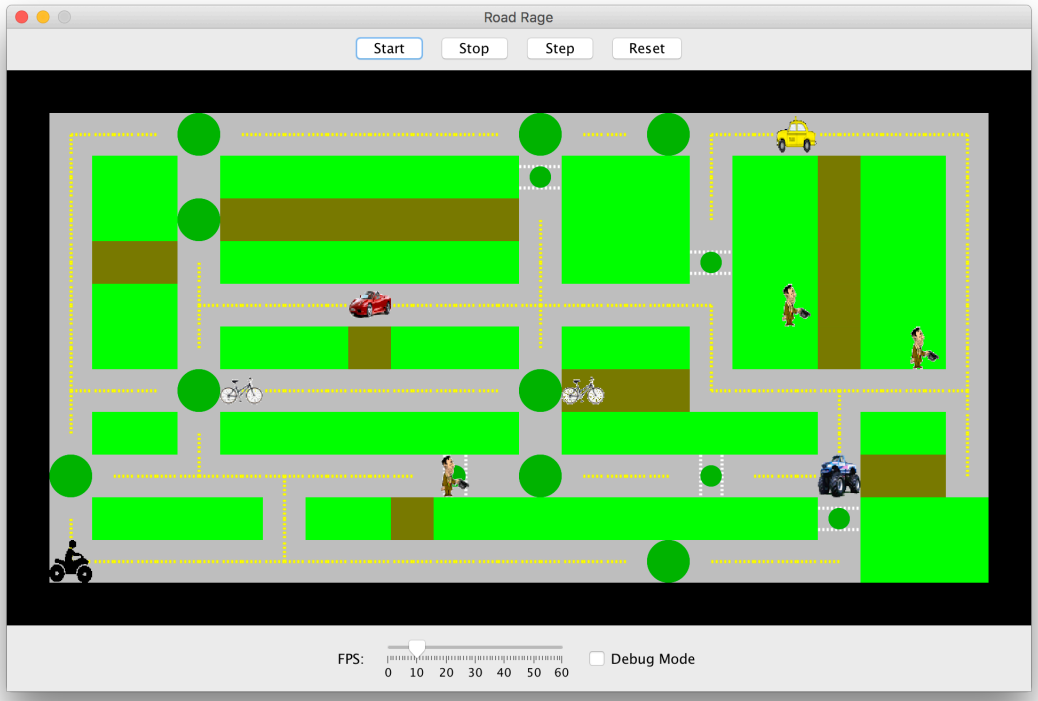
Institute of Technology, University of Washington Tacoma
TCSS 305 Programming Practicum, Autumn 2018
Assignment 3 – Road Rage
Value: 8% of the course grade
Due: Friday, 26 October 2018, 23:59:59

Program Description:

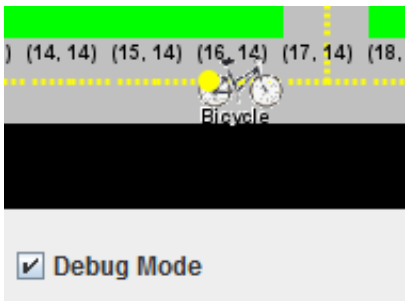
This assignment tests your understanding of inheritance, polymorphism, interfaces, and abstract classes. The program is a simulator of city streets, traffic lights, and vehicles.



Your task is to write the classes to model the various vehicles in the city. The classes will have many similarities, so you should use inheritance to design the classes effectively.

The starter code for the project includes a complete and correct graphical user interface (GUI). The GUI shows pictures for the map terrain and vehicles. The GUI loads the city map and vehicle starting positions from the provided file `city_map.txt`. There are buttons to start and stop the city's animation, as well as to step through it one frame at a time and to reset the vehicles to their initial layout. Each time the GUI redraws, it moves each of the vehicles. If any two vehicles end up on the same square, the GUI tells the vehicles that they have collided with each other, which may "kill" them temporarily. "Dead" vehicles appear upside down on the screen.

The rate of city animation is controllable via a slider at the bottom of the interface. Also, debug information can be shown using a checkbox. If debug information is enabled, all (x, y) grid positions on the map are labeled, and every vehicle's `toString` output is shown next to the vehicle. You can put any information you like into your vehicles' `toString` methods to help you debug their behavior.



You must write Vehicle sub-classes (and a common parent class called AbstractVehicle) for this assignment. Each of the Vehicle classes must implement the provided `Vehicle` interface described in this document. Note that a Human is a "Vehicle" for the purposes of this assignment. Vehicle behavior includes some randomness. If you need to create an object of type Random, then your project code should only create a single object of type Random that all Vehicle objects can share.

Vehicle Descriptions:

Truck

Constructor: `public Truck(int theX, int theY, Direction theDir)`
Images: `truck.gif` *(alive),* `truck_dead.gif` *(dead)*
Movement behavior: Trucks travel only on streets and through lights and crosswalks.

- Trucks randomly select to go straight, turn left, or turn right. *As a last resort*, if none of these three directions is legal (all not streets, lights, or crosswalks), the truck turns around.
- Trucks drive through all traffic lights without stopping!
- Trucks stop for *red* crosswalk lights, but drive through *yellow* or *green* crosswalk lights without stopping.

***Collision behavior*: A truck survives a collision with anything, living or dead.**

Car

Constructor: `public Car(int theX, int theY, Direction theDir)`
Images: `car.gif` *(alive),* `car_dead.gif` *(dead)*
Movement behavior: Cars can only travel on streets and through lights and crosswalks.

- A car prefers to drive straight ahead on the street if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible; as a last resort, it turns around.
- Cars stop for red lights; if a traffic light is immediately ahead of the car and the light is *red*, the car stays still and does not move. It *does not* turn to avoid the light. When the light turns green, the car resumes its original direction.
- Cars ignore yellow and green lights.
- Cars stop for *red* and *yellow* crosswalk lights, but drive through *green* crosswalk lights without stopping.

***Collision behavior*: A car dies if it collides with a living truck and stays dead for 15 moves.**

Taxi

Constructor: `public Taxi(int theX, int theY, Direction theDir)`
Images: `taxi.gif` *(alive),* `taxi_dead.gif` *(dead)*
Movement behavior: Taxis can only travel on streets and through lights and crosswalks.

- A taxi prefers to drive straight ahead if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible; as a last resort, it turns around.
- Taxis stop for red lights; if a traffic light is immediately ahead of the taxi and the light is *red*, the Taxi stays still and does not move until the light turns green. It *does not* turn to avoid the light. When the light turns green the taxi resumes its original direction. Taxis ignore yellow and green lights.
- Taxis stop for (temporarily) *red* crosswalk lights. If a crosswalk light is immediately ahead of the taxi and the crosswalk light is *red*, the Taxi stays still and does not move for 3 clock cycles or until the crosswalk light turns green, whichever occurs first. It *does not* turn to avoid the crosswalk light. When the crosswalk light turns green, or after 3 clock cycles, whichever happens first, the taxi resumes its original direction. A Taxi will drive through *yellow* or *green* crosswalk lights without stopping.

***Collision behavior*: A taxi dies if it collides with a living truck and stays dead for 15 moves.**

All-Terrain Vehicle (ATV)

Constructor: `public Atv(int theX, int theY, Direction theDir)`
Images: `atv.gif` *(alive),* `atv_dead.gif` *(dead)*
Movement behavior: ATVs can travel on any terrain except walls. They randomly select to go straight, turn left, or turn right. ATV's never reverse direction (they never need to). ATV's drive through all traffic lights and crosswalk lights without stopping!
***Collision behavior*: An ATV dies if it collides with a living truck, car, or taxi, and stays dead for 25 moves.**

Bicycle

Constructor: `public Bicycle(int theX, int theY, Direction theDir)`
Images: `bicycle.gif` *(alive),* `bicycle_dead.gif` *(dead)*
Movement behavior: Bicycles can travel on streets and through lights and crosswalk lights, but they prefer to travel on trails.

- If the terrain in front of a bicycle is a trail, the bicycle *always* goes straight ahead in the direction it is facing. Trails are guaranteed to be straight (horizontal or vertical) lines that end at streets, and you are guaranteed that a bicycle will never start on a trail facing terrain it cannot traverse.
- If a bicycle is not facing a trail, but there is a trail either to the left or to the right of the bicycle's current direction, then the bicycle turns to face the trail and moves in that direction. You may assume that the map is laid out so that only one trail will neighbor a bicycle at any given time.
- If there is no trail straight ahead, to the left, or to the right, the bicycle prefers to move straight ahead on a street (or light or crosswalk light) if it can. If it cannot move straight ahead, it turns left if possible; if it cannot turn left, it turns right if possible. *As a last resort*, if none of these three directions is legal (all not streets or lights or crosswalk lights), the bicycle turns around.
- Bicycles ignore green lights.
- Bicycles stop for yellow and red lights; if a traffic light or crosswalk light is immediately ahead of the bicycle and the light is *not green*, the bicycle stays still and does not move unless a trail is to the left or right. If a bicycle is facing a red or yellow light and there is a trail to the left or right, the bicycle will turn to face the trail.

***Collision behavior*: A bicycle dies if it collides with a living truck, car, taxi, or ATV. It stays dead for 35 moves.**

Human

Constructor: `public Human(int theX, int theY, Direction theDir)`
Images: `human.gif` *(alive),* `human_dead.gif` *(dead)*
Movement behavior:

- Humans move in a random direction (straight, left, or right), always on grass or crosswalks.
- A human never reverses direction unless there is no other option.
- If a human is next to a crosswalk it will always choose to turn to face in the direction of the crosswalk. (The map of terrain will never contain crosswalks that are so close together that a human might be adjacent to more than one at the same time.)
- Humans do not travel through crosswalks when the crosswalk light is green. If a human is facing a green crosswalk, it will wait until the light changes to yellow and then cross through the crosswalk. The human will not turn to avoid the crosswalk.
- Humans travel through crosswalks when the crosswalk light is yellow or red.
- Humans ignore the color of traffic lights.

***Collision behavior*: A human dies if it collides with any living vehicle except another human and stays dead for 45 moves.**

Implementation Guidelines:

The GUI's interaction with your various vehicle classes is the following:

- When the GUI initially loads, it creates several instances of each of your vehicle classes (this is why there are compile errors in the GUI as initially provided to you; it is trying to create instances of classes you have not yet written).
- The GUI draws each vehicle on the map by asking for its `getImageFileName` string, and then loading an image file from the current directory. So, if a `Human` object returns `"human.gif"` from its `getImageFileName` method, it will be drawn by showing that image.
- When you click the Start or Step buttons, the GUI updates the state of the overall map and the state of every vehicle. The Start button causes repeated updates, while the Step button causes a single update.
- On each update, the GUI calls the `chooseDirection` method on each living vehicle to see which way it *prefers* to move. The `chooseDirection` method parameter informs the vehicle about what terrain is around it. The vehicle uses this information to pick the direction it would like to move. Note that a vehicle may prefer to move in a direction that it is not *currently* able to move in. In particular, when stopped at a traffic light, the `chooseDirection` method can return the direction that would take the vehicle through the traffic light, while the `canPass` method (described next) reports that the vehicle cannot actually pass through the traffic light.
- After each vehicle reports its preferred direction of movement, the GUI controller checks each vehicle to see whether it can move in the preferred direction by calling the `canPass` method, passing the appropriate type of terrain and street light status. If the vehicle can traverse the given terrain with the given street light status, the GUI controller updates the vehicle's X and Y coordinates by calling its `setX` and `setY` methods, and modifies its direction by calling its `setDirection` method. If the vehicle cannot traverse the given terrain with the given street light status, it sits still for the round. NOTE: **Every live vehicle should move on every update cycle unless stopped at a street light or crosswalk light.**
- On each update, the GUI notifies any dead vehicles that an update has occurred by calling their `poke` method. Each type of vehicle has a predefined number of pokes after which it should revive itself; for example, a dead Taxi should revive itself after 10 pokes. Live vehicles are *never* poked. Vehicles should not move on the update in which they revive. Instead, they should revive after the correct number of pokes and face in a random direction. On the next update after they revive they should move.
- At predefined intervals, the GUI changes the map's lights in a cycle (green, yellow, red, green).
- Note: No vehicles will start on walls and no vehicles should at any time move onto walls.

Each of your vehicle classes must have a constructor as specified in the descriptions above. Do not change the number, type or order of the parameters. You may change parameter *names* if you choose.

Each vehicle must also implement the following methods of the instructor-provided `Vehicle` interface.

```
public Direction chooseDirection(Map<Direction, Terrain> theNeighbors)
```

A query that returns the direction in which this vehicle would *like* to move, given the specified information. Different vehicles have different movement behaviors, as described previously.

`theNeighbors` is a Map containing the types of terrain that neighbor this vehicle. The keys are instances of the `Direction` enumeration, such as `Direction.WEST`, and the values are instances of the `Terrain` enumeration, such as `Terrain.STREET` and `Terrain.GRASS`. The values are guaranteed to be non-null. To

access the terrain in a particular direction, retrieve it by specifying the direction you wish to check.

For example, to see whether the square to the west contains a street, one could write code such as the following:

```
if (theNeighbors.get(Direction.WEST) == Terrain.STREET) {// something}
```

OR to see whether the square to the left contains a street, one could write code such as the following:

```
if (theNeighbors.get(getDirection().left()) == Terrain.STREET) {// something}
```

`public boolean canPass(Terrain theTerrain, Light theLight)`
A query that returns whether this vehicle can pass through the given type of terrain, when the street lights are in the given state. Different vehicles respond in different ways, as described previously. For example, a Bicycle can pass the `Terrain.STREET` terrain type and can also pass the `Terrain.LIGHT` terrain type if the light status represented by `theLight` is `Light.GREEN`.

`public void collide(Vehicle theOther)`
A command that notifies this vehicle that it has collided with the given other Vehicle object. Different vehicles respond in different ways, as described previously. *Collisions should only have an effect when they occur*
between two vehicles that are alive. When the GUI notices that two vehicles have collided (whether they are both alive or not), it calls this method on each vehicle; the order in which this happens is not defined. Each vehicle should handle only the update of *its own* status and not the update of the other vehicle's status.

`public String getImageFileName()`
A query that returns the name of the image file that the GUI will use to draw this Vehicle object on the screen. For example, a living Taxi object should return the string `"taxi.gif"` and a dead Human object should return the string `"human_dead.gif"`.

`public int getDeathTime()`
A query that returns the number of updates between this vehicle's death and when it should revive. For example, a taxi should lie dead for 10 updates and then revive (on the 10th update). Calling `getDeathTime()` on a `Taxi` object should always return 10.

`public Direction getDirection()`
A query that returns the direction this vehicle is facing, one of `Direction.NORTH`, `Direction.SOUTH`, `Direction.EAST`, or `Direction.WEST`.

`public int getX()`
`public int getY()`
Queries that return the *x* and *y* coordinates of this vehicle.

`public boolean isAlive()`
A query that returns whether this vehicle is alive; that is, if it has not collided with a more powerful vehicle and gotten killed. Killed vehicles revive themselves after a certain number of turns, as described previously.

`public void poke()`
A command called by the graphical user interface once for each time the city animates one turn. This allows dead vehicles to keep track of how long they have been dead and revive themselves appropriately. Live vehicles are *never* poked.

When a dead vehicle revives, it must set its direction to be a random direction.

The static method `Direction.random()` is useful for this purpose.

`public void reset()`
A command that instructs this Vehicle object to return to the initial state (including position, direction, and being alive) it had when it was constructed.

`public void setDirection(Direction theDir)`
A command that sets the movement direction of this vehicle. This should only be called by the GUI controller, and (possibly) your `poke()` and/or `reset()` methods to set direction at revival or reset.

It should never be called by `canPass()` or `chooseDirection()`

`public void setX(int theX)`
`public void setY(int theY)`
Commands that set the *x* and *y* coordinates of this Vehicle. These should only be called by the GUI controller, and (possibly) your `reset()` method. It should never be called by `canPass()` or `chooseDirection()`

Provided Enumeration Types:

The following enumeration types are provided to you (there are additional methods in some of them that are used internally by the provided code, but that you will likely not need to use).

```
public enum Direction {
    NORTH, WEST, SOUTH, EAST;

    // returns a random direction
    public static Direction random()

    // returns the direction 90 degrees counter-clockwise from this direction
    public Direction left()

    // returns the direction 90 degrees clockwise from this direction
    public Direction right()

    // returns the direction opposite this direction
    public Direction reverse()
}

public enum Light { GREEN, YELLOW, RED }

public enum Terrain { GRASS, STREET, LIGHT, WALL, TRAIL, CROSSWALK }
```

You should use these enumeration types to implement the behavior of your vehicles. The `Light` and `Terrain` enumerations contain constant values, so they would be used in your code in tests such as "`if (light == Light.GREEN)`". The `Direction` enumeration also contains a few methods, such as the static method `Direction.random` and the instance methods `left`, `right`, and `reverse`. You should use these methods in your code whenever possible to describe the vehicles' movement behavior. For example, to check whether your vehicle's neighbor in its "left" direction (the direction it would be facing if it turned left) is a street, you would write code like the following:

```
if (theNeighbors.get(getDirection().left()) == Terrain.STREET)
{
  // do something
}
```

You can learn more about Enum types in the Core Java and Effective Java books. You can also learn more in the Oracle tutorial on Enum types.

Inheritance Hierarchy Guidelines:

Since a main purpose of this assignment is to demonstrate your knowledge of inheritance, you must use inheritance to reduce redundancy among your vehicle classes. Specifically, you must implement a parent class for the concrete vehicle sub-types. The parent class should contain *as much of the vehicle behavior as possible*; that is, all the behavior that can be made common among the sub-classes.

These are the *required* features of your inheritance hierarchy:

- The parent class must not be instantiable and must implement the `Vehicle` interface.
- The parent class must contain all common instance fields shared by the vehicle sub-types.
- The parent class must have a protected constructor that initializes these fields appropriately. Note that this constructor need not have the same signature as the individual vehicle class constructors (it can take more, or fewer, parameters as you choose. Think about what data would be useful to pass from the child class constructors to the parent class constructor.)

- The parent class and all other classes must be properly encapsulated: instance fields *must* be private.
- The parent class *must not* explicitly contain information about all the child classes; for example, the parent class should not store the names of every child class's image files or the number of moves until every type of vehicle revives. It is acceptable for information specific to a particular instance to be passed from the child classes to the parent class constructor upon construction of the object (and stored in a field). It is not acceptable, under *any* circumstances, for any of your classes to perform an `instanceof` test (or equivalent) to determine proper behavior.
- toString() must return a reasonable String representation for each vehicle, which means that you must override toString(). This would ideally be done once in the parent class or in each child class individually if there is a good reason for doing it differently for each child class.
- You may NOT add any new *public* or *package-level* methods or any new abstract methods to the Vehicle interface or to the AbstractVehicle class (except public toString()). You may add *protected* non-abstract methods to the AbstractVehicle class if such methods are used by more than half of the concrete Vehicle child classes (so that the new method represents a default behavior for a majority of Vehicle types). You may, of course, add private methods to any of the Vehicle classes if you wish.
- Your Vehicle child classes may contain *only* fields used only by that sub-class, constants used by only that sub-class, constructor(s) and implementations of their respective `canPass` and `chooseDirection` methods. All other data and behavior should be moved upward into the parent class. A child class may declare data fields or helper methods *if they are needed solely by that vehicle type*, but such fields or methods *must* be declared private; you may not add public, protected, or package-level methods or fields to the Vehicle subclasses.
- The javadoc comments for methods implemented in the vehicle child classes must clearly indicate the specific behavior for each vehicle child class. (You should not simply inherit javadoc from the parent class because each vehicle child class implements these methods in a unique way.)

Unit testing:

A unit test for class **Human** has been provided (HumanTest.java) with tests of the canPass and chooseDirection methods (and for several inherited methods). When you complete your Human class and AbstractVehicle class, these tests should pass when run on your code. (Since these tests were developed as 'black box' tests, they may or may not provide complete code coverage of your Human class, depending on how you write your code. I suggest that you should NOT use the code coverage tool with my tests. Just ensure that my tests pass when run on your code.)

You are **required** to write unit tests for the **Truck** class. The unit tests that you write for your Truck class will be 'white box' tests and therefore should achieve full code coverage of all methods which you choose to implement in the Truck class. It is highly recommended (but not required) that you write unit tests for all Vehicle subclasses. A unit test for a subclass could also test the inherited methods implemented in AbstractVehicle.

Hints:

It is a challenge to move so much behavior upward into your parent class, especially when much of it seems to be different depending on which child class is being used. The following hints may help:

- *To implement collision behavior in the parent class*: Since each vehicle knows its death time, you can compare them and have your vehicle "die" if it collides with another vehicle with a smaller death time.

- *To implement the image file name behavior in the parent class*: Calling the method

  ```
  getClass().getSimpleName().toLowerCase()
  ```

  yields a convenient `String` that is the same as the current object's class name, in lowercase; for example, the above code, when run on an object of the `Bicycle` class, would yield the string `"bicycle"`.

- *To implement reset in the parent class*: When the object is constructed, remember its initial position and direction for restoration later if `reset` is called.

- *To make your parent class generalized so that it can work with all child classes*: Think about what exactly is unique about each vehicle type. Each has a unique set of terrain it can pass, and each has a unique way of choosing which direction it would like to move. You'll represent this by putting the `canPass` and `chooseDirection` methods in the child classes.

  In addition, you should put a constructor in each child class that initializes the object by calling the parent class's constructor. The parent class needs to implement behavior such as the collisions, so consider passing your vehicle's death time and other needed information upward as arguments to the parent class's constructor.

- Are there any fields which will never change after the Vehicle is instantiated? If so consider making those fields final. Are there any classes in your project which will never be extended? If so, consider making such classes final. Are there any methods which will never be overridden? If so, consider making them final.

NOTE: ***Cyclomatic complexity*** (CC) is a statistical measure of software complexity and is directly related to the number of possible executable paths through a section of code (usually a method). When the CC value of a section of code is high, that is a general indication that it has become too complex to be easily understood by others who may later need to maintain the code. The Metrics tool (and sometimes other tools) will warn for high CC values. Ways to lower complexity include reducing the number of exit points in a method, reducing the number of nested if/else statements, reducing the number of switch statements cases, reducing the use of nested loop structures, and reducing the use of recursion. Often the simplest way to reduce the CC value of a complex method is to create a helper method to do part of the work. For this assignment (and all assignments in this course) you will lose points if any part of your code has CC > 10.

Submission and Grading:

Create your Eclipse project by downloading the `hw3-project.zip` file from the Assignment 3 page on Canvas, importing it into your workspace (as described for `hw0-project.zip` in Assignment 0), and using "Refactor" to change "username" in the project name to your UWNetID. Remember to make this change ***before*** you first commit the project to Subversion.

Also as with Assignment 0, you can use whichever bracket style you like. The provided templates and classes use the same line style; if you switch to the next line style, execute a "Format" command (in Eclipse's "Source" menu) on your project. You may need to perform some manual format cleanup after doing this.

You must check your Eclipse project into Subversion (following the instructions from Lecture 1 and Assignment 0), including all configuration files that were supplied with it (even if you have not changed them from the ones that were distributed). If you have any questions about this, ask the instructor as soon as possible. When you have checked in the revision of your code you wish to submit, make a note of its Subversion revision number. To get the revision number, *perform an update* on the top level of your project; the revision number will then be displayed next to the project name. Your revision number will pick up where you left off on Assignment 2; if you submitted revision 27 of Assignment 2, the first commit of Assignment 3 will have a number greater than 27. This is because you have a single Subversion repository for all your projects, and the revision number counts revisions to the entire repository.

After checking your project into Subversion, you must submit (on Canvas) an *executive summary*, containing the Subversion revision number of your submission, an "assignment overview" (1 paragraph, up to about 250 words) explaining what you understand to be the purpose and scope of the assignment, and a "technical impression" section (1-2 paragraphs, about 200-500 words) describing your experiences while carrying out the assignment (especially any difficulties you had installing or using the tools). The assignment overview shows how well you understand the assignment; the technical impression section helps to determine what parts of the assignment need clarification, improvement, *etc.* for the future.

The filename for your executive summary must be "**username-roadrage.txt**", where username is your UWNetID. An executive summary template, which you *must* use, is available on Canvas. In particular, your executive summary must have a line "Subversion Revision Number: #", with no leading spaces, where "#" is the Subversion revision number you made a note of above (with no parentheses or other symbols). Executive summaries without a line following this exact format will be penalized. Executive summaries will *only* be accepted in plain text format – other file formats (RTF, Microsoft Word, Adobe PDF, Apple Pages) are *not* acceptable.

Part of your program's score will come from its "external correctness." For this assignment, external correctness is graded on the vehicles' behavior, which is observed both through automated testing and by running the GUI and examining the result. Exceptions should not occur under normal usage. Your program should not produce any console output.

Another part of your program's score will come from its "internal correctness." Internal correctness includes meaningful and systematically assigned identifier names, proper encapsulation, avoidance of redundancy, good choices of data representation, the use of comments on particularly complex code sections, and the inclusion of file header comments in your classes. *Internal correctness also includes whether you design your vehicle classes and abstract parent class as described in these instructions.*

Note that the supplied RoadRageGUI class contains errors and warnings (the errors and warnings should disappear when you implement your vehicle classes). Your submitted classes should ideally have no warnings. If you feel that you are justified in coding something in a way that generates a warning, it is always best to ask about it before submitting it. Internal correctness also includes criteria such as the presence of Javadoc comments on *every* method and field (even private ones!), the use of variable names, spacing, indentation, and bracket placement specified in the class coding standard, and the absence of certain common coding errors that

can be detected by the tools. It is therefore to your advantage to be sure the plugin tools like your code before you submit it.

You will, in general, not lose any points on your executive summary itself unless you fail to turn it in, it does not meet the minimum length requirements, or it is trivial.

For this assignment, the percentage breakdown is 10% executive summary, 55% external correctness, and 35% internal correctness.