for you. You're writing the "helper" code.

Code Overview

Code that's *partially* written for you:

- **Car** (**Car.java**) – This will represent a single car of a train. The only code written for you is the ASCII art representation. Here's what you need to know about cars:
 - Cars have names.
 - Cars are connected to other cars front and back. They will work like nodes in a doubly linked list!
- **Train** (**Train.java**) – This will represent a single train. The only code written is **toString()** and the ASCII art representation of a train. Here is some useful information about trains:
 - Trains have names.
 - Trains have cars attached to them. You do not need an array, cars are already nodes for of a linked list!
 - When disconnecting a car from a train, all cars *after* the disconnected car are disconnected as well.
 - When connecting a car to a train, the car (and all connected cars) are attached to the end. This is essentially concatenating two doubly linked lists together (the linked list of cars currently attached to the train + the list of cars being attached to the train).
- **Simulator** (**Simulator.java**) – This class is 95% complete, but you need to add the code for **setupInitialTrainPositions()**.
 - This is *very similar* to **setupInitialPeoplePositions()** which is written for you. Look at that if you don't remember how to use Scanner or break apart a string.
 - An outline of this method is provided (with comments).
 - Look at Page 3 for a description of how train initial positions are written (e.g. T1 C0 C1 C3)

Code *not* written for you:

- **Person** (**Person.java**) – This represents a single person. Here's the important properties of people:
 - People have names.
 - People keep track of what car they are in.
 - People can try to move from one car to another, but they can only move to a neighboring car of their current car. Remember that cars are nodes in a doubly linked list, so if you know your current car it is easy to tell what cars you can/can't move to.
- **UniqueList** (**UniqueList.java**) – You need to build a list of unique things. Here's some answers to common questions about this class:
 - This class is a generic linked list, with minimal operations, that doesn't allow duplicates.
 - You are not allowed to use arrays to implement this... so you're stuck with linked lists structures.
 - You get to design this linked list. If you need instance variables, add private instance variables. If you need an additional class, add a private nested one. It's your design, the methods defined just have to work.
- **Pair** (in **UniquePairList.java**) – Your professor is using your **UniqueList** to implement a list of pairs.
 - Each pair has a key (unique id) and value
 - **UniquePairList** has been written for you, but the **Pair** class, which stores one key with one value has not been.
 - The **Pair** class has two generics, one for the key type and one for the value type.
 - You are implementing this just to make sure you're comfortable with two generics on a class (which may be important in later projects).

TL;DR: There is a lot of information above about how to write the classes... you'll need to read this completely.

Code that's *completely* written for you:

- Rest of `UniquePairList` (`UniquePairList.java`) – See `Pair` above.
- Rest of `Simulator` (`Simulator.java`) – For curious people, here's how `getMurderSuspects()` works:
 - The following are murder suspects: People in the murder car **OR** if the murder car is empty, the people in adjoining cars (either direction) who could have gotten to the murder location, e.g. given the following five cars containing P1 through P4: [`P1`]-[`P2 P3`]-[`empty`]-[`empty`]-[`P4`]
 If the murder happened in the first car, the suspect would be P1 (the person in the car). If the murder happened in the second car it would be P2 and P3. If the murder happened in either of the empty cars, the suspects would be P2, P3, and P4 (the people who could have gotten to the empty cars without anyone noticing; P1 would not be a suspect since they would have to pass P2/P3).

Every method you are required to implement has been put into the code. Dummy return statements have been used to make sure the code compiles. Do not alter any method signatures written in this template (though you may add additional private methods, fields, and classes). Additional comments are in the code describing what each method should do, what they should return, and any other requirements (such as Big-O, code reuse, etc.).

TL;DR: The code skeleton has been written for you. It compiles. Don't break it. The code template has more tips and requirements (e.g. Big-O requirements).

Testing?

Incremental Testing

This assignment has a lot of moving parts, so we are providing you with a few (very basic) self-checks to get you started testing. If you develop your code in the order we suggested in the next section, you'll be able to run little main methods in each class to double check yourself along the way. You can edit these main methods as much as you like to do additional testing on your own (but don't edit the main method in **Simulator**).

TL;DR: Don't just write code without testing. The above describes how to write your project WHILE testing.

Below is sample output from running the test main methods for each class on working code:

```
>javac *.java
```

```
>java Car
```

```
Yay 1
```

```
Yay 2
```

```
Yay 3
```

```

| C1 | | C2 |
-|---|-|---|
  o  o  o  o

```

```
>java Person
```

```
Yay 1
```

```
Yay 2
```

```
Yay 3
```

```
>java Train
```

```
Yay 1
```

```
T1  o o
```

```
  _] [__|o|
```

```
  |_____|
```

```
  /O-O-O
```

```
T1  o o
```

```
  _] [__|o| | C1 | | C2 |
```

```
  |_____|-|_____|-|_____|
```

```
  /O-O-O  o  o  o  o
```

```
T1  o o
```

```
  _] [__|o| | C1 | | C2 | | C3 | | C4 |
```

```
  |_____|-|_____|-|_____|-|_____|-|_____|
```

```
  /O-O-O  o  o  o  o  o  o
```

```
T1  o o
```

```
  _] [__|o| | C4 | | C3 | | C2 | | C1 |
```

```
  |_____|-|_____|-|_____|-|_____|-|_____|
```

```
  /O-O-O  o  o  o  o  o  o
```

```
>java UniqueList
```

```
Yay 0
```

```
Yay 1
```

```
Yay 2
```

```
Yay 3
```

```
Yay 4
```

```
>java UniquePairList
```

```
Yay 1
```

```
Yay 2
```

```
Yay 3
```

Simulation Testing

Once you are done writing the helper code, the simulator can be used to try out different scenarios, validate user statements, and discover the murderer(s)!

Simulator can be run from the command line as follows:

```
java Simulator [TrainNotes] [PeopleNotes] [CarOfMurder] [TimeOfMurder]
```

For example, the command:

```
java Simulator ./scenario1/trains-scenario1.txt ./scenario1/people-scenario1.txt C0 23:59
```

Will run scenario 1, and give a list of suspects in car C0 at 11:59pm.

And the command:

```
java Simulator ./scenario2/trains-scenario2.txt ./scenario2/people-scenario2.txt C1 10:15
```

Will run scenario 2, and give a list of suspects in car C1 at 10:15am.

TL;DR: You need to use command line arguments to make the simulator run the correct scenario.

There are three test scenarios which describe plausible person/train movement combinations (this is described in an earlier section), but you can test out invalid scenarios by creating your own or running people/train combinations that don't work together. For example, combining the trains from scenario 2 with the person movements from scenario 1 will result in:

```
>java Simulator ./scenario2/trains-scenario2.txt ./scenario1/people-scenario1.txt C0 23:59
00:00
```

Trains:

```
T0  o o
   _] [__|o| | C3   | | C5   |
   |_____|-|_____|-|_____|
   /O-O-O   o o   o o
```

```
T1  o o
   _] [__|o| | C0   | | C1   | | C2   | | C4   | | C8   | | C9   |
   |_____|-|_____|-|_____|-|_____|-|_____|-|_____|
   /O-O-O   o o   o o   o o   o o   o o   o o
```

```
T2  o o
   _] [__|o| | C6   | | C7   |
   |_____|-|_____|-|_____|
   /O-O-O   o o   o o
```

Decoupled Cars:

People:

```
P0 on car C2
P1 on car C1
P2 on car C2
P3 on car C0
P4 on car C2
P5 on car C3
```

Hit enter to continue

If you were to delete some train actions (as is done in the “brokenscenario” folder) you can also see outputs from invalid train movements:

```
java Simulator ./brokenscenario/trains-broken-scenario.txt ./brokenscenario/people-
broken-scenario.txt C0 23:59
00:00
```

Trains:

```
T0  o o
  _] [__|o| | C2 |
  |_____|-|_____|
  /O-O-O   o  o

T1  o o
  _] [__|o| | C0 | | C1 | | C3 |
  |_____|-|_____|-|_____|-|_____|
  /O-O-O   o  o   o  o   o  o
```

Decoupled Cars:

People:

Hit enter to continue

01:17

Trains:

```
T0  o o
  _] [__|o| | C2 |
  |_____|-|_____|
  /O-O-O   o  o

T1  o o
  _] [__|o| | C0 | | C1 |
  |_____|-|_____|-|_____|
  /O-O-O   o  o   o  o
```

Decoupled Cars:

```
| C3 |
-|_____|
  o  o
```

People:

Hit enter to continue

```
[4:37] Car C1 is not in the decoupledCars list
java.lang.RuntimeException: [4:37] Car C1 is not in the decoupledCars list
    at Simulator.doNextTrainAction(Simulator.java:292)
    at Simulator.simulate(Simulator.java:200)
    at Simulator.main(Simulator.java:25)
```

TL;DR: The above is what should happen if you have invalid actions... it crashes with a useful message.