CST 239 Activity Report 1 Patrick Buhmbi 2/3/2024

Part 1: Tools Installation and Validation .....

a. Screenshot of the Eclipse About Box.

b. Screenshot of console output when running the HelloWorld class.

Part 2: Designing, Coding, and Documenting a Person Class .....

a. Theory of operation write-ups.

b. UML class diagram of the Person class.

c. Screenshot of the console output when running the Person class.

d. Submit a zip file of your source code, including the generated Javadoc files.

Part 3: Designing, Coding, and Documenting a Race Car Class .....

a. UML class diagram of your complete model.

b. Screenshots demonstrating that you can start, drive, and stop your Car.

c. Submit a zip file of your source code, including the generated Javadoc files.

Part 4: Using the Debugger .....

a. Screenshot from the Setting Breakpoints task.

b. Screenshots from the Inspecting Variables task.

c. Screenshots from the Stepping task.

d. Screenshot from the Inspecting Call Stack task.

Part 1 Tools Installation and Validation.

a. Screenshot of the Eclipse About Box.

For this task, I kicked things off by grabbing the most up-to-date version of Eclipse for Java from their website (https://www.eclipse.org). Once I had it downloaded, I ran the installer. I made sure to select the option for Eclipse IDE for Java Developers, followed the instructor's advice by picking the right version from the Java VM dropdown menu, and then chose the folder where I wanted Eclipse to be installed. With that done, I clicked the Install button and let the installation process do its thing. Once it was all set up, I fired up the Eclipse IDE. To wrap things up, I captured a screenshot of the About Box in Eclipse for Java.

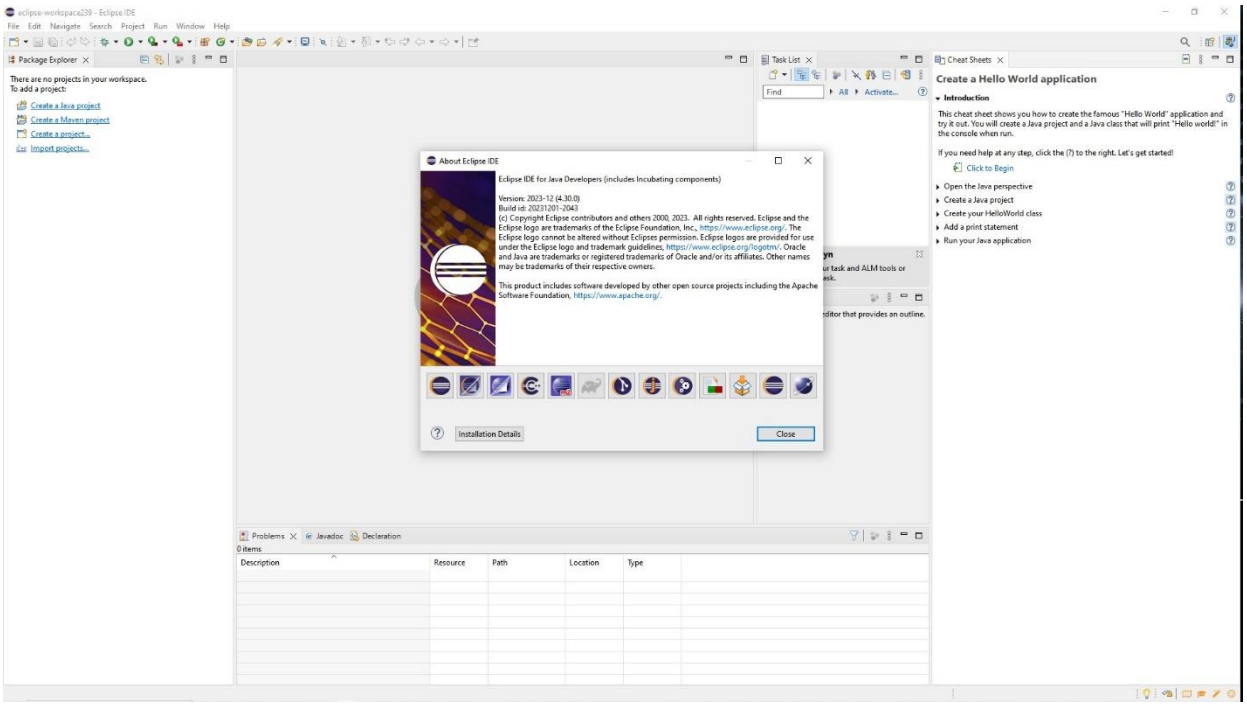

Figure 1 shows the Screenshot of the Eclipse About Box.

b. Screenshot of console output when running the HelloWorld class

First, I installed the Eclipse IDE and established a fresh Eclipse Workspace named workspaceCST-239, designated for all tasks. Then, I adhered to the instructions outlined in Activity 1 by initiating a new project titled topic 1-1. Within a designated source folder, I crafted a class dubbed HelloWorld. Subsequently, In the code, I instructed the program to display the message "Hello my name is" along with the provided name parameter. Finally, I directed the program to output this message to the system console.
Figure 2. below shows the Screenshot of console output when running the HelloWorld class

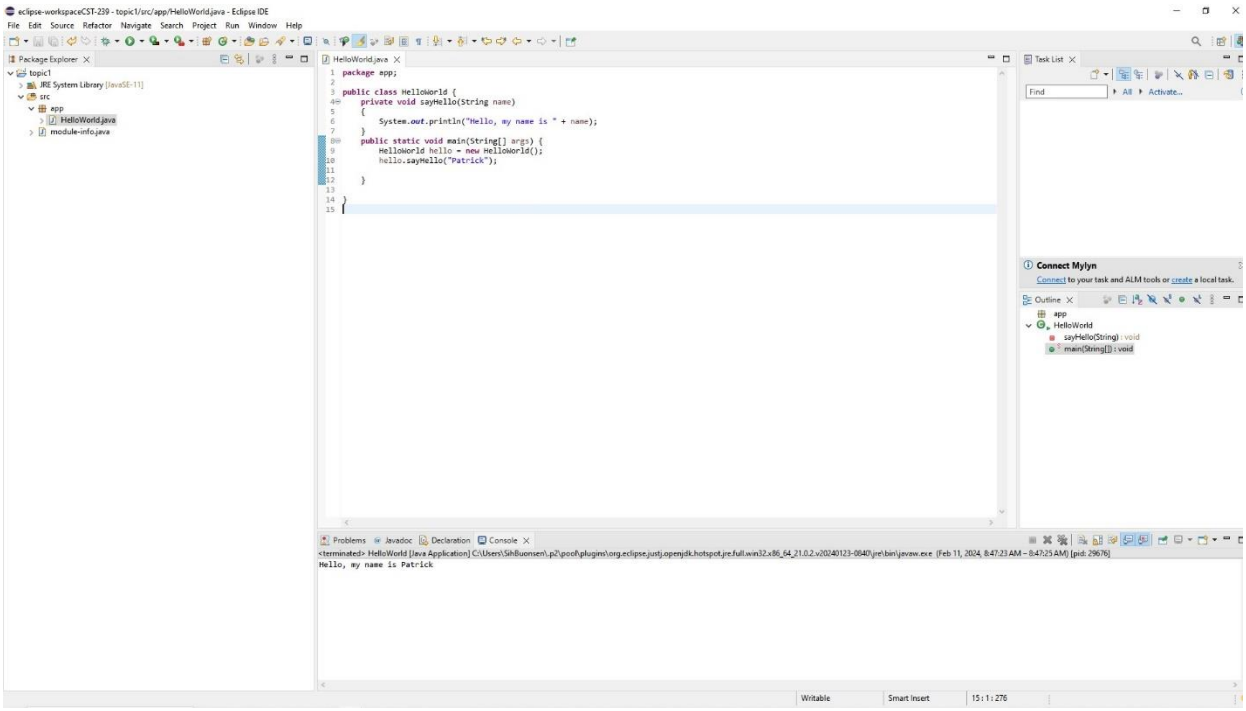

Figure 2. below shows Screenshot of console output when running the HelloWorld class,

Part 2: Designing, Coding, and Documenting a Person Class

Theory of operation write-ups.

The theory of operation lays out the steps needed to create and develop a Person class in Java, highlighting the importance of planning, setting up the project, writing code, and testing. Initially, it's crucial to sketch a UML diagram representing the class's attributes and behaviors, like the person's name, age, and weight. This diagram guides the coding process and can be crafted using tools like Microsoft Word or Microsoft Visio. Setting up the Java project in Eclipse ensures a structured environment for coding, ensuring compatibility with JavaSE-11 JRE. Then, create the Person class, adding private member variables for attributes and a constructor to initialize them. Use Eclipse's Refactoring tools to generate getter and setter methods for data access and ensure data integrity. Additionally, include behavior methods for defining class functionality. Console print statements are integrated into each method for testing and debugging purposes. Finally, by invoking public methods within the main() method, the program displays outputs in the console, affirming the correct functionality of the Person class. This systematic approach guarantees the class operates as intended, ensuring code integrity and functionality.

Provide a brief (3- to 4-sentence) description of how and why the output was displayed.
The output was displayed because each method in the Person class contained console print statements that were executed when the methods were called. These print statements were

added for testing purposes and to provide feedback on the execution of each method. By invoking the methods within the main() method and creating an instance of the Person class, the program ran through each method, displaying the appropriate testing messages to the console, thereby verifying the functionality of the class.

UML class diagram of the Person class

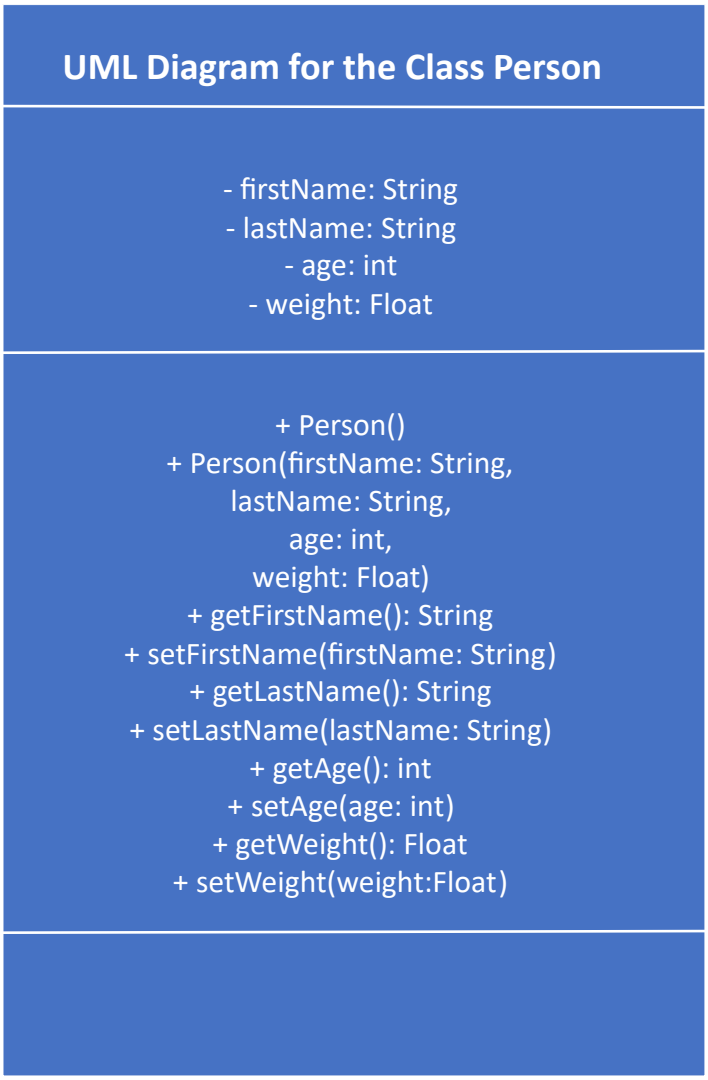

Figure 3 shows the UML diagram for a Person class.

The UML diagram for the Person class gives a clear picture of the traits and actions linked with the Person class. Let's break it down:
Class Header: At the top, you see the class name, which is "Person" here. It's like a template for making Person objects in the program.

Attributes Section: Underneath the class name, you find a list of characteristics or features of the Person class. Each one is marked with a hyphen (-), showing they're private, only for use within the class. These features include:

- firstName: String
- lastName: String
- age: int
- weight: double They represent details like a person's first name, last name, age, and weight.

Methods Section: After the attributes, you'll see a list of actions related to the Person class. Each action has a plus sign (+), indicating they are public and can be used from outside the class. The actions include:

- firstName(): String
- lastName(): String
- age(): int
- weight(): double
- setFirstName(firstName: String): void
- setLastName(lastName: String): void
- setAge(age: int): void
- setWeight(weight: double): void These actions let you do things with a person object, like getting their first name or updating it. In short, the UML diagram of the Person class is like a map showing the makeup and abilities of a person in the program. It helps programmers see what details they can store about a person and what actions they can perform with those details. This diagram makes it easier for developers to create, understand, and use the Person class effectively in their Java projects.

Screenshot of the console output when running the Person class.

Below (Figure 4) is a screenshot of the console output when running the Person class. On the right side of figure 4 is the output when the program was executed.

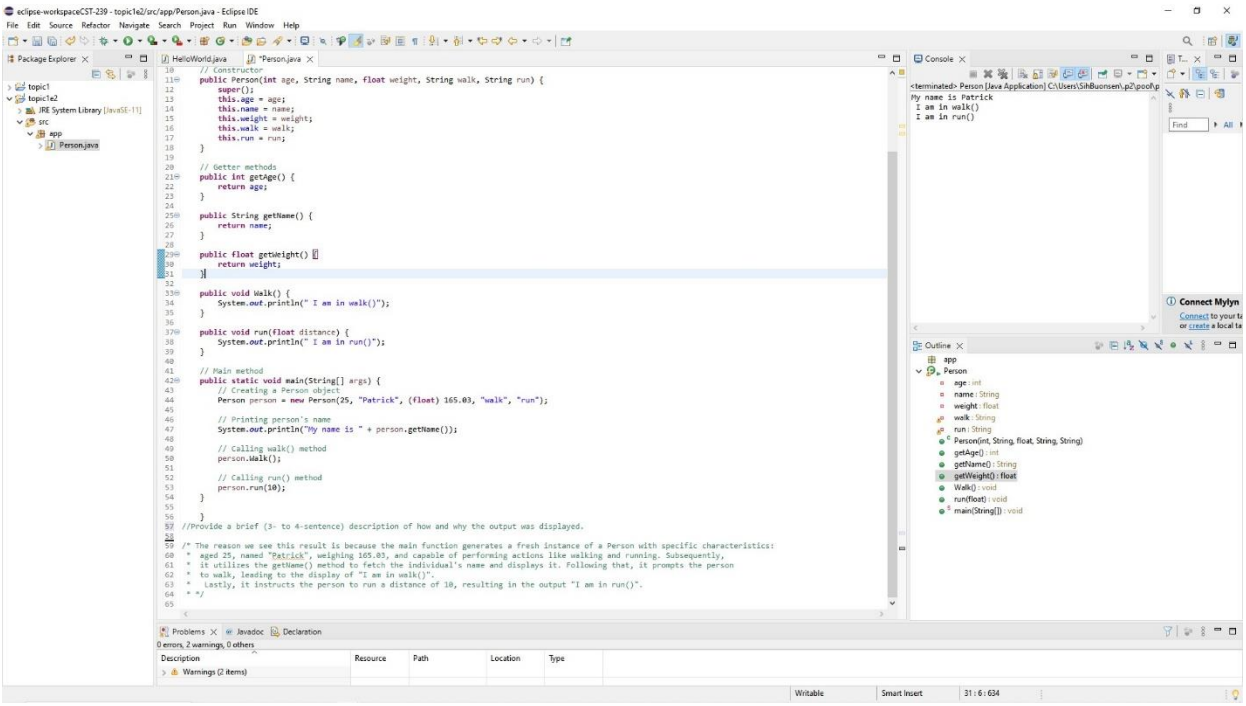

Figure 4. screenshot of the console output when running the Person class.

Submit a zip file of your source code, including the generated Javadoc files

Please see the zip file(CST-239Activity1_2) CST-239Activity1_2.zip This zip file contain the Person class code and the generated JavaDoc files.

Part 3: Designing, Coding, and Documenting a Race Car Class

UML class diagram of your complete model.

The UML diagram for the Car Race Game classes provides a clear overview of the characteristics and behaviors associated with each class. Let's simplify it:

Engine Class:

Attributes:

- **running**: boolean - This indicates whether the engine is currently running (true) or not (false).

Methods:

- There aren't any methods specified in the diagram, but we can assume the Engine class includes methods for starting, stopping, and restarting the engine.

Tires Class:

Attributes:

- pressure: int - This represents the pressure of the tires.

Methods:

- checkPressure(): This method checks the pressure of the tires.

Car Class:

Attributes:

- engine: Engine - This is the car's engine, represented as an Engine object.
- tires: Tires - These are the car's tires, represented as a Tires object.
- speed: double - This indicates the speed of the car.

Constructor:

- Car(engine: Engine, tires: Tires): This initializes a Car object with a specific engine and tires.

Methods:

- start(): Begins the car's engine.
- run(speed: double): Drives the car at a specified speed.
- stop(): Halts the car's movement.
- restart(): Restarts the car's engine.

UML Car Race Game

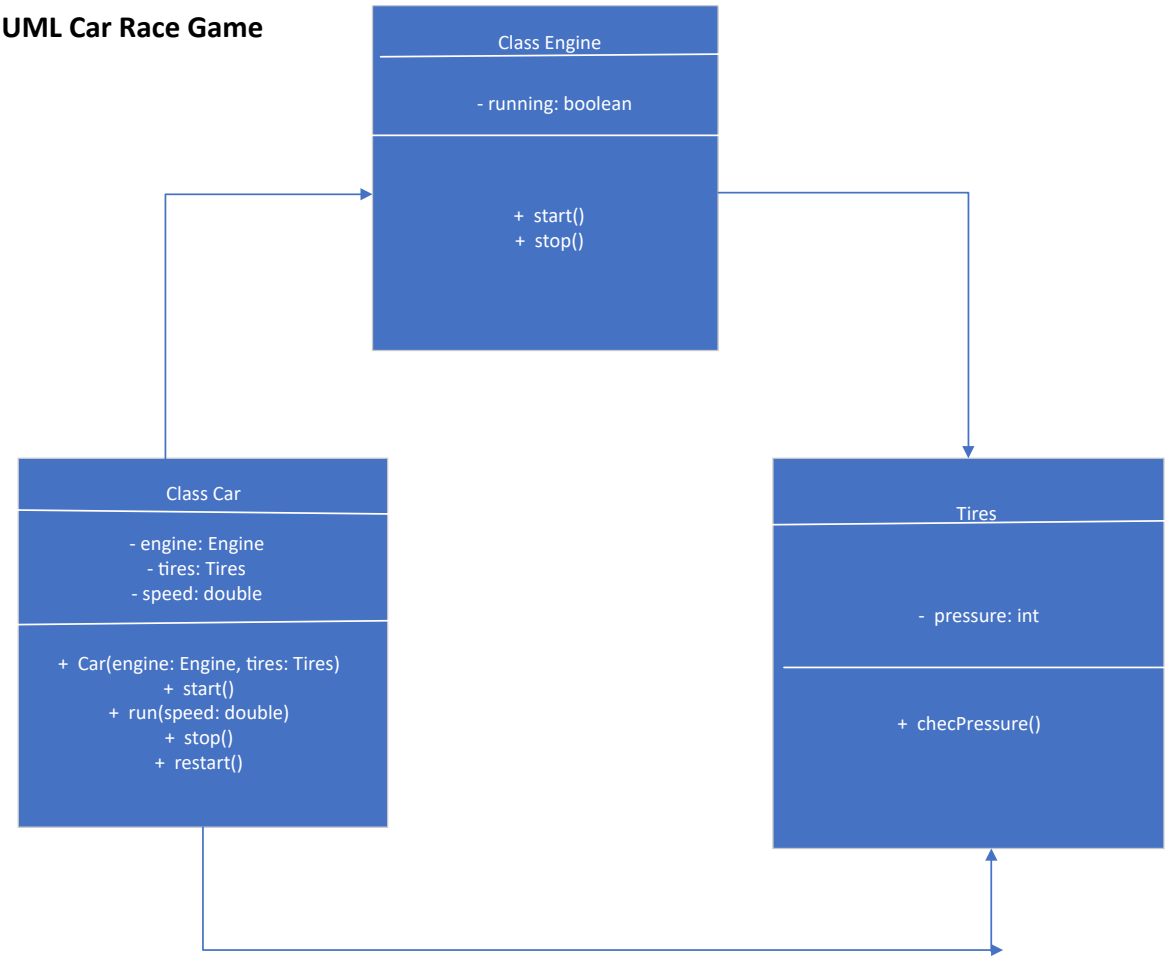

Figure 5 UML class diagram of your complete model.

Relationships:

- The Car class has a composition relationship with the Engine and Tires classes, meaning it contains objects of these classes as its attributes.
- The Car class utilizes the Engine and Tires classes to execute various functions like starting, running, stopping, and restarting the car.
- While there aren't explicit relationships displayed between the Engine and Tires classes, they are integral to the Car's functionality, working together within the Car class to manage the car's operations and monitor its status.

b. **Screenshots demonstrating that you can start, drive, and stop your Car**.

Below is a Screenshots demonstrating that you can start, drive, and stop your Car after executing the code.

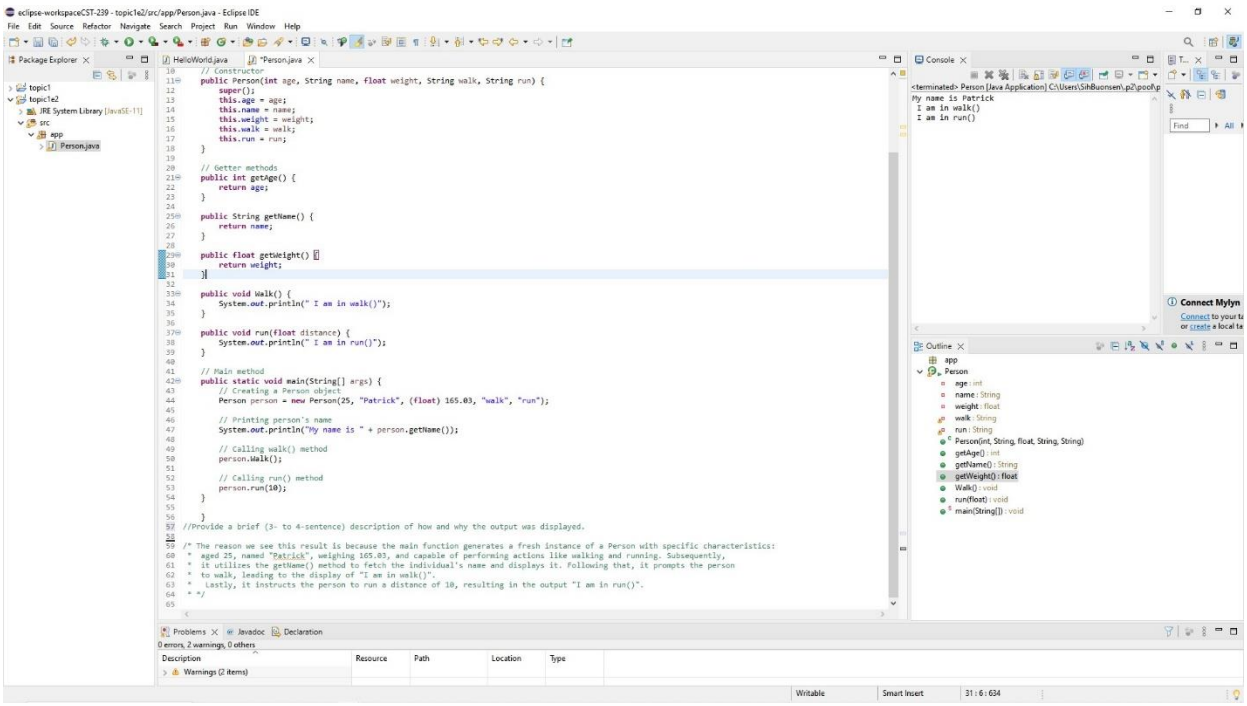

Figure 6 Screenshots demonstrating that you can start, drive, and stop your Car.

Here is what the Car Class does.

This code defines a Car class that has attributes for an engine, tires, and speed, along with methods to start, run, stop, and restart the car. The Car class interacts with the Engine and Tire classes to manage the car's operation and monitor its status.

The DriverScript class sets up a basic program where we create a car, start its engine, make it run, stop it, and then start it again. It's like playing with a virtual car and seeing how it behaves when we give it commands. It has the following:

- Main Method:
    - The main method is like the front door of the program, it's where the program begins.
    - It's like the starting point and it needs to be there for the program to start running.
    - It's written in a specific way, so the computer knows where to begin executing the program.
- Car Object Creation:
    - This part creates a new car, just like how you would make a new car in a game or a simulation.
    - It's like bringing a car to life in the program.
    - We're using a special template called the Car class to make the car, and we're giving it the name "myCar."
- Car Operations:
    - myCar.start(): This is like turning the key to start the car's engine.
    - myCar.run(50): This is like making the car move at a speed of 50 miles per hour. It's like stepping on the gas pedal.
    - myCar.stop(): This is like hitting the brakes to stop the car.
    - myCar.restart(): This is like starting the car's engine again after it's stopped. It's like giving the car a fresh start.

c. Submit a zip file of your source code, including the generated Javadoc files.

Please see attached zip file for the source code and the Javadoc file.

CST-239Activity1_3.zip

Part 4: Using the Debugger

a. Screenshot from the Setting Breakpoints task.

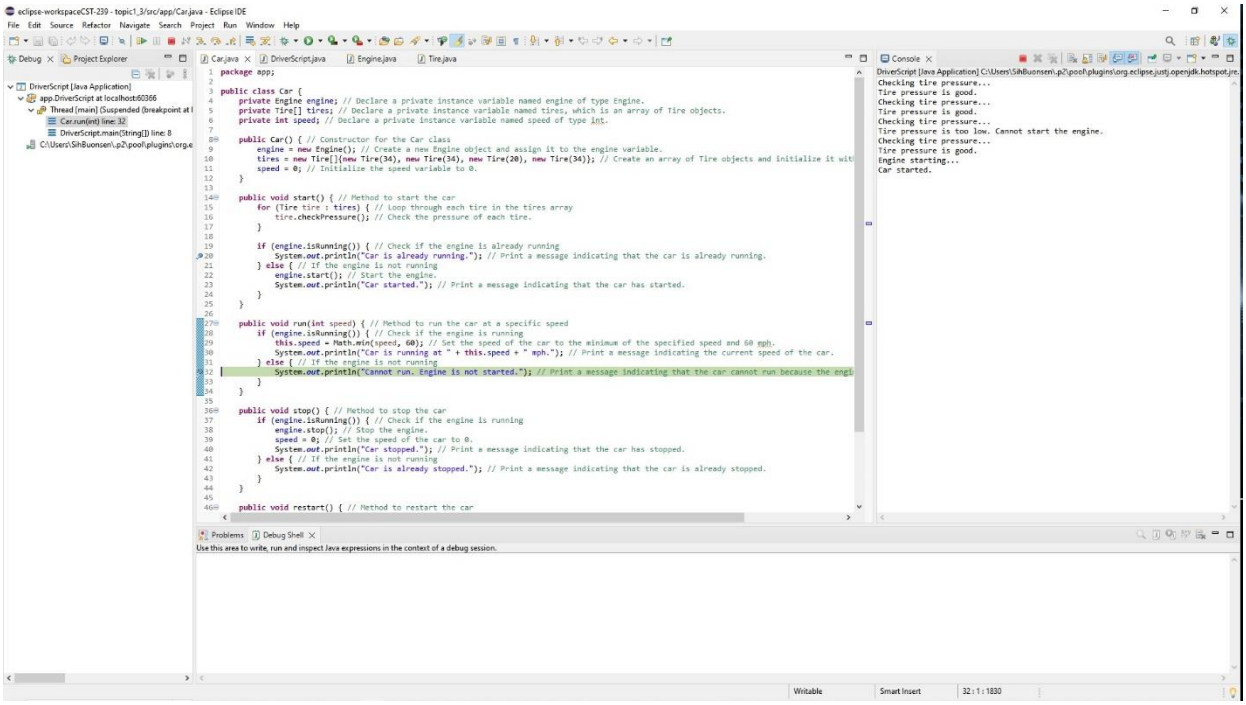

Figure 7. Screenshot from the Setting Breakpoints task

b. Screenshots from the Inspecting Variables task.

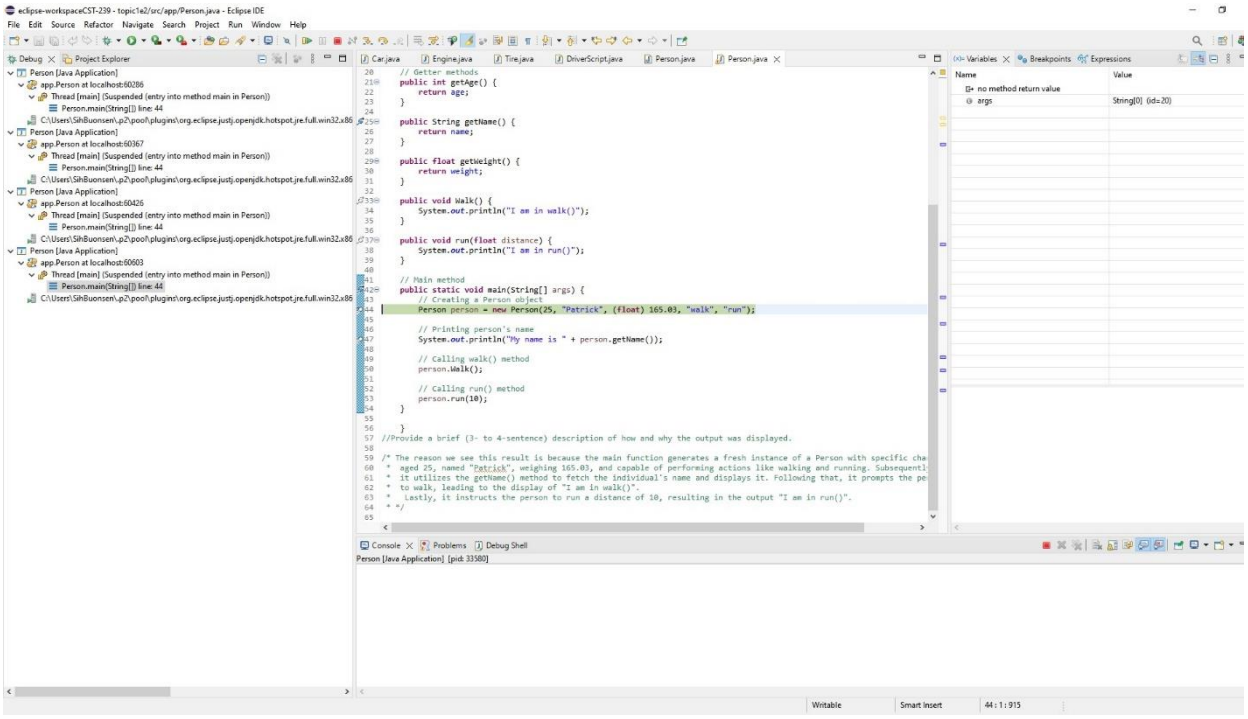

Figure 8. Screenshots from the Inspecting Variables task.

c. Screenshots from the Stepping task.

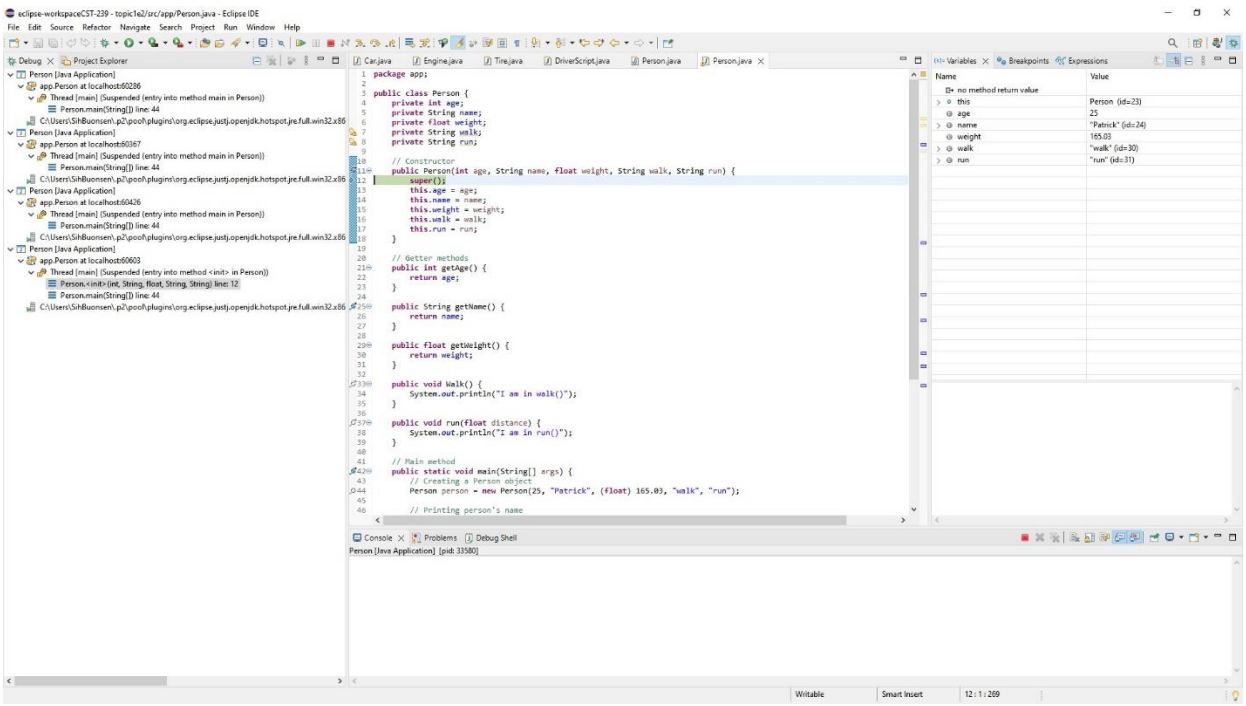

Figure 9. Screenshots from the Stepping task.

d. Screenshot from the Inspecting Call Stack task.

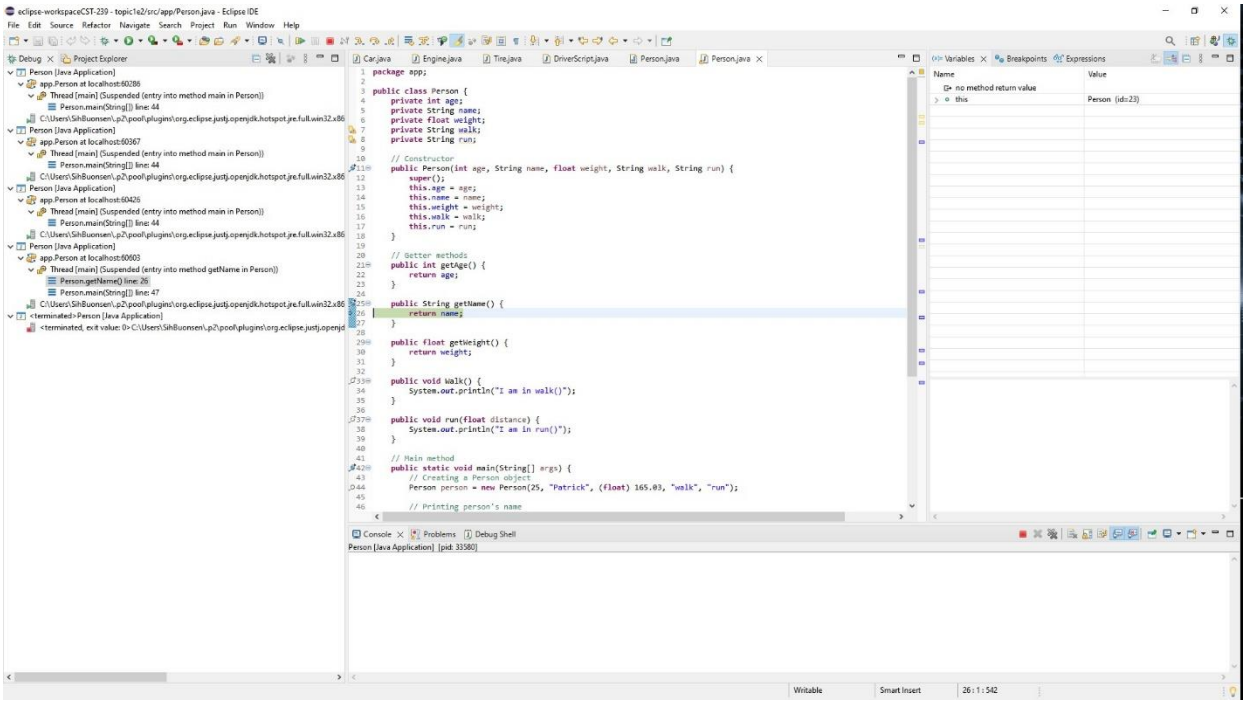

Figure 10. Screenshot from the Inspecting Call Stack task.