National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Mission 16
NUS Bus Simulator

Release date: 01 May 2023
Due: 01 June 2023, 23:59

Required Files

- mission16.zip

Background

"I've finally mastered Python, and I'm all ready to start school!" you thought to yourself. All those days of staying hungry have been worth it.

Staggering into NUS, you soon found yourself in quite a pickle. "I can't find the place I want to go!" After checking with a passerby, it dawned on you that you needed to take a bus to get there.

"But I have no clue which bus to take and when it will arrive! If only I had something that could tell me all that... wait a minute!"

Administrivia

This mission will give you a taste of how programs are written in Java and hopefully also an appreciation of the differences, as well as the similarities, between Java and Python. At the end of this mission, you would have finished building an event simulator for the buses in NUS! How cool is that?

We have provided you two versions of the same code templates to work with. One of them contains multiple `.java` files, one for each class. The other contains all the classes in a single file. The latter has been provided for students who wish to use online compilers or editors.

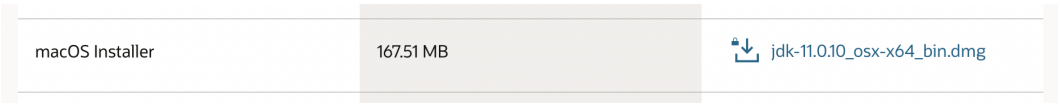
When you are done, copy only the class(es)/method(s) you were asked to modify or implement into Coursemology.

Getting Started

You will first need access to a Java compiler and source editor. We **strongly recommend** you to install and learn how to use the Java Development Kit (JDK), which will greatly improve your coding experience and reduce the amount of set-up required for your future modules.

- **MacOS:**

  First, head over to the website linked here and download the macOS installer.



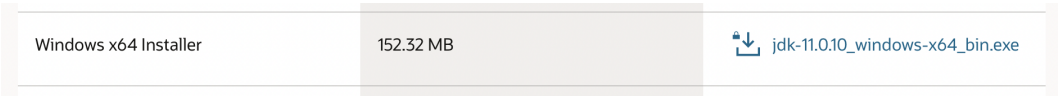
  The numbers you see may be different, but we want Java 11.

  You may need to register for an Oracle account to download the installer. There are workarounds for this, though we won't be sharing those here :)

  Then, open the `.dmg` file and run the installation, accepting all default configurations.

- **Windows:**

  First, head over to the website linked here and download the Windows installer.

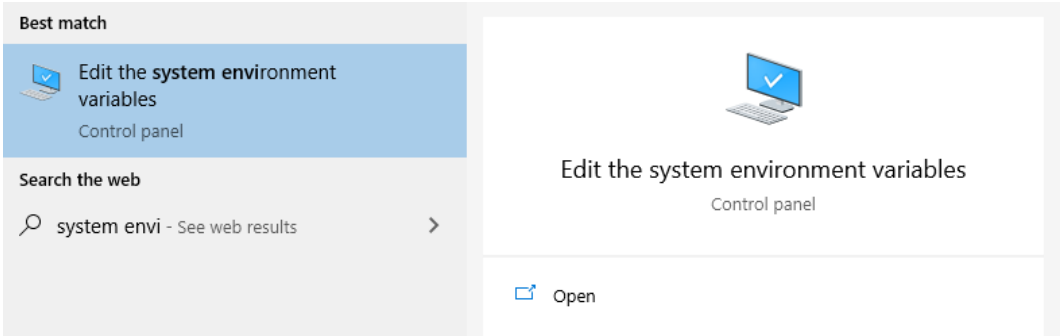


  The numbers you see may be different, but we want Java 11.

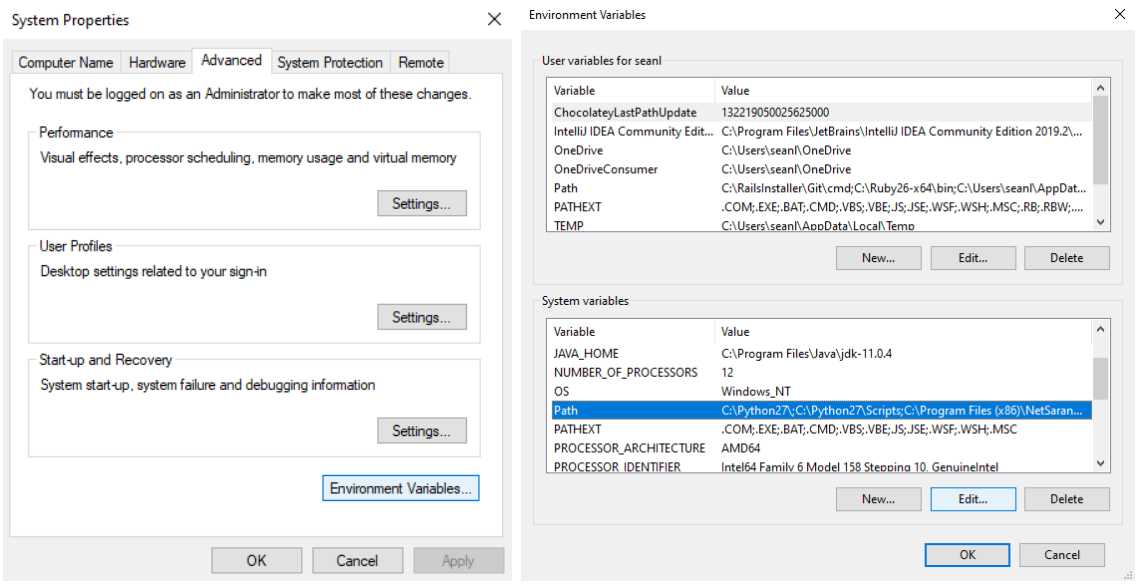
  You may need to register for an Oracle account to download the installer. There are workarounds for this, though we won't be sharing those here :)

  Open the `.exe` file and run the installation, accepting all default configurations.

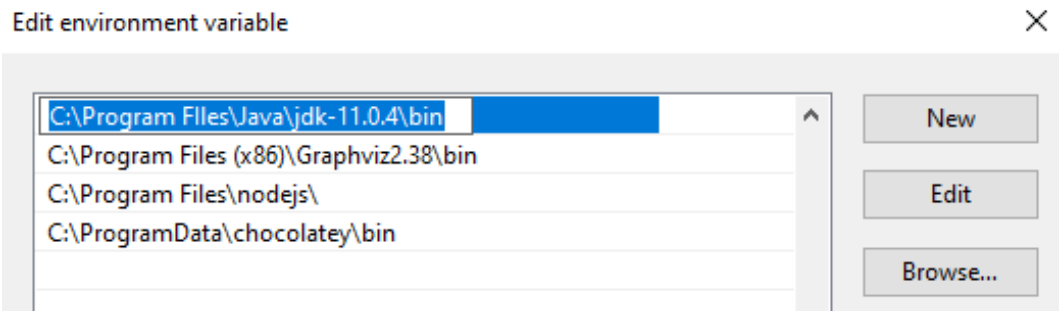
  You then need to update your environment variables. Search for the below Control Panel option and select it.



  In the Systems Properties window that appears, click on "Environment Variables", then select the "Path" system variable and click "Edit".



Finally, add a new environment variable which links to the `bin` folder of the JDK folder in your `Program Files`. Once you are done, click OK.



To make sure you have JDK installed properly, you can run `javac --version` on your terminal.

- **Others:** Do let us know if you're on some other operating system and we can help you out. Very likely, the download page linked above also contains the installation packages you may require.

Alternatives

If you're facing issues with the installations above, you may consider one of the alternatives below:

- Use an online Java editor/compiler to do this mission. You can simply copy and paste the code from the combined template. However, beware that it may be difficult to navigate around the file given its length.
- Connect to SoC's sunfire UNIX server. You can follow the instructions found here to create a SoC UNIX account and use it to connect to the school's server. On the server, Java has already been configured for you. Do ignore the CS1010-specific and C-specific instructions.

  After connecting to the server, you can either choose to use Vim, a console-based editor, to directly code on the server, or code locally on your computer and use the `scp` command to transfer files onto the server for compilation and testing.

Compilation

Java programs must first be *compiled* before you can run them. This means your code is translated into Java bytecode for the Java Virtual Machine (JVM) on your computer to evaluate, allowing any device with JVM to run Java code and produce the same behaviour.

To compile all the Java files in the same directory with the Java compiler, type the following in the terminal:

```
$ javac *.java
```

Note that you will first need to use the 'cd' (change directory) command to navigate to the folder containing the question files. The `*.java` wildcard tells your terminal to find all files that end in `.java` and pass them to the compiler. If successful, you should be able to run the program by doing:

```
$ java BusSimulator < test1.in
```

Note that even though the compiled file is called `BusSimulator.class`, we do not include the extension at the back when executing it. The `< test1.in` command does input redirection, where the contents in the file `test1.in` are passed into the executed program as user input.

You may edit the `.java` files with any text editor (or a specialised Java code editor) to do the tasks. It will be up to you to find an editor that you are comfortable with.

If none of these instructions make sense to you, please let us know via the forums to get it sorted out.

Task 1: Warming Up (3 marks)

We will first get ourselves warmed up to the syntax of Java. During this process, we will also get to see how the classes are organised. Let us take a look at the file `BusEvent.java`.

Notice that `BusEvent` contains a `bus`, a `time` (in minutes), and an `eventType`, and represents an event that occurs for a certain bus at a certain time. It implements the `Comparable<BusEvent>` interface, which means that it can be compared against other `BusEvents`, allowing them to be sorted.

```
public class BusEvent implements Comparable<BusEvent> {
    // class details
}
```

(a) Implement the `compareTo` method that is required by the `Comparable` interface. This method compares `this` against the specified object, and returns a negative number if `this` is smaller and should go first, a positive number if the specified object is smaller and should go first, or `0` if they are equal. You can read more about the `Comparable` interface here.

```
@Override
public int compareTo(BusEvent o)
```

For our `BusEvent`, we want events with an earlier time, i.e. a lower `time` value, to go first. **(1 mark)**

We will now take a look at the `BusMap.java` file. The `BusMap` class contains `static` members that help us to get the bus stops for a given bus.

Most of these members are actually arrays of `Strings` and `int`s, which contain the stop names and the time it takes to travel to that stop from the previous stop, respectively.

For example, for the second bus stop along bus B1's route, `"IT (OPP CLB)"`, it takes 5 minutes to travel there from `"KR BUS TERMINAL"`, as denoted by the second integer in `B1_TIMINGS`.

```
private static final String[] B1_STOPS = {"KR BUS TERMINAL", "IT (OPP CLB)", ...};
private static final int[] B1_TIMINGS = {0, 5, ...};
```

(b) Implement the `getNextStopAndTimeTaken` method that takes in a `busName` and a `currentStop`, and returns a `Pair` containing the next stop and the time it takes to get there.

```
public static Pair getNextStopAndTimeTaken(String busName, String currentStop)
```

The `Pair` class is defined in the same `BusMap` class, and is an inner class used to associate a `stopName` and the `timeTakenFromPreviousStop`. We will also expect `busName` to take one of these four values: `"A1"`, `"B1"`, `"C"`, `"D1"`.

If `currentStop` passed in is `null`, i.e. the bus is not at any stop yet, we will return the very first stop along the route. If the `currentStop` is the final stop along the route, we will instead return `null`, i.e. no pair. You may assume that all bus stops along the same route have unique names. **(2 marks)**

Tip #1: Instead of writing lengthy `if-else` statements, you can look into using a `switch` statement instead. Check out the `readBusEvents` method in `BusSimulator.java` to see an example of it in action.

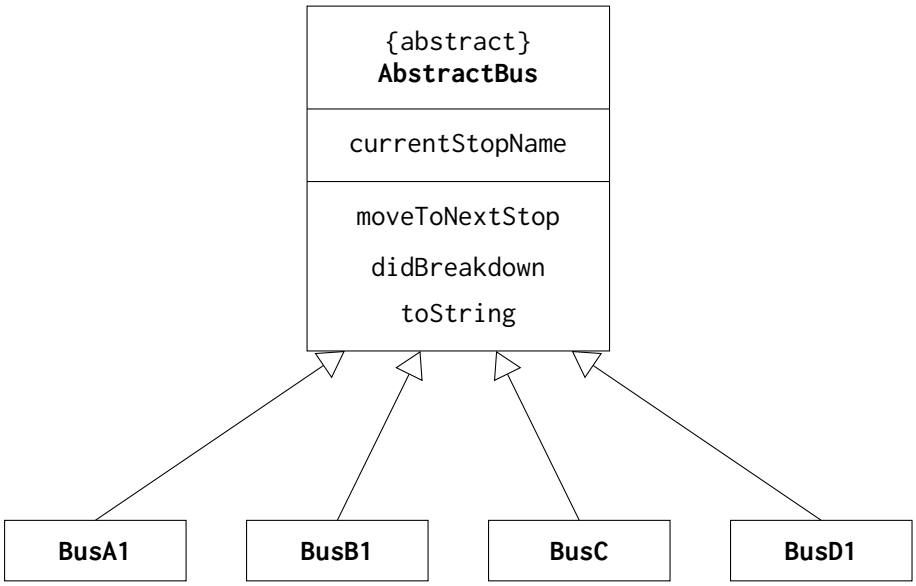
Tip #2: Just like in Python, we also have exception handling in Java. What if the `busName` or `currentStop` passed in are invalid?

Task 2: Basic Java OOP (3 marks)

With our auxiliary classes sorted out, let us get to the meat of modelling the buses. We will now take a look at the abstract class `AbstractBus`.

An abstract class cannot be instantiated, and can support attributes, abstract methods and methods with a default implementation. Our `AbstractBus` class contains one attribute and three abstract methods. An abstract method must be implemented by a concrete class that inherits from the abstract class.

Our bus event simulator will support four of the many buses that NUS has: A1, B1, C and D1. Each of these buses will be represented as a class that inherits from `AbstractBus`.



A class diagram of the buses

For this task, we will be implementing one of the three abstract methods: `moveToNextStop`. Notice the `@Override` annotation above the method. Though not compulsory to include it, it brings about benefits, both in terms of readability and in helping your compiler detect errors when compiling your code. You have actually seen this before with `BusEvent` and the `Comparable` interface!

```
@Override
public int moveToNextStop()
```

(a) Implement `moveToNextStop` for each of the four bus classes. The superclass attribute `currentStopName` will start as a valid stop name, as initialised in the constructor of each of the four classes.

This method should update the bus' `currentStopName` and return the time taken to travel to this new stop, if the bus is not already at the final stop. Otherwise, if the bus is already at its last stop, return `-1`. **(2 marks)**

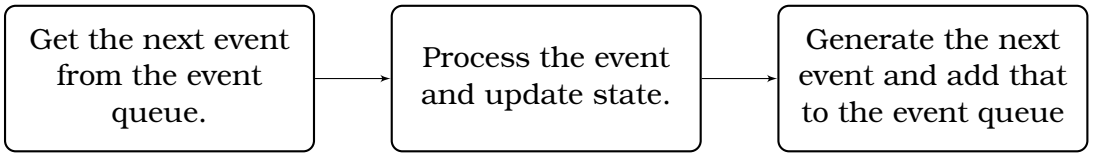
(b) Very likely, you would be repeating yourself in each of the classes. This is the intended outcome, as we want you to practice overriding in Java. However, are there other ways we can implement `moveToNextStop` so as to reduce code duplication? Discuss this and propose an alternative implementation. **(1 mark)**

Task 3: Simulating Buses (4 marks)

You have finally warmed up and configured your buses. You were just about to confidently head over to the bus stop when you suddenly realised...

"Ack! I completely forgot about the simulation logic itself!"

For our event simulation, we want to follow this flow:



For example, we can have the following event: bus A1 is at `"PGP (START)"` at $T = 5$ minutes. Since it takes 3 minutes for it to reach its next stop, `"KR MRT"`, we can generate the next event where this same bus A1 is at `"KR MRT"` at $T = 8$ minutes. Repeat this for all bus events in the event queue until all buses have reached their final stop and no further events are generated, and we are done!

We will simulate the event queue using a `PriorityQueue` from the Java library. A priority queue is essentially a sorted queue, where elements of higher priority will be shifted to the front, while between elements of the same priority, they will be ordered in a First-In-First-Out (FIFO) order. We have already implemented the priority comparison logic via `Comparable` earlier!

(a) Before we begin, let us make sure that our current changes so far have no issues. Run the following commands:

(a) Compile the code:

```
$ javac *.java
```

(b) Execute the compiled code and redirect the test file as input:

```
$ java BusSimulator < test1.in
```

(c) You should see the following output in your console:

```
Bus simulator is ready!
There are a total of 2 events.
```

(b) Implement the `run` method for `BusSimulator`. The event queue has been populated for you. You just need to follow the flow illustrated above while the queue is not empty.

For each bus event, you should print its information to the console, move the bus to its next bus stop, and generate the next event, if any, to add back into the queue. Here are some methods for the Java Queue interface that may help you with this part:

```
queue.poll();                // Removes and return the head of the queue
queue.offer(BusEvent event); // Inserts the specified event into this queue
queue.isEmpty();             // Returns true if this queue contains no elements.
```

Here's the expected output for `test1.in`:

```
Bus simulator is ready!
There are a total of 2 events.
3: Bus A1 arrives at PGP (START).
5: Bus B1 arrives at KR BUS TERMINAL.
6: Bus A1 arrives at KR MRT.
9: Bus A1 arrives at LT 27.
10: Bus B1 arrives at IT (OPP CLB).
11: Bus A1 arrives at UNIVERSITY HALL.
11: Bus B1 arrives at OPP YIH.
13: Bus A1 arrives at OPP UHC.
13: Bus B1 arrives at UNIVERSITY TOWN.
14: Bus A1 arrives at YIH.
15: Bus B1 arrives at YIH.
16: Bus A1 arrives at CENTRAL LIBRARY.
16: Bus B1 arrives at CENTRAL LIBRARY.
18: Bus A1 arrives at LT 13.
18: Bus B1 arrives at LT 13.
20: Bus B1 arrives at AS 5.
21: Bus B1 arrives at BIZ 2.
23: Bus A1 arrives at AS 5.
29: Bus A1 arrives at COM 2.
32: Bus A1 arrives at BIZ 2.
33: Bus A1 arrives at OPP TCOMS.
34: Bus A1 arrives at PGP (END).
```

To avoid manual checking, you can also use the `test1.out` provided. To do so, simply run this following command:

```
$ java BusSimulator < test1.in | diff test1.out -
```

This command pipes the printed output into the `diff` command, which compares it against `test1.out`. If you don't see any output after running the above command, this means the output is identical to that in the file! **(4 marks)**

Tip: To make the printing easier, you can consider implementing the overridden `toString` method in each of the four bus classes, as well as the `toString` method in `BusEvent`. They function much like Python's `__str__` method.

Task 4: Pseudorandom Fun! (6 marks)

As you watch your simulator churn out scenarios with great ease, you start to feel amazed at your own ingenuity. Perhaps you can even sell this idea to NUS itself!

Just as you were thinking that, you suddenly heard a loud sound coming from the nearby bus stop. It turns out a bus just broke down!

Glancing over, it soon dawned on you that you've forgotten something yet again. Sometimes, mishaps occur! The bus had just broken down at the nearby bus stop *1 minute* after it arrived. Staring blankly at it, you saw that the repair team arrived and got the bus up and running *10 minutes* after it broke down.

To simulate such unfortunate events, we shall make use of a **pseudorandom number generator** (PRNG). PRNGs produce numbers that look seemingly random, but are actually completely determined by an initial value. By providing a fixed initial value, we can guarantee that we get the same "random" numbers every time.

We have already initialised an instance of `Random` in the `BusSimulator` class.

```
private static final Random random = new Random(999);
```

We will be *passing around* this `random` instance to wherever it's needed. If you initialise new instances, you will not get the same expected output.

(a) Implement the `didBreakdown` method for the four bus classes. It should take in the `random` instance that was previously initialised in `BusSimulator`, and return a boolean value.

Let's say an event has a probability of 0.3 of occurring. We can compute whether it occurs by doing:

```
boolean didOccur = random.nextDouble() < 0.3;
```

Apply this definition to the `didBreakdown` method, where the probability for each bus to break down is:

- A1: 0.1
- B1: 0.2
- C: 0.05
- D1: 0.4

(2 marks)

(b) Now, let us implement a new `runWithBreakdowns` method in `BusSimulator` to handle these scenarios. You observed the following scenarios earlier when staring at the broken bus:

- **Operational:** While operational, a bus has a chance of breaking down at every stop, with the probability determined by the `didBreakdown` method from part (a). If a break down occurs, then a Broken Down event should be generated and added to the queue. This new event should occur 1 minute after the bus' arrival.
- **Broken Down:** When broken down, the bus will need to wait for repairs. This wait will take 10 minutes, and a Repaired event occurring 10 minutes later should be generated and added to the queue.

- **Repaired:** Upon being repaired, the bus can then continue on with its journey to the next stop. An Operational event should then be added to the queue *for the next stop*, i.e. the bus resumes travelling the moment it's repaired.

Here's a sample output demonstrating these three types of events:

```
3: Bus A1 arrives at PGP (START).
6: Bus A1 arrives at KR MRT.
7: Bus A1 has broken down at KR MRT.
17: Bus A1 has been repaired at KR MRT.
20: Bus A1 arrives at LT 27.
```

To help with your implementation, we have provided an `EventType` enum in the `BusEvent` class, along with the `eventType` attribute. You can perform a `switch` statement using enums as well.

To run this method, make sure you change update the `main` method of the `BusSimulator` from:

```
simulator.printStatus();
simulator.run();
```

to

```
simulator.printStatus();
simulator.runWithBreakdowns();
```

Once again, you can test your solution by compiling, this time executing it with the following command:

```
$ java BusSimulator < test1.in | diff test1_random.out -
```

(4 marks)

That is all for this simulator. All the best! Do remember to submit carefully on Coursemology.