/*
THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS OR COPIED FROM ONLINE RESOURCES.
Philip Cardozo
*/

1. Provide a 3-5 sentence general description of your set of code files and the domain you are representing.

This program models an F1 car and a driver system. The Car class represents a Formula 1 car, while the Driver class enforces eligibility rules (age ≥ 16, sober). The Main class simulates different drivers attempting to drive the car, enforcing basic driving laws. The program also integrates polymorphism, encapsulation, and inheritance to structure the simulation effectively.

2. For each file, identify if it is a concrete class, abstract class, or interface. Additionally provide a 1-2 sentence description of the design choice reasoning for making the file concrete vs. abstract or an interface.

Vehicle.java (Abstract Class) - Defines the base structure for vehicles but doesn't implement specific functionality. This allows flexibility for different types of vehicles while enforcing a shared structure.

Car.java (Concrete Class) - Extends Vehicle and implements actual F1 car behavior. It must be concrete because we need to instantiate and simulate an F1 car with real attributes like engine type and starting/stopping behavior.

Drivable.java (Interface) - Ensures all vehicles have start() and stop(). Used as an interface because different vehicle types might have unique implementations but should always be drivable.

DriverRequirements.java (Interface) - Defines canDrive() to enforce driver eligibility rules. It's an interface to allow different types of drivers (even unrealistic) to implement their own versions of eligibility checks.

Driver.java (Concrete Class) - Implements DriverRequirements and handles driver attributes (name, age, sobriety). Concrete because we need to create actual drivers and check their eligibility.

Main.java (Driver Class) - Runs the simulation, creating cars and drivers while demonstrating OOP concepts like polymorphism and encapsulation. It's concrete because it's the entry point of the program.

3. Identify where the inheritance relationships are in your code and provide a 1-2 sentence description of the design choice reasoning for these relationships.

When Car extends Vehicle, it allows the Car class to inherit shared vehicle properties (make, model, year) while adding F1-specific overrides like engineNote(), which simulates the engine sound of an F1 car.

When Vehicle implements Drivable, it must use the methods start() and stop(), enforcing a interface for drivable objects, like a set of rules.

When Driver implements DriverRequirements → Standardizes eligibility checks so that only legal and capable drivers (age ≥ 16, sober) can operate the F1 car, enforcing real-world safety rules.

Driver implements DriverRequirements, it creates the eligibility checks for drivers, ensuring all drivers meet the required conditions (age ≥ 16, sober) before attempting to drive. (I did not consider a license because then only f1 drivers woud be able to drive and not usual people)

4. Identify the variables and/or methods with different encapsulation levels in your code. Provide a 1-2 sentence description of the design choice reasoning for these different encapsulation levels.

- Private variables: chosen to keep sensitive data hidden and prevent possible direct modification. (engineRunning, engineType, doors in Car; name, age, sober in Driver). This ensures that changes to these values only happen through controlled methods.
- Protected variables: Used in Vehicle (make, model, year) so subclasses like Car can inherit them without exposing them to external modification. This maintains flexibility while keeping data secure.
- Public methods: Used for controlled interaction with objects (start(), stop(), getDetails() in Car and Vehicle; canDrive() in Driver). These methods allow safe access to object behavior without exposing internal details.

5. Identify which file is your driver file and briefly describe how it demonstrates your code and the domain you are representing. Give a 1-2 sentence description of where you demonstrate polymorphism in the driver.

Driver File: Main.java initializes a Car and a few Driver objects, testing if each driver meets the eligibility rules before allowing them to drive. It runs the full simulation of an F1 quick hop in hop off scenario by handling car startup, engine sound, and stopping. A Car object is referenced as a Vehicle, allowing start(), stop(), and engineNote() to be called dynamically based on the actual object type. This demonstrates method overriding and ensures consistent behavior across different vehicle types.