

## Location & Vehicle

In this individual project, you will create a Traffic Jam-like game, which is a popular puzzle-style game in which you have to move cars around in order to get your own car to the exit. The rules in Traffic Jam are as follows. 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.

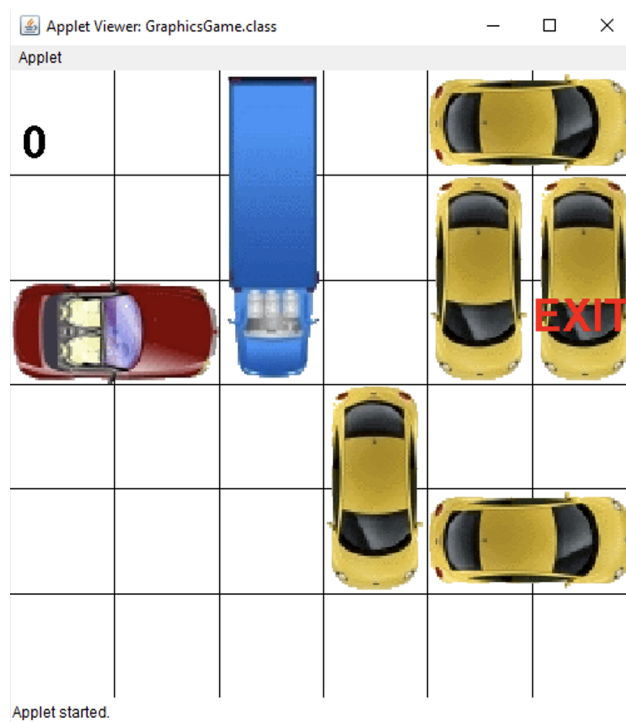


Figure 1: A screenshot of the Traffic Jame game.

## The Text-Based Version

While we will eventually get to a graphical version as shown in figure 1, you are first tasked with creating a text-based version that is much closer to the types of assignments that you have had in previous courses. This version is large enough that we are going to break it into three parts. In the final version, we will ask the user to type in a location for a car they wish to move and ask for the number of spaces (positive or negative) that they want to move it. If they picked a correct location, the vehicle would then move accordingly.

This first part of the project is meant to reintroduce you to creating classes, so we will focus on creating two classes, **Location** and **Vehicle**. As part of this assignment, you will be writing the building blocks for interactions in both the graphical and text versions of the game.

## The Programming Requirements

In the real-world, you do not always start from scratch or jump right in. Instead, you might come up with specifications and a design, or start working on a project that has already been developed. We are going to work in a similar fashion here.

Your first job is to take the existing Java files from the starter code and to complete two of the files given, `Location.java` and `Vehicle.java`. This will require some time and effort, as many of you will have started assignments in the past from scratch. To help you along in the process, the following sections explain the files in detail and provide you with a small plan of attack.

While there are seven Java files in the starter code, for this first part of the project 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 **Location** class is meant to hold a single row and a column, like a coordinate on the board. Some of the classes are big, and others are small. Figure 2 is a small diagram that shows how the three different files are related to each other.

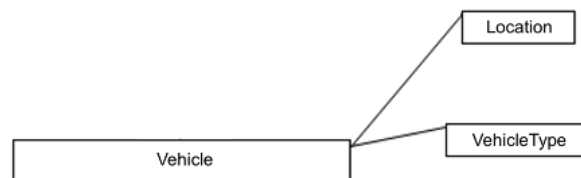


Figure 2: Relationship between classes in this assignment.

Notice that we also have **VehicleType**, which is a small utility class that we will use. **VehicleType** is something called an enum. (For more details on enums, look below.)

These three boxes represent the files that you will be working on. Once you finish implementing them, you should be able to run **Vehicle** with the supplied code without any errors. In order to get to this stage, I would suggest the following plan of attack.

## The Plan of Attack

### Step 0) Download the provided starter code

Download the starter code and import it into Eclipse via File → Import... → Existing Projects into Workspace. For this assignment, you only have to focus on implementing `Location` and `Vehicle`. The other files are meant for subsequent versions.

### Step 1) Understand the diagram in figure 2 and the `VehicleType` class

`VehicleType.java` This file is already implemented for you. `VehicleType` is an enum, or enumerated type. Rather than try to store a character or something else to let us know which type of vehicle we have, this file defines the three vehicle types (truck, auto, and car) for us. If you are not sure about the use of enums, this short article provides additional information:

<http://crunchify.com/why-and-for-what-should-i-use-enum-java-enum-examples/>

### Step 2) Implement the `Location` class

`Location.java` This class is meant to be very simple. Rather than using individual variables for each row and column, we will bundle them together in a `Location` class, as having a class that represents an individual location is a cleaner approach. For instance, given a new grid of three rows and three columns in figure 3, the entire yellow square would consist of a location that is row 0, column 2. (We will work on the remaining aspects of this grid in future assignments.) The `Location` class is used throughout the project and has very little error checking. It should include getters and setters.

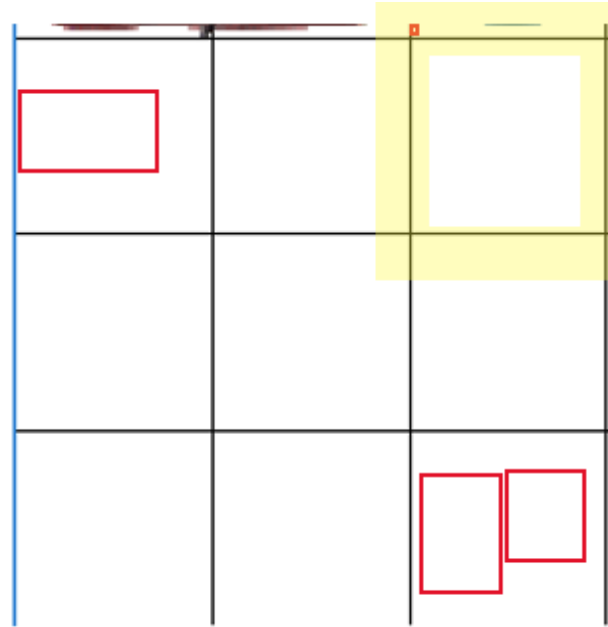


Figure 3: A sample board with three rows and columns.

You can test the `Location` class by writing a main method that creates a `Location` object and then prints out its row and column individually. Here is a main function that you can place inside your `Location` class and run in Eclipse. If you get the right print statements here, you can move on to step 3:

```
// Small test code to put in Location.java to check if your class works
public static void main(String[] args) {
    Location one = new Location(3, 4);
    Location two = new Location(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());
}
```

### Step 3) Implement the Vehicle constructor and getters

**Vehicle.java** This class is meant to represent a single vehicle, which will know its type, its current starting position, if it is vertical, and its length. The starting position is the top-left part of the vehicle and the rest of it occupies the spaces down or to the right.

Implementing `Vehicle` will be more work. As part of the `Vehicle` class, you will have to write the constructor, which should require the type, a starting row and column, whether it is vertical, and a length (in that order). Part of `Vehicle.java` will also be getters and setters – and trying to figure out what else you should store in a `Vehicle`. The only other functions that will require more thought and testing are the `move` and `locationsOn` functions in the next step.

#### Step 4) Implement `move`, `potentialMove` and `locationsOn` in `Vehicle`

**move** The purpose of having a `move` function in the `Vehicle` class is simply **to update the vehicle's own starting location**. You may think that this would be complicated (like actually moving the vehicle on a board or checking to see if it is in bounds), but since we are just dealing with the `Vehicle` class at the moment – and the `Vehicle` class knows nothing about the type or size of the board that it will be on – this makes it more modular. This means that we could use this class on some other kind of board in the future. It also helps us break down the problem into smaller parts, which allows us to test and implement our solution more easily.

The `move` function will take in a parameter ***numSpaces***, which is the number of spaces that the vehicle should move. Instead of having a separate indicator for the direction, we simply use a positive and negative integer to help the vehicle determine what its new start position should be and to update it. Since the vehicle knows whether its vertical, `move` is the first function where we are using some of the knowledge the vehicle has to update its own values.

**potantialMove** Similar to `move`, `potentialMove` is a function that works like `move`. However, it does not actually update the `Vehicle`'s own start row and column. Instead, it just returns a new `Location` with the starting row and column of where the vehicle would end up “if it were to move”.

*Both `move` and `potentialMove` do not have method declarations in the starter files, so you will have to add them in as well.*

**locationsOn** While you may be tempted to store all of the locations a vehicle occupies, this can lead to bugs (since you would have to keep track of multiple variables, you might for instance forget to update all of the locations). Instead of storing all the locations a vehicle is on, we will create a function that calculates those locations for us based on the starting location. This method, `locationsOn`, will return an array

of `Locations` that a vehicle is currently on top of. This can be determined using the starting position, the number of spaces, and its orientation.

For instance, for a vehicle with the following properties:

```
start: row 4 col 5
```

```
length: 2
```

```
vertical: true
```

`locationsOn()` would return an array with the following locations:

```
row 4 col 5; row 5 col 5
```

`locationsPath` One other helper method is `locationsPath`, which works like `locationsOn` but takes in an additional number. This number represents the number of spaces that a vehicle would move. `locationsPath` then returns an array of all of those locations that a vehicle would travel over in ascending order. Note that any locations that are currently occupied by the vehicle (`locationsOn`) should be excluded.

For instance, for a vehicle with the following properties:

```
start: row 3 col 3
```

```
length: 4
```

```
vertical: true
```

`locationsPath(-4)` would return an array with the following locations:

```
row -2 col 3; row -1 col 3; row 0 col 3; row 1 col 3
```

Similarly, for a vehicle with the following properties:

```
start: row 0 col 0
```

```
length: 2
```

```
vertical: false
```

`locationsPath(3)` would return an array with the following locations:

```
row 0 col 2; row 0 col 3; row 0 col 4
```

`toString` 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.

## Testing

Once you have implemented these methods, write 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 correct. The following code will allow you to test out the `locationsOn` and `locationsPath` functions. You also should also write additional test code for the move functions and include it in your code.

```
// This snippet would go inside of a public static void main in Vehicle.java
// Assume Vehicle constructor is type, startRow, startCol, isVertical, length
Vehicle someTruck = new Vehicle(VehicleType.TRUCK, 1, 1, true, 3);
Vehicle someAuto = new Vehicle(VehicleType.AUTO, 2, 2, false, 2);
System.out.println("This next test is for locationsOn:");
System.out.println("vert truck at r1c1 should give you r1c1; r2c1; r3c1
                    as the locations its on top of... does it?");
printLocations(someTruck.locationsOn());
System.out.println("horiz auto at r2c2 should give you r2c2; r2c3
                    as the locations its on top of... does it?");
printLocations(someAuto.locationsOn());
System.out.println("if we were to move horiz auto -2 it should give
                    you at least r2c0; r2c1... does it?");
printLocations(someAuto.locationsPath(-2));
// ADD SOME MOVE AND POTENTIALMOVE TEST CODE BELOW THIS LINE
```

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

```
// Prints out more legibly the row & columns for an array of locations
public static void printLocations(Location[] arr) {
    for(int i = 0; i < arr.length; i++) {
        System.out.print("r" + arr[i].getRow() + "c" + arr[i].getCol() + "; ");
    }
    System.out.println();
}
```

## UML Diagram

In order to help you keep track of everything that you need to write, we have provided you with a basic UML model below. This shows the methods that you should write (including

what to pass in and to return) and the instance variables that you should store in more detail. It does not include how you should write your constructors.

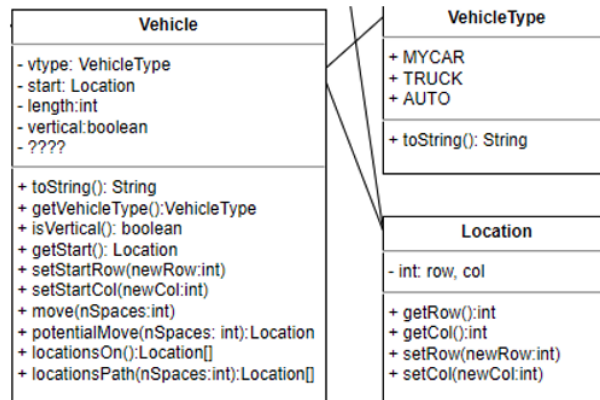


Figure 4: A UML diagram of the classes for this assignment.

## Deliverables & Advice

To submit this project, submit all of its contents as a .zip file. The best way to do this is via File → Export... in Eclipse and to export the entire project as a .zip file. We are looking for you to start programming on a regular basis. Please start early and, if something does not make sense, do not hesitate to ask.