Міністерство освіти і науки України
Національний університет «Львівська політехніка»
Кафедра «Електронних обчислювальних машин»

Звіт

з лабораторної роботи № 3

з дисципліни: «Кросплатформенні засоби програмування»

на тему: «Спадкування та інтерфейси»

Виконав:

студент групи КІ-306

Глухенький Д. Ю.

Прийняв:

доцент кафедри

ЕОМ Іванов Ю. С.

Львів – 2023

Мета роботи: ознайомитися з спадкуванням та інтерфейсами у мові Java.

Завдання (варіант № 5)

1. Написати та налагодити програму на мові Java, що розширює клас, що реалізований у лабораторній роботі №3, для реалізації предметної області заданої варіантом. Суперклас, що реалізований у лабораторній роботі №3, зробити абстрактним. Розроблений підклас має забезпечувати механізми свого коректного функціонування та реалізовувати мінімум один інтерфейс. Програма має розміщуватися в пакеті Група.Прізвище.Lab3 та володіти коментарями, які дозволять автоматично згенерувати документацію до розробленого пакету.
2. Автоматично згенерувати документацію до розробленого пакету.
3. Скласти звіт про виконану роботу з приведенням тексту програми, результату її виконання та фрагменту згенерованої документації.
4. Дати відповідь на контрольні запитання

Вихідний код програми

Файл CarDriver.java

```
package main.java.ki306.hlukhenkyi.lab3;

import main.java.ki306.hlukhenkyi.lab3.models.*;

/**
 * The {@code CarDriver} class serves as a driver program to test and demonstrate
 * the functionality of the {@link Car} class, which represents a car in the automotive
domain.
 * It creates instances of the Car class, performs various actions on them, and displays
 * information about the cars and their engines.
 */
public class CarDriver {
    /**
     * The main method is the entry point for the CarDriver program.
     * It creates instances of the Car class, demonstrates their functionality,
     * and displays information about the cars.
     *
     * @param args The command-line arguments (not used in this program).
     */
    public static void main(String[] args) {
        // Create an engine for the first car
        Engine engine1 = new Engine("Gasoline", 2.0);

        // Create a cargo truck
        CargoTruck cargoTruck = new CargoTruck("Ford", 2022, 5000); // Specify cargo
capacity (in kg)

        // Display information about the cargo truck
        cargoTruck.displayInformation();

        // Load cargo into the cargo truck
        cargoTruck.loadCargo(3000); // Load 3000 kg of cargo
```

```
        // Display updated information about the cargo truck
        cargoTruck.displayInformation();

        // Unload cargo from the cargo truck
        cargoTruck.unloadCargo(2000); // Unload 2000 kg of cargo

        // Display updated information about the cargo truck
        cargoTruck.displayInformation();
    }
}
```

Файл Car.java

```
package main.java.ki306.hlukhenkyi.lab3.models;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/**
 * Class Car represents a car entity.
 *
 * This class models a car with a brand, manufacturing year, current speed,
 * and an optional engine. It also provides methods for car operations like starting,
 * stopping, changing the engine, accelerating, decelerating, turning on lights,
 * and displaying car information.
 *
 * @author Hlukehnkyi Dmytro
 * @version 1.0
 * @since 1.0
 */
public abstract class Car implements CargoTransport {
    private final String brand;
    private final int manufacturingYear;
    private int currentSpeed;
    private Engine engine;
    private FileWriter logFileWriter;
    private final String logFileName = "CarDriver.txt";

    /**
     * Constructs a car with the specified brand, manufacturing year, and engine.
     *
     * @param brand            The brand of the car.
     * @param manufacturingYear The manufacturing year of the car.
     * @param engine           The Engine object representing the car's engine.
     */
    public Car(String brand, int manufacturingYear, Engine engine) {
        this.brand = brand;
        this.manufacturingYear = manufacturingYear;
        this.engine = engine;
        initLogFileWriter();
    }

    /**
     * Constructs a car with the specified brand and manufacturing year.
     * The engine is set to null by default.
     *
     * @param brand            The brand of the car.
     * @param manufacturingYear The manufacturing year of the car.
     */
    public Car(String brand, int manufacturingYear) {
        this.brand = brand;
        this.manufacturingYear = manufacturingYear;
        this.engine = null;
        initLogFileWriter();
    }

    // Private method to initialize the log file writer
    private void initLogFileWriter() {
        try {
            File logFile = new File(logFileName);
```

```
            if (!logFile.exists()) {
                logFile.createNewFile();
            }

            logFileWriter = new FileWriter(logFile, true); // Append mode
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Logs a message to the car's log file.
     *
     * @param message The message to be logged.
     */
    public void log(String message) {
        try {
            logFileWriter.write(message + "\n");
            logFileWriter.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Disposes of the log file writer and closes the log file.
     *
     * @throws IOException Thrown in case of input/output errors.
     */
    public void dispose() throws IOException {
        if (logFileWriter != null) {
            logFileWriter.close();
        }
    }

    /**
     * Starts the car if an engine is available and logs the activity.
     */
    public void startCar() {
        if (engine != null) {
            System.out.println("Brand: " + brand + " Car started.");
            log("Brand: " + brand + " Car started.");
        } else {
            System.out.println("Cannot start the car without an engine.");
            log("Cannot start the car without an engine.");
        }
    }

    /**
     * Stops the car and logs the activity.
     */
    public void stopCar() {
        System.out.println("Brand: " + brand + " Car stopped.");
        log("Brand: " + brand + " Car stopped.");
    }

    /**
     * Changes the car's engine and logs the activity.
     *
     * @param newEngine The new Engine object to replace the current engine.
     */
    public void changeEngine(Engine newEngine) {
        engine = newEngine;
        System.out.println("Brand: " + brand + " Engine changed.");
        log("Brand: " + brand + " Engine changed.");
    }

    /**
     * Displays information about the car including brand, manufacturing year,
     * and engine details if available.
     */
    public void displayInformation() {
        System.out.println("Brand: " + brand);
```

```
        log("Brand: " + brand);
        System.out.println("Manufacturing Year: " + manufacturingYear);
        log("Manufacturing Year: " + manufacturingYear);
        if (engine != null) {
            engine.displayInformation();
        } else {
            System.out.println("Engine not available.");
            log("Engine not available.");
        }
    }

    /**
     * Accelerates the car by increasing its speed by a specified amount.
     *
     * @param accelerationAmount The amount by which to accelerate the car in km/h.
     */
    public void accelerate(int accelerationAmount) {
        if (engine != null) {
            int newSpeed = currentSpeed + accelerationAmount;
            System.out.println("Brand: " + brand + " Car accelerated to " + newSpeed + "
km/h.");
            log("Brand: " + brand + " Car accelerated to " + newSpeed + " km/h.");
            currentSpeed = newSpeed;
        } else {
            System.out.println("Cannot accelerate the car without an engine.");
            log("Cannot accelerate the car without an engine.");
        }
    }

    /**
     * Decelerates the car by reducing its speed by a specified amount.
     *
     * @param decelerationAmount The amount by which to decelerate the car in km/h.
     */
    public void decelerate(int decelerationAmount) {
        if (engine != null) {
            int newSpeed = currentSpeed - decelerationAmount;
            if (newSpeed < 0) {
                newSpeed = 0;
            }
            System.out.println("Brand: " + brand + " Car decelerated to " + newSpeed + "
km/h.");
            log("Brand: " + brand + " Car decelerated to " + newSpeed + " km/h.");
            currentSpeed = newSpeed;
        } else {
            System.out.println("Cannot decelerate the car without an engine.");
            log("Cannot decelerate the car without an engine.");
        }
    }

    /**
     * Turns on the lights of the car and logs the activity.
     */
    public void turnLightsOn() {
        if (engine != null) {
            Boolean lightsOn = true;
            System.out.println("Brand: " + brand + " Car lights turned on.");
            log("Brand: " + brand + " Car lights turned on.");
        } else {
            System.out.println("Cannot turn on the lights without an engine.");
            log("Cannot turn on the lights without an engine.");
        }
    }
}
```

Файл CargoTruckjava

```
package main.java.ki306.hlukhenkyi.lab3.models;

/**
 * The {@code CargoTruck} class represents a cargo truck, which is a specific type of car
 * with the ability to load and unload cargo. It extends the {@link Car} class.
 *
 * @author Dmytro Hlukhenkyi
```

```
 * @version 1.0
 * @since [Date]
 */
public class CargoTruck extends Car {
    private int cargoCapacity;

    /**
     * Constructs a cargo truck with the specified brand, manufacturing year, and cargo
capacity.
     *
     * @param brand            The brand of the cargo truck.
     * @param manufacturingYear The manufacturing year of the cargo truck.
     * @param cargoCapacity    The maximum cargo capacity in kilograms that the truck can
carry.
     */
    public CargoTruck(String brand, int manufacturingYear, int cargoCapacity) {
        super(brand, manufacturingYear);
        this.cargoCapacity = cargoCapacity;
    }

    /**
     * Loads cargo into the cargo truck, if there is enough available cargo space.
     *
     * @param weight The weight of the cargo to be loaded in kilograms.
     */
    @Override
    public void loadCargo(int weight) {
        if (weight <= cargoCapacity) {
            cargoCapacity -= weight; // Decrease the available cargo space.
            System.out.println("Loaded " + weight + " kg of cargo.");
        } else {
            System.out.println("Not enough cargo space to load " + weight + " kg.");
        }
    }

    /**
     * Unloads cargo from the cargo truck, if the specified weight can be accommodated.
     *
     * @param weight The weight of the cargo to be unloaded in kilograms.
     */
    @Override
    public void unloadCargo(int weight) {
        if (weight <= cargoCapacity) {
            cargoCapacity += weight; // Increase the available cargo space.
            System.out.println("Unloaded " + weight + " kg of cargo.");
        } else {
            System.out.println("Cannot unload " + weight + " kg, as it exceeds the cargo
capacity.");
        }
    }
}
```

Файл CargoTransport.java

```
package main.java.ki306.hlukhenkyi.lab3.models;

/**
 * The {@code CargoTransport} interface defines the contract for any class that
 * represents a cargo transport vehicle. Classes implementing this interface
 * must provide implementations for loading and unloading cargo.
 *
 * @author Dmytro Hlukhenkyi
 * @version 1.0
 * @since now
 */
public interface CargoTransport {

    /**
     * Loads cargo into the cargo transport vehicle.
     *
     * @param weight The weight of the cargo to be loaded in kilograms.
     */
    void loadCargo(int weight);
```

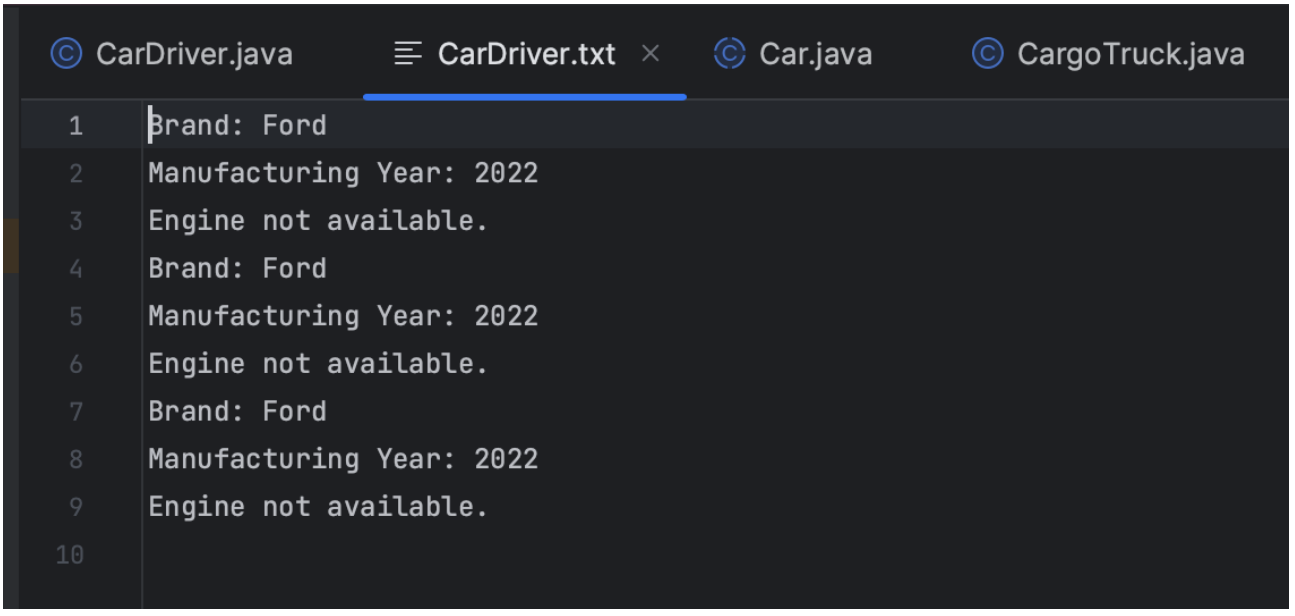
```
    /**
     * Unloads cargo from the cargo transport vehicle.
     *
     * @param weight The weight of the cargo to be unloaded in kilograms.
     */
    void unloadCargo(int weight);
}
```

Файл Engine.java

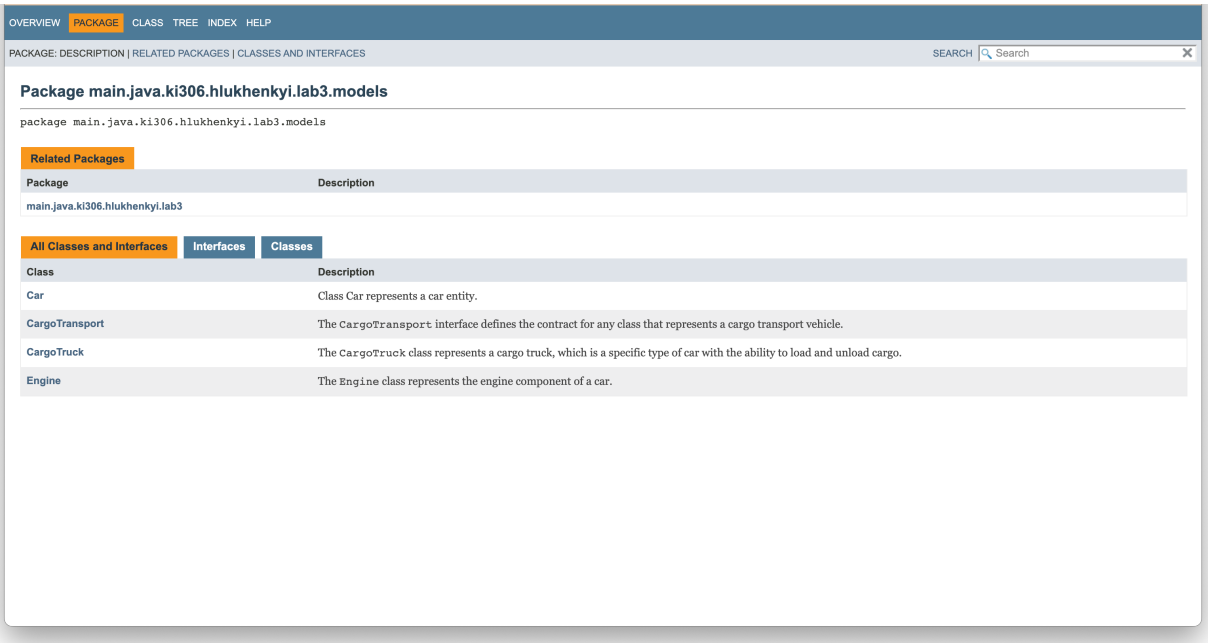
```
package main.java.ki306.hlukhenkyi.lab3.models;

/**
 * The {@code Engine} class represents the engine component of a car.
 * It stores information about the engine type and displacement.
 */
public class Engine {
    private String type;         // The type of the engine (e.g., "Gasoline", "Diesel")
    private final double displacement; // The engine displacement in liters (e.g., 2.0,
2.5)

    /**
     * Constructs an Engine object with the specified type and displacement.
     *
     * @param type         The type of the engine (e.g., "Gasoline", "Diesel").
     * @param displacement The engine displacement in liters (e.g., 2.0, 2.5).
     */
    public Engine(String type, double displacement) {
        this.type = type;
        this.displacement = displacement;
    }

    /**
     * Displays information about the engine, including its type and displacement.
     * Example output:
     * Engine Type: Gasoline
     * Displacement: 2.0 L
     */
    public void displayInformation() {
        System.out.println("Engine Type: " + type);
        System.out.println("Displacement: " + displacement + " L");
    }
}
```

Результат виконання програми

CarDriver.txt:



Фрагмент згенерованої документації



Відповіді на контрольні запитання

1. Синтаксис реалізації спадкування.
   - class МійКлас implements Інтерфейс {
     // тіло класу
     }

2. Що таке суперклас та підклас?
   - суперклас - це клас, від якого інший клас успадковує властивості та методи. Підклас - це клас, який успадковує властивості та методи від суперкласу.

3. Як звернутися до членів суперкласу з підкласу?
   - super.назваМетоду([параметри]); // виклик методу суперкласу super.назваПоля; // звернення до поля суперкласу
4. Коли використовується статичне зв’язування при виклику методу?
   - Статичне зв’язування використовується, коли метод є приватним, статичним, фінальним або конструктором. В таких випадках вибір методу відбувається на етапі компіляції.

5. Як відбувається динамічне зв’язування при виклику методу?
   - вибір методу для виклику відбувається під час виконання програми на основі фактичного типу об’єкта.

6. Що таке абстрактний клас та як його реалізувати?
   - це клас, який має один або більше абстрактних методів (методів без реалізації). Щоб створити абстрактний клас, використовується ключове слово abstract.
     Приклад:
     abstract class АбстрактнийКлас {
     abstract void абстрактнийМетод();

```
}
```

7. Для чого використовується ключове слово instanceof?
   - для перевірки, чи об'єкт належить до певного класу або інтерфейсу.
     Синтаксис: if (об'єкт
     instanceof Клас) {
     // код, який виконується, якщо об'єкт належить до класу
     }

8. Як перевірити чи клас є підкласом іншого класу?
   - В Java використовується ключове слово extends, щоб вказати, що клас є підкласом іншого класу. Перевірити, чи один клас є підкласом іншого класу можна шляхом аналізу ієрархії успадкування.

9. Що таке інтерфейс?
   - це абстрактний тип даних, який визначає набір методів, але не надає їх реалізацію. Всі методи інтерфейсу є загальнодоступними та автоматично є public. Інтерфейси використовуються для створення контрактів, які класи повинні реалізувати.

10. Як оголосити та застосувати інтерфейс?
    - Для оголошення інтерфейсу використовується ключове слово interface.
      Синтаксис:
      interface Інтерфейс {
      // оголошення методів та констант
      }

    - Для застосування інтерфейсу в класі використовується ключове слово implements.
      Синтаксис:
      class МійКлас implements Інтерфейс {
      // реалізація методів інтерфейсу
      }

Висновок

У ході виконання даної лабораторної роботи, отримав навички роботи з концепціями спадкування та інтерфейсами в мові програмування Java. Ознайомившись з цими важливими аспектами об'єктно-орієнтованого програмування, зрозумів їх роль у створенні більш структурованих і гнучких програм.