# CSE374 – Spring 2021 – Assignment 1

This assignment is done individually. You can search for information online, use your textbook, ask the instructor or Teaching Assistant for help. Working with anyone else is a non-authorized collaboration, that is, an act of academic dishonesty. If you have any doubt or question about what qualifies as academic dishonesty, please contact the instructor to make an informed decision.

*Objective*: Implementing and testing ADTs in the Eclipse environment.
*Due date*: Sunday Feb 7th at 11:59pm. Assignments don't receive a grade after midnight.
*How to submit*: A ZIP file sent through the course website (no emails accepted).
ZIP your whole Eclipse project, name it "*Firstname_Lastname*_Assignment1.zip"
Submissions in other file formats, *or using other environments*, will not be graded.
You are responsible for ensuring that your submission was submitted. A quick way to check is to download it back from the website and open it.
*Grading (14 pts)*:Each box shows the number of points in a question. Commenting counts for 1 point.

3 points

## Part 1

Go to the folder named Java Refresher. Start the "index". This learning module will help you refresh your skills in Java, understand how to use Maven in Eclipse, and prompt you to write modern code. There is a main menu in this learning module, so you can learn about one aspect and cover the other one on another day. Once you have finished learning, take the quiz contained in this module. At the end of the quiz, there is a screenshot. **Include the screenshot in your ZIP submission**. *Please carefully double check that your submission does include the screenshot!*

*This is a very forgiving quiz. You have an infinite number of attempts to get each question right. It will only let you move onto the next question once you got it right. The goal is to encourage you to get all these foundational notions clarified. Future quizzes may be less forgiving.*

## Part 2

### I. Overview

In our "review of Java and Abstract Data Types (ADTs)", you were introduced to the cab problem:

*Students are waiting in line to get to Cincinnati. There are cabs. Find cabs for the students.*

We created a cab ADT and wrote one algorithm that matches students with cabs. In this assignment, we will look at three classical algorithmic problems: (i) different implementations of an ADT; (ii) creating synthetic data (i.e. using simulators) to test implementations; and (iii) re-designing the algorithm as more real constraints are taken into account.

**To start, re-create the code as you saw it in the lecture**. Then you can proceed with the questions below, in the order in which they are given. Section V is the core algorithm section (worth many points!) and sections II to IV are more of a refresher / getting used to Eclipse.

### II. Implementing our two ADTs through three classes

$\frac{1}{4}$ point

So far, we cannot really run our algorithm because we only have the ADTs but no implementation. In your structures package, implement the People ADT via a *Person* class whose constructor takes a name (String) and saves it as the class' only variable. In this class, redefine the default toString[1] method so that printing a People object will show its name. Here is how to test your code:

```
People p1 = new Person("James");
People p2 = new Person("Chen");
System.out.println(p1);    //this should show James
```

[1] See the textbook on page 85 if you do not remember the structure of the toString method.

Then, you will implement the Cab ADT in three different ways, so you can appreciate how the requirements of an ADT can be satisfied by different implementations. The classes to code are:

1 pt

• *AggregateCab* class. This only tracks of the number of passenger seats (passed to the constructor) and number of passengers (initially 0). Said otherwise, this class needs only two number variables.

• *IndividualsCab* class. This tracks each individual passenger using an array. The array is initialized by the constructor given the number of passenger seats.

1 pt

• *Greyhound* class. Although it's simpler to know that all classes ending in -Cab are an implementation of this ADT, they don't have to be. A Greyhound is organized in rows, where each row consists of two seats on the left and two seats on the right, separated by a central alley. Here is a Greyhound with 2 rows:

| | | | | |
|---|---|---|---|---|
| Passenger 1 | Passenger 2 | *Alley* | Passenger 3 | Passenger 4 |
| Passenger 5 | Passenger 6 | *Alley* | Passenger 7 | Passenger 8 |

Like all classes implementing a Cab, a Greyhound is given the number of required seats. Since it only works by rows, it will create enough rows to support at least the desired number of seats. For instance, if you need 6 seats then the Greyhound will make two rows.

The following is an *example* to test your code:

```
Cab aggregate = new AggregateCab(3);      //creates a cab with three available seats
aggregate.addPassenger(new Person("Paula"));
aggregate.addPassenger(new Person("Vijay"));
aggregate.addPassenger(new Person("Brad"));
if(aggregate.isFull()){ System.out.println("Your implementation seems to work!"); }
```

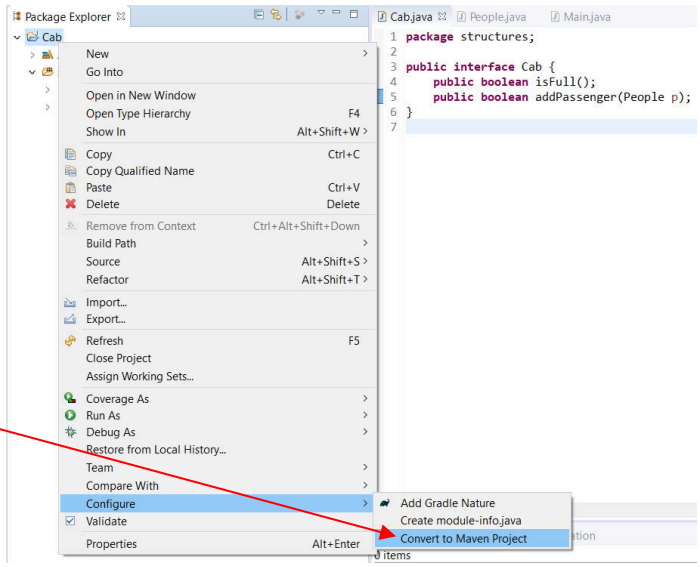
**III. Controlled environments to test algorithms: connecting to a simulator**

There are two broad ways to check whether an algorithm works. First, we *could* automatically check whether it meets a mathematical specification, but (*i*) that requires advanced software engineering skills[2] and (*ii*) the algorithm might be correctly programmed but its design turns out to be wrong. Second, we can simulate input data for the algorithm and see what it does. We'll do simulations.

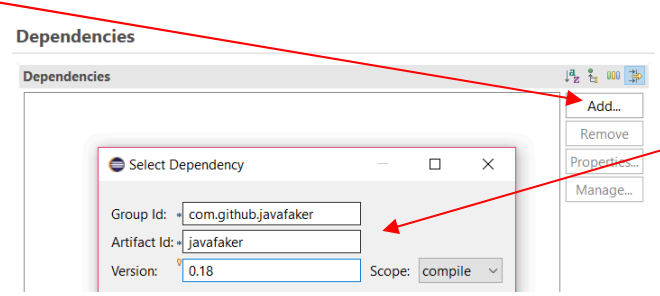
*Continue on the next page to see how to use Apache Maven. Remember that Part 1 of this assignment would also have shown you how to operate Maven!*

[2] See https://en.wikipedia.org/wiki/Model_checking

Simulating input data can be a complicated task by itself. Fortunately, many simulators already exist and they just need to be integrated into your project. The old-and-dirty way of integrating with a Java project would be to manually download a library/Java code and add it to your project. But keeping track of your downloads and their version becomes problematic. The more modern way is to specify that your project *depends* on something else. **Apache Maven** is a common tool to specify such **dependencies**.




Right click onto your project and convert it to Maven. Eclipse may hang for 10 seconds, so *just wait* (don't click more). A pop-up appears. Click 'Finish'. Voila! It will open Cab/pom.xml, which is a configuration file. Click on 'Dependencies' and Add.


Complete the three fields exactly as shown in this picture. This is a great library to generate data (e.g. names, addresses, even restaurants and reviews…)

1 point

Run your project once so Maven dependencies (i.e. Faker) are loaded.

Here is a code snippet on how you can now use this library in your code:

```
Faker faker = new Faker(); //do this just once
//for each Person you make up, generate a full name:
People p1 = new Person(faker.name().fullName());
People p1 = new Person(faker.name().fullName());
System.out.println(p1+"\n"+p2);//should see two plausible names
```

Given that you have implemented the ADTs and gotten a simulator to create input data, you now have all you need to test the algorithm. Professionally, we should use JUnit to write tests but installing Maven was enough software engineering for this first assignment. So we'll do tests the old way, by writing two static methods in the Main class.

In the Main class, create a static method `test1` that:

- creates 50 people with random names (using Faker) and enough cabs with 3 or 5 empty seats to accommodate all 50 people
- calls `storePeople` to ensure that the algorithm works.

Similarly, create a static method `test2` that provides too few cabs for the 50 people, and shows the name of all the people who couldn't get a ride.

$\frac{3}{4}$ point

Use your test methods to demonstrate whether your implementations of the ADT work.

**IV. Choosing the right ADT**

So far, our implementation 'works' but this is not setting the bar very high. In this course, we are concerned about correctness *and efficiency*. Consequently, we need to find the best ADT given our needs, instead of just grabbing the first ADT that seems to do the job.

Read chapter 1.3 of the textbook, then do the following:

½ point each

• Create a class ConstrainedCapacityBag that implements the bag ADT (see page 154-155) but has a capacity limit, given by the constructor.
• Create a class BaggyCab that implements the Cab ADT and relies on ConstrainedCapacityBag to store the passengers.
• Assigning people to cabs by processing a *list* is inefficient. People form a queue waiting for cabs, and cabs are organized in a queue. And so it happens there is a Queue ADT for this situation! In the Main, create a static method fastStorePeople which takes a queue of people, a queue of cabs, and uses operations from the Queue ADT. Do *not* implement the Queue yourself: use one of Java's existing implementations.

*(You should make it a habit of testing your code even if we're not asking for it. You're at home so you can test and ensure that what you submit is right! Consider testing the code above…)*

**V. Real people are complicated**

Squeezing people into cabs would be simple if they behaved like bags of potatoes. It allows us to think along the lines of "space for one person = put one person there". In practice, people (unlike potatoes) tend to have ***preferences***. For instance, they come in groups and want to stay together. Consider that Amy, Chen and Mark form a group. Even if two cabs have respectively 1 and 2 free seats, the people would rather refuse and wait for a cab with three seats.

2 points

Modify your code such that each person can also be given an optional group number. Then, in the Main, write a static method storeGroupPeople that takes in queues of people and cabs (per section IV) and ensures that people who are in the same group do board together.

For instance, consider the following queue of people:
[ (Cersei, 1), (Jaime, 1), (Joffrey, 1), (Sansa, 2), (Ramsay, 3), (Missandei, 4), (Daenerys, 4)]
And the queue of cabs:
[ (BobTaxi, 2), (PhilTaxi, 4), (SnowMobile, 1), (PartyBarge, 10)]
Then, one possible assignment would be: Sansa in BobTaxi, Cersei/Jaime/Joffrey in PhilTaxi, Ramsay in SnowMobile, Missandei/Daenerys in PartyBarge. If there is an assignment that satisfies all customers, your algorithm should do its best to find it. Partial points are given for sub-optimal code.

½ pt

Extend the People ADT so that it has two additional methods:
*hasCOVID()* which returns a Boolean is the person has COVID-19; and
*getsCOVID()* which contaminates a person with COVID-19
Ensure that your Person class remains compliant to the People ADT by implementing these methods. When created, a person should have a 10% chance of having COVID-19 (use randomness).

2 pts

In our strange post-apocalyptic-yet-usual society, assume that cabs will avoid mixing people who are healthy with those who are sick. Write a static method apocalypticCabs that takes in queues of people and cabs (per section IV) and ensures that a cab does not mix sick and healthy people. The group number does not need to be used in this question, so just ignore the group.

1 pt

We do not always know who has COVID-19, because about half of the cases are asymptomatic. Let's account for how a disease spreads in this case, by focusing on the Greyhound (because it has the most interesting geometry). In a Greyhound, you know precisely where people are; they're highly likely to spread to those sitting immediately next to them, and less likely for the ones in front of them. Modify the Greyhound so that the disease spreads around each person who is sick upon entering the cab. The algorithm does not deal with "disease spread chain" (e.g. Hanbo infects Yujia in the cab who then infects…). Test your Greyhound class to ensure that it does spread the disease.

Assignment questions? Doubts? Feel free to contact our teaching assistant, Elinore, at:

eavensek@miamioh.edu