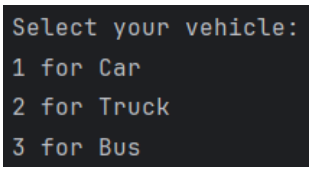
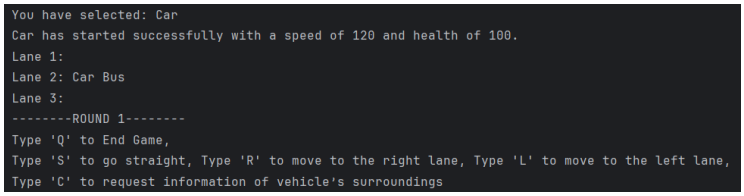
SUMANT PATEL (6796841) and ROHAL KABIR (7105836)

COSC 3P91 – Assignment 2

1 HOW TO PLAY

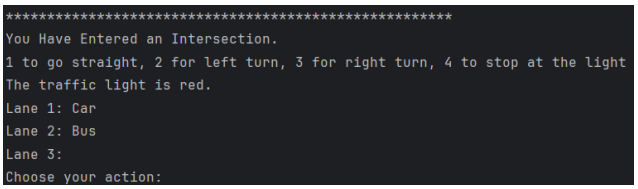
Once the program has been started either through a Java IDE or terminal a screen will appear (left image) to select the player's vehicle. After Selecting the vehicle the image to the right will appear, to confirm the selection, show the current map status with other entities(bots) and finally the Round 1.

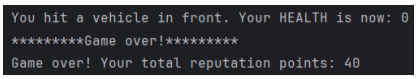




Round 1 is part of a series of rounds that will create challenges for the user to either switch lanes or continue straight with the option of requesting a surrounding view ("c"). This will go on till an intersection has been reached.

If the player makes a choice that violates traffic laws or collides with another vehicle not only will they take damage but a message will update them about their fault and provide the current health.





Finally the vehicle has entered an intersection. The Player will be prompted with a variety of decisions. The player must keep in mind traffic laws such as the color of the light and what turn you can do in what lane (example: Can't turn left in right lane). Not obeying these rules will result in health loss.

When the health has reached 0 the game is over. A reputation score will be made available, which would have added points for every correct choice that was made during game play.

Authors' address: Sumant Patel (6796841); Rohal Kabir (7105836).

Update date: 3 March 2024

2 GAMEMAP PACKAGE OVERVIEW

The GameMap Package is responsible for representing and managing the game environment in which the vehicles move and navigate. This includes the layout of the lanes, intersections, traffic lights, and traffic signals, which ensures that the game is accurately simulating a road network.

2.1 Lane Class

The Lane class focuses on the cars that occupy each lane, and is made to handle separate lanes on the game map. It stores references to Vehicle objects in a **LinkedList (java util class "collections")**, making it possible to add and remove objects quickly as cars enter and exit lanes during gaming. Vehicle movement is made easier by this data structure selection, which enables fast changes to the lane's condition at negligible performance. The class has three primary methods: addVehicle to add a vehicle to the lane, removeVehicle to remove a vehicle from the lane, and constructor to initialize the lane with an empty list of cars. It also has a getVehicles function that gives the list of automobiles in the lane as of right now. The ability to make decisions in advance regarding vehicle placement and movement, such as detecting crashes or controlling vehicle speeds depending on lane occupancy, makes this arrangement crucial for simulating traffic flow within the game.

2.2 Map Class

The Map class is made to simulate the game environment, it focuses on how lanes and intersections are laid up and controlled so that cars may drive through them. Upon construction, the game map is initialized with 3 lanes, which are dynamically stored in an **ArrayList (java util class "collections")**. To prevent problems caused by erroneous lane references, the class performs necessary features like adding and deleting cars from lanes only if the given lane numbers are valid. It also has a way to show how the map is now looking, so you can see which cars are in each lane. This helps to improve the UI of the game by giving you a clear picture of the gaming area. The ability to determine if a certain car is in front of another vehicle in a lane is another essential feature that is required for the game's collision detection and navigation logic to work. This class serves as the game's fundamental element, enabling the fundamental dynamics of vehicle movement and interaction. This utilizes Java **Generics,** primarily in the manipulation and management of collections, such as lists of lanes and intersections within a game map context. Through declarations like **private List<Lane> lanes and private List<Intersection> intersections,** the code specifies that these collections will contain objects of type Lane and Intersection respectively, ensuring type safety by enforcing that only the specified type of objects can be added or retrieved, thereby eliminating the need for type casting and reducing runtime errors. The **use of generics is evident in the instantiation of these collections with new ArrayList<>(), where the diamond operator <> leverages type inference for cleaner syntax.**

2.3 Traffic Light Class

This simulates the traffic lights at the intersection, which are crucial for controlling the flow of vehicles through intersections to ensure orderly traffic movement. Here we are able to set the color of a light and check what the light color is which is needed in the main class. **An anonymous class is created that extends the TrafficLight class. It overrides the setSignal method of the TrafficLight class to add additional behavior whenever the traffic signal is set. This use case for anonymous classes, allows for customization of behavior with minimal syntax and without the need for a separate named class definition.**

2.4 Traffic Signal Enumeration

This defines the possible states of a traffic light as an enumeration, which is used by the traffic light class to represent the state of the traffic light. It is also used across the package to standardize the representation of the states.

3 PLAYER PACKAGE OVERVIEW

The Player Package controls all aspects of the player's experience in the game, including decisions, actions, and controlled vehicles. With the ability to include user input and have bots-controlled vehicles that make automatic choices.

3.1 MyPlayer Class

The MyPlayer class, which mostly focuses on interactions about vehicle selection. To hold the player's selected vehicle, it encapsulates an instance of a Vehicle. The class has a method called **'selectVehicle'** that allows the player to choose their vehicle from a console-based menu when it is initialized without any parameters. To obtain user input, this approach uses a scanner to select between three distinct vehicle categories (car, truck, and bus), denoted by the numbers 1, 2, and 3, respectively. The selection procedure is done in a loop to make sure the player selects the right option before moving further. A good selection results in the instantiation and assignment of the matching Vehicle object (Car, Truck, or Bus) to the vehicle attribute. The vehicle's name and initialization procedure are printed on the console. This configuration incorporates user input into the game's flow by giving players an interactive way to choose their vehicle at the beginning of the game. Interactions between the player's car and the game world are made easier by the **'getVehicle'** method, which gives other game components access to the vehicle the player has chosen.

3.2 Bots Class

The purpose of the bots class is to mimic bots, in the game, with a particular emphasis on the vehicle selection feature. It provides a method called selectRandomVehicle that uses the Java Random class to produce a random integer between 0 and 2, which corresponds to the various vehicle types described in the VehiclePackage (car, truck, and bus). The method chooses and returns an instance of one of these vehicle kinds based on the produced random number. This selection process is made easier by the switch-case structure, which guarantees that every potential random value corresponds to a different type of vehicle. In the unusual event that the random number deviates from the predicted range, Car is the default option. With the help of this system, bots may play the game with a randomly chosen vehicle, which adds unpredictability and diversity to the gameplay experience by mimicking various opponent strategies without requiring human involvement.

4 VEHICLE PACKAGE OVERVIEW

4.1 Vehicle Class

Vehicle is an abstract class; all vehicle subtypes(Car, Bus, Truck) in this game inherit from vehicle. It contains the fundamental characteristics and features shared by all cars, such as their name, speed, and state of health. Initializing these characteristics involves passing arguments for the vehicle's name, speed, and maximum health to the class constructor. Each vehicle has its own health and speed. In order to control the vehicle's health and make sure it stays below the designated maximum health number, it additionally constructs a Health object. It also has a method called initializeAndStart() that is in charge of starting the car and confirming that it has completed initialization. **The local class, "PreStartCheck" is used within the "initializeAndStart" method** to encapsulate the logic for checking whether the vehicle has been properly initialized. This demonstrates the use of a local class for a specific purpose within a method, while the broader starting logic is handled directly in the method itself.

4.2 Health Class

The Health class manages the health of player vehicles in the game. It tracks two main attributes: maxHealth, representing the vehicle's maximum health capacity, and currentHealth, indicating the current health level during gameplay. The class initializes these values upon instantiation and provides methods to retrieve both current and maximum health. Additionally, it offers a decrease method to simulate collision damage, reducing the vehicle's health by a specified amount. This encapsulated approach ensures effective health management and facilitates the implementation of collision mechanics within the game.

4.3 Vehicle Reputation Class

The VehicleReputation class is responsible for overseeing the reputation points of the cars in the game. To keep track of the overall reputation points, it keeps an integer variable called points. Custom operations on reputation points are made possible by the class's flexible approach, which is enabled through the **PointsOperation interface which encompasses the Lambda Expression functionality.** Reputation points are initially set to 0. Using the operate method specified in the PointsOperation interface, the adjustPoints method allows reputation points to be dynamically adjusted. Reputation points may be changed in a flexible way depending on interactions or game events thanks to this approach. Lastly, the current total points are retrieved using the getPoints function. All things considered, the class offers a strong foundation for determining and controlling vehicle repute in the game, improving gameplay dynamics and user involvement. In this scenario, adjustPoints becomes a versatile method that can add the points based on **the Lambda Expression passed (which is used in the main class) to it.**

5 MAIN CLASS

The main code acts as the master controller for the game in which users steer cars through junctions and lanes. The first step is setting up the gaming environment, which includes the player, bots, and the three-lane layout. After the program is initialized, the player's movements in lanes and junctions are handled in a loop until the player chooses to quit the game or their health reaches zero. Methods like laneWork() and intersectionWork(), which ask the player to move, check for collisions or traffic infractions, and update the game state appropriately, manage the player's activities during each iteration of the loop. The player's health is monitored by additional methods like AmIALive(), and traffic offenses are **handled by the TrafficException class.** In the end, the primary code controls how the game progresses, how players interact with it, and how the game ends depending on what the player chooses to do and how they interact with the game environment. This makes for an immersive and interesting gameplay experience. The reputation object's adjustPoints() method **receives the lambda expression as an input. This method accepts a functional interface as its parameter.** Adjusting reputation points based on current points and points to add is defined by the single abstract method adjust() defined by this functional interface. The player's reputation points are updated by incrementing the current points by the points to add inside the lambda expression.
E.g of Lambda Expression in the code:
reputation.adjustPoints((currentPoints, pointsToAdd) -> currentPoints + pointsToAdd, 5);

6 TRAFFIC EXCEPTION CLASS

The **TrafficException class is a custom exception designed to handle violations of traffic laws** within the game simulation. It extends the built-in Exception class, inheriting its properties and behaviors while adding specific functionality tailored to traffic violations. In the constructor of TrafficException, a message is provided to describe the nature of the violation, which in this case is "You have violated traffic laws." This message will be displayed when the exception is thrown, providing clarity to the user about the reason for the exception. When the player tries to make actions that break traffic regulations, such as passing a red light, instances of TrafficException are thrown in the main code, notably within the **IntersectionWork() method.** A new TrafficException object is constructed and thrown using the **throw keyword** when such a violation takes place. This interrupts the program's regular execution and moves control to the closest enclosing **try-catch block**, or ends it if the attempt is not successful.