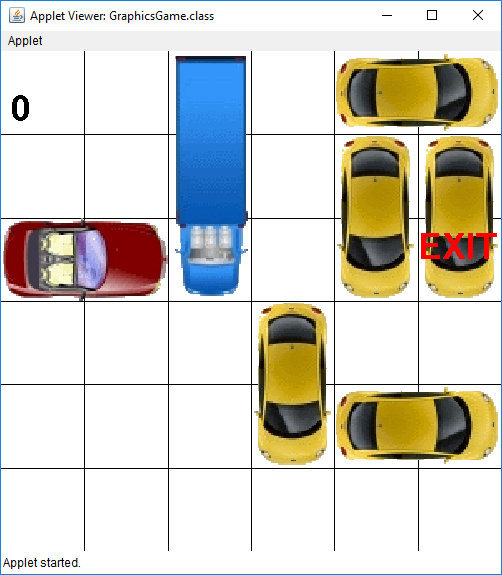
# COMP 55 – Traffic Jam Space & Vehicle

You’re tasked with creating a Traffic Jam like game, which is a popular puzzle style game in which you have to move cars around in order to be able to get your car to the exit.



## The Text-Based Version

While we’ll eventually get to this graphical version, you’ll first be tasked with creating a version that is much closer to the types of assignments that you had in COMP 53. The text-based version is large enough that we’re going to break it into three parts, where this first part is just meant to get you back in making classes, so we’ll just focus on making two classes, **Space & Vehicle**.

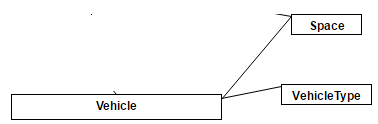
What we are looking for is for you to start programming on a regular basis at a steady pace, so you only have a few days to complete this assignment, check Canvas to see when it’s due.

You’ll pass this assignment if you can get through without the program crashing or causing any types of exceptions, which we’ll review. What you’ll be writing here are the building blocks that we’ll use for interactions in both the graphical and text versions of this game. What we will eventually ask the user to do is to type in a location for a car they wish to move and ask for the number of spaces (positive or negative) they want that particular vehicle to move. If they picked a correct location, the vehicle would then move to that location. The rules in traffic jam are that a car must stay in the bounds of the board, can only move in the direction it is currently facing, and must not collide with any of the other cars as it moves to its new position

## The programming requirements

In the real-world, you don’t always start from scratch and you don’t just jump in. Instead, you come up with some specification and design or you come in to start working on a project that’s already been designed, so we are going to work in this fashion here. So I’ve decided to start this assignment and to give some requirements to you. Your first job is to take the existing java files and to complete two of the files given, Space.java and Vehicle.java. This will require more time & effort on your part, since many of you have always started an assignment from scratch. To help you along in the process, I’m going to explain the files in detail and give you a small plan of attack.

While there are seven java files in the starter folder, for this first part of the assignment we are only going to worry about three. Each file represents a different class that is responsible for a different part of the game. For example, the Vehicle class is meant to represent a single vehicle. The Space class is really meant to hold a single row and a column, like a space (or location/coordinate) on the board. Some of the classes are big, and others are small. Here’s a small diagram that shows how the three different java files are related to each other.



Notice that we also have here VehicleType, which is meant a small utility class that you’ll use. VehicleType is something that is called an enum. For help with enums, look below.

These three boxes represent the three files that you will be opening up and reading for TrafficJam.zip, which is the starter code that you can download from Canvas. Once you finish implementing the two files, you will be able to run Vehicle with the supplied code and have it not raise any exception or crash on you. In order to get you to this stage, I would suggest you follow this plan of attack.

## The Plan of Attack

### Step 1) Understand the diagram above and VehicleType class (Suggested Deadline: today)

*VehicleType*.java – This file is already implemented for you (Yay!). VehicleType is an enum, or enumerated type. Rather than try to store a character or something else to let us know the type of vehicle we have, this file defines the three vehicle types (truck, auto, & car) for us. If you are not sure or are completely thrown off by the use of enums, you should read this short article about enums: <http://crunchify.com/why-and-for-what-should-i-use-enum-java-enum-examples/>

### Step 2) Implement the Space class (Suggested Deadline: today/tomorrow)

*Space*.java – This class is meant to be a very simple. Rather than constantly having a bunch of variables each for rows and columns, we want to bundle them together, almost like a struct. Having the Space class allows us to bundle the row and column together. Because vehicles will be occupying particular rows and columns, having a class that is responsible for attaching a row to a column to represent a space/location is a cleaner approach. This is a class that is used throughout, and has very little error checking. **Make sure to complete the Intro to Java lab with the Fraction class before attempting to write this class.** You can test this Space class by writing a main that creates a Space object and then prints out its row and column individually, and then choosing to run it in eclipse. In fact, here’s a main function you can place inside your Space class. Run this and if you get right print statement here, you can move on to Step 3:

**//Small test code to put in Space.java to check to see if your class works**

**public static void main(String[] args) {**

**Space one = new Space(3, 4);**

**Space two = new Space(1, 6);**

**two.setRow(two.getRow()+1);**

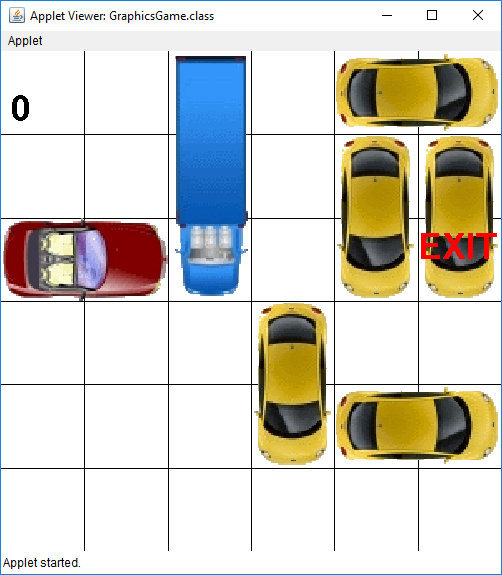
**two.setCol(two.getCol()-1);**

**System.out.println("one r: " + one.getRow() + ", c: " + one.getCol());**

**System.out.println("two r: " + two.getRow() + ", c: " + two.getCol());**

**}**

Just to be super clear on what a **Space** will eventually be, let’s leverage a part of the drawing up top, and given a new grid of 3 rows and 3 columns, you can see what three different spaces would represent. In this case, the entire yellow square would consist of a space that is row 0, column 2. I drew some other examples to help you all out, but we don’t have a grid just yet, we’ll work on that in the future.



### Step 3) Implement the Vehicle Constructor and getters (Deadline: early in day before assignment is due)

*Vehicle*.java – This class is meant to represent a single vehicle, which will know its own current starting position, its type, if it is vertical, and its length. For example vehicles like trucks are longer and have a length of 3. Implementing Vehicle will be more work. As part of the Vehicle, you’ll have to write the constructor, which should require the type, a starting row and column, a length, and then whether it’s vertical in that order. Part of Vehicle.java will be getters and setters and trying to figure out what else you should store in a Vehicle. If you are having trouble with getVehicleType, you should look at the resources posted on canvas. The only other functions that will require more thought and testing are the move and spacesOccupied functions.

### Step 4) Implement move and spacesOccupied in Vehicle (Deadline: mid to late day before due)

*move* – The purpose of having a move function in the Vehicle class is simply **to update the vehicle’s own start row and column**. You may think that many more things would need to happen in there, like actually moving it on a board or checking to see if it’s in bounds, but since we are just dealing with the Vehicle class at the moment, the Vehicle class knows nothing about the type or size that the board will be on, which makes it more modular. This means that we could potentially use this class on some other type or sized board for example. It also helps us break down the problem into smaller parts, which allows us to more easily test and implement our solution. The move function will take in a parameter ***numSpaces,*** which is the number of spaces that the vehicle should move. Instead of having some sort of direction or other way to indicate the direction that we want the vehicle to move, we simply use a positive and negative integer to help the vehicle determine what its new start position should be and update it. Since the vehicle knows whether or not its vertical, move is the first function where we are using some of the knowledge that vehicle has to update its own values. Similar to move, ifIWereToMove is a function that works like move, however, it does not update the Vehicle’s own start row and column, but instead, just returns a new Space with the start and row/col of where the vehicle would end up “if it were to move” *Both of these do not have the method declarations in the starter files, so you’ll have to add them in as well.*

*spacesOccupied* - While you may be tempted to store all of the spaces a vehicle occupies, you only need to store the row and column for where the back wheels of the car are located (the start row and column). Instead of storing the spaces, we are going to write two additional methods. The first method, spacesOccupied, will return an array of Spaces that a vehicle occupies. This can be determined by using the front wheels, the number of spaces and its orientation. The other helper method is spacesOccupiedOnTrail, which works like spacesOccupied, but takes in an additional number. That number represents the amount that a vehicle would move. In that situation, spacesOccupiedOnTrail would return an array of all of those spaces that a vehicle would travel over. In addition to that, the other two methods that will require time will be the move methods, which given the number of spaces, will change the car’s current starting location based on whether or not it is vertical. Another helpful method to have is a toString() method, which will transform the object into something that is readable in string format. This will allow you to have a textual representation of a vehicle that is meaningful in some way. Once you have this implemented, like what was given to you in step 2, make a main function that creates a vehicle, prints out the vehicle’s contents, moves the vehicle around and checks to see if the location is valid/correct. **To check your understanding of this, I’ve created a quiz on canvas, that you can retake until you have a full understanding of how spacesOccupied works. Please complete the quiz before trying to write any of the spacesOccupied functions to make sure they work.**

In order to make sure that you get a handle on everything that you need to write, I’ve given you a basic UML model below, which shows in more detail the methods you should write (including what to pass in and return), the instance variables you should store, and a little bit on how everything is related. It does not include how you should write your constructors. Once you get that done, you should write some test code similar to what you had above. I have some code to help you out that I’ve provided here to test out the spacesOccupied functions. You also should write some test code for move, which I’ve placed a comment about below

**//this snippet would go inside of a public static void main in Vehicle.java**

**//Assume Vehicle constructor is type, startRow, startCol, length, isVertical**

**Vehicle someTruck = new Vehicle(VehicleType.*TRUCK*, 1, 1, 3, true);**

**Vehicle someAuto = new Vehicle(VehicleType.*AUTO*, 2, 2, 2, false);**

**System.*out*.println("This next test is for spacesOccupied: ");**

**System.*out*.println("vert truck at r1c1 should give you r1c1; r2c1; r3c1 as the spaces occupied:does it?");**

***printSpaces*(someTruck.spacesOccupied());**

**System.*out*.println("horiz auto at r2c2 should give you r2c2; r2c3 as the spaces occupied:does it?");**

***printSpaces*(someAuto.spacesOccupied());**

**System.*out*.println("if we were to move horiz auto -2 it should give you at least r2c0; r2c1; it may also add r2c2; r2c3 to its answer:does it?");**

***printSpaces*(someAuto.spacesOccupiedOnTrail(-2));**

**//ADD SOME MOVE AND IFIWERETOMOVE TEST CODE BELOW THIS LINE**

The above snippet uses a helper method called **printSpaces** that you can place in Vehicle.java before or after the main method (but not inside the main method)

**// prints out more legibly the row & columns for an array of spaces**

**public static void printSpaces(Space[] arr) {**

**for(int i = 0; i < arr.length; i++) {**

**System.*out*.print("r" + arr[i].getRow() + "c" + arr[i].getCol() + "; ");**

**}**

**System.*out*.println();**

**}**

Please Please PLEASE, start early, and if something does not make sense as you read the code or this handout, please don’t hesitate in asking me. This shouldn’t be a high pressure situation, it’s just meant to get you back in the swing of things from a programming perspective.

C:\Users\Osvaldo\Downloads\Traffic Jam (5).png

If you do this piece by piece like I outlined above, it will make more sense. I will be providing additional resources that will be helpful that will be posted and or Teams. But you can also look at the set of resources here:

<https://www.dropbox.com/s/yga237cg8pqujxg/Initial%20Set%20of%20help%20resources.xlsx?dl=0>

You can ignore the other files for this first part. They are meant for the subsequent versions. For this first part, you just have to worry about implementing **Space** and **Vehicle**.

To submit this project submit all the contents as a zip file. The best way to do this for grading purposes is to go into eclipse and say *File->Export*…And the export the entire project as a zip file.