

Міністерство освіти і науки України
Національний університет «Львівська політехніка»

Кафедра ЕОМ



Лабораторна робота №3

з дисципліни: «Кросплатформенні засоби програмування»

на тему: «Спадкування та інтерфейси»

Варіант № 25

Виконав:

ст. гр. КІ-305

Федусь Н.В.

Прийняв:

Іванов Ю. С.

Львів – 2023

Мета: Ознайомитися з спадкуванням та інтерфейсами у мові Java.

Завдання:

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.Lab4 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.

Код Lab4FedusNazar.java:

```
package KI305.Fedus.Lab3;
```

```
import java.io.FileNotFoundException;
```

```
/**
```

```
 * Lab4FedusNazar class demonstrates the usage of the ClimateControlDevice class.
```

```
 */
```

```
public class Lab4FedusNazar {
```

```
/**
```

```
 * The main method for the Lab4FedusNazar class.
```

```
 *
```

```
 * @param args Command line arguments (not used in this program).
```

```
 * @throws FileNotFoundException if an error occurs while handling files.
```

```
 */
```

```
public static void main(String[] args) throws FileNotFoundException {
```

```
 // Create a ClimateControlDevice object with default settings
```

```
 ClimateControlDevice climateControlDevice = new ClimateControlDevice();
```

```
 // Display the current humidity, temperature, and factory name
```

```
 System.out.println("Humidity: " + climateControlDevice.checkHumidity());
```

```
 System.out.println("Temperature: " + climateControlDevice.checkTemperature());
```

```
 System.out.println("Factory Name: " + climateControlDevice.getFactoryName());
```

```

climateControlDevice.closeLoggerFile();

// Create another ClimateControlDevice object with custom settings
ClimateControlDevice climateControlDevice2 = new ClimateControlDevice("Conditioner 3005",
Conditioner.ConditionColor.GREEN);

// Display the temperature, humidity, and factory name for the second device
System.out.println("Temperature (Device 2): " + climateControlDevice2.checkTemperature());
System.out.println("Humidity (Device 2): " + climateControlDevice2.checkHumidity());
System.out.println("Factory Name (Device 2): " + climateControlDevice2.getFactoryName());

// Close the logger files associated with both devices
climateControlDevice2.closeLoggerFile();
}
}

```

Код ClimateControlDevice.java:

```

package KI305.Fedus.Lab3;

import java.io.FileNotFoundException;

/**
 * The TemperatureControlDeviceInterface provides an interface for devices
 * capable of checking and controlling temperature.
 */
interface TemperatureControlDeviceInterface {
/**
 * Checks the temperature level.
 *
 * @return the current temperature level.
 */
double checkTemperature();
}

/**
 * The HumidityControlDeviceInterface provides an interface for devices
 * capable of checking and controlling humidity.
 */
interface HumidityControlDeviceInterface {
/**
 * Checks the humidity level.

```

```

*
* @return the current humidity level.
*/
double checkHumidity();
}

/**
 * The ClimateControlDevice class represents a climate control device
 * extending the Conditioner class. It implements temperature and humidity
 * control interfaces and introduces a charge field for power management.
 */
public class ClimateControlDevice extends Conditioner
implements TemperatureControlDeviceInterface, HumidityControlDeviceInterface {
private double charge; // Reusing the charge field from Conditioner
private static int controllerNumber = 1;

/**
 * Default constructor for the ClimateControlDevice class.
 * Initializes the climate control device with a generated factory name
 * and a full charge. Increments the controller number.
 *
 * @throws FileNotFoundException if the log file cannot be created.
 */
public ClimateControlDevice() throws FileNotFoundException {
super(String.format("#%s ClimateControlDevice", controllerNumber));
this.charge = 100.0;
++controllerNumber;
}

/**
 * Constructor for the ClimateControlDevice class with a custom factory name.
 * Initializes the climate control device with the specified factory name
 * and a full charge. Increments the controller number.
 *
 * @param factoryName the custom factory name for the climate control device.
 * @throws FileNotFoundException if the log file cannot be created.
 */
public ClimateControlDevice(String factoryName) throws FileNotFoundException {
super(factoryName);
this.charge = 100.0;
++controllerNumber;
}

```

```

/**
 * Constructor for the ClimateControlDevice class with a custom factory name and
 * condition color.
 * Initializes the climate control device with the specified factory name,
 * condition color,
 * and a full charge. Increments the controller number.
 *
 * @param factoryName the custom factory name for the climate control device.
 * @param conditionColor the color of the climate control device.
 * @throws FileNotFoundException if the log file cannot be created.
 */
public ClimateControlDevice(String factoryName, Conditioner.ConditionColor conditionColor)
throws FileNotFoundException {
    super(factoryName, conditionColor);
    this.charge = 100.0;
    ++controllerNumber;
}

/**
 * Checks the humidity level, simulating a decrease in charge.
 *
 * @return a random humidity level within the range [-50, 50].
 */
@Override
public double checkHumidity() {
    if (charge - 0.2 < 0) {
        System.out.println("Device is discharged.");
        return 0;
    }
    charge -= 0.2;

    if (outputStream == null) {
        try {
            openOutputStream();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    double humidityValue = (Math.random() * (50 + 50)) - 50;
    outputStream.println("checkHumidity: " + humidityValue);
    return humidityValue;
}

```

```

}

/**
 * Checks the temperature level, simulating a decrease in charge.
 *
 * @return a random temperature level within the range [0, 100].
 */
@Override
public double checkTemperature() {
    if (charge - 0.2 < 0) {
        System.out.println("Device is discharged.");
        return 0;
    }
    charge -= 0.2;
    if (outputStream == null) {
        try {
            openOutputStream();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    double temperatureValue = Math.random() * 100;
    outputStream.println("checkTemperature: " + temperatureValue);
    return temperatureValue;
}

/**
 * Charges the climate control device by a small amount if not fully charged.
 */
public void toCharge() {
    if (charge == 100.0)
        System.out.println("Is already charged.");
    else
        charge += 0.1;
}
}

```

Conditioner.java

```
package KI305.Fedus.Lab3;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

/**
 * The abstract class Conditioner represents a basic air conditioner.
 * It provides functionality for managing the state of the conditioner
 * and logging information to a file.
 */
public abstract class Conditioner {
    private String factoryName; // Factory name of the conditioner
    private ConditionColor conditionColor; // Color of the conditioner
    private ConditionMode conditionMode; // Mode of the conditioner
    private static int conditionerNumber = 1; // Static counter for generating unique conditioner numbers
    protected PrintWriter outputStream; // PrintWriter for logging information to a file

    /**
     * Default constructor for the Conditioner class.
     * Initializes the conditioner with default values and increments the conditioner number.
     */
    public Conditioner() {
        factoryName = String.format("#%s Conditioner", conditionerNumber);
        conditionColor = ConditionColor.WHITE;
        conditionMode = ConditionMode.TURNED_OFF;
        conditionerNumber++;
    }

    /**
     * Constructor for the Conditioner class with a custom factory name.
     * Initializes the conditioner with the specified factory name and default values.
     * @param factoryName the custom factory name for the conditioner.
     */
    public Conditioner(String factoryName) {
        this.factoryName = factoryName;
        conditionColor = ConditionColor.WHITE;
        conditionMode = ConditionMode.TURNED_OFF;
        conditionerNumber++;
    }
}
```

```

/**
 * Constructor for the Conditioner class with a custom factory name and condition color.
 * Initializes the conditioner with the specified factory name, condition color, and default mode.
 * @param factoryName the custom factory name for the conditioner.
 * @param conditionColor the color of the conditioner.
 */
public Conditioner(String factoryName, ConditionColor conditionColor) {
    this.factoryName = factoryName;
    this.conditionColor = conditionColor;
    conditionMode = ConditionMode.TURNED_OFF;
    conditionerNumber++;
}

/**
 * Opens the output stream for logging if not already opened.
 * @throws FileNotFoundException if the log file cannot be created.
 */
protected void openOutputStream() throws FileNotFoundException {
    outputStream = new PrintWriter(new File(String.format("ConditionerLogger%s.txt",
        conditionerNumber)));
    outputStream.println("Creating a conditioner");
}

/**
 * Get the factory name of the conditioner.
 * @return the factory name of the conditioner.
 */
public String getFactoryName() {
    if (outputStream == null) {
        try {
            openOutputStream();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    outputStream.println("getFactoryName: " + factoryName);
    return factoryName;
}

/**
 * Get the current mode of the conditioner.

```



```

* @return the current mode of the conditioner.
*/
public ConditionMode getConditionMode() {
    if (outputStream == null) {
        try {
            openOutputStream();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    outputStream.println("getConditionMode: " + conditionMode);
    return conditionMode;
}

/**
 * Get the color of the conditioner.
 * @return the color of the conditioner.
 */
public ConditionColor getConditionColor() {
    if (outputStream == null) {
        try {
            openOutputStream();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    outputStream.println("getConditionColor: " + conditionColor);
    return conditionColor;
}

/**
 * Closes the logger file associated with the conditioner.
 */
public void closeLoggerFile() {
    if (outputStream != null) {
        outputStream.println("Close logger file.");
        outputStream.close();
    }
}

/**
 * Turn off the conditioner if it is not already turned off.

```

```

*/
public void turnOffCondition() {
    if (conditionMode == ConditionMode.TURNED_OFF) {
        System.out.println("Condition is already turned off.");
        outputStream.println("Condition is already turned off.");
    } else {
        System.out.println("Condition is turned off.");
        outputStream.println("Condition is turned off.");
        conditionMode = ConditionMode.TURNED_OFF;
    }
}

/**
 * Set the conditioner to low mode if it is not already in low mode.
 */
public void setLowCondition() {
    if (conditionMode == ConditionMode.LOW) {
        System.out.println("Condition is already in low mode.");
        outputStream.println("Condition is already in low mode.");
    } else {
        System.out.println("Condition is set in low mode.");
        outputStream.println("Condition is set in low mode.");
        conditionMode = ConditionMode.LOW;
    }
}

/**
 * Set the conditioner to medium mode if it is not already in medium mode.
 */
public void setMediumCondition() {
    if (conditionMode == ConditionMode.MEDIUM) {
        System.out.println("Condition is already in medium mode.");
        outputStream.println("Condition is already in medium mode.");
    } else {
        System.out.println("Condition is set in medium mode.");
        outputStream.println("Condition is set in medium mode.");
        conditionMode = ConditionMode.MEDIUM;
    }
}

/**
 * Set the conditioner to high mode if it is not already in high mode.

```

```

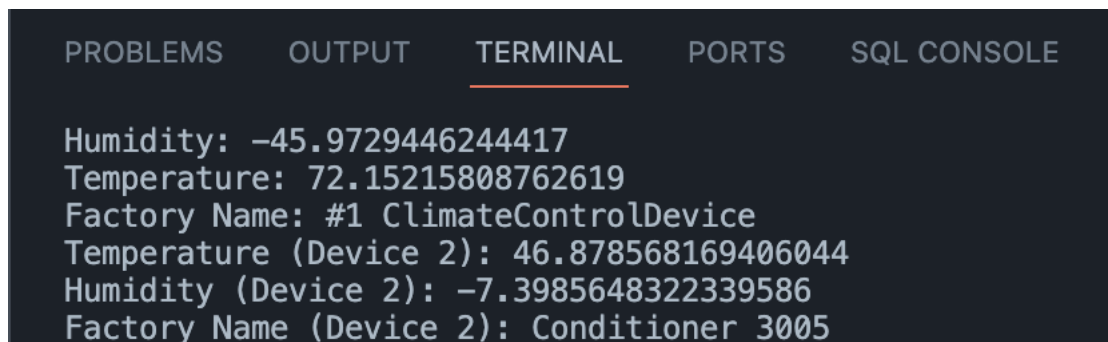
*/
public void setHighCondition() {
    if (conditionMode == ConditionMode.HIGH) {
        System.out.println("Condition is already in high mode.");
        outputStream.println("Condition is already in high mode.");
    } else {
        System.out.println("Condition is set in high mode.");
        outputStream.println("Condition is set in high mode.");
        conditionMode = ConditionMode.HIGH;
    }
}

/**
 * Enum representing different modes of the conditioner.
 */
private enum ConditionMode {
    TURNED_OFF, LOW, MEDIUM, HIGH
}

/**
 * Enum representing different colors of the conditioner.
 */
public enum ConditionColor {
    WHITE, BLACK, RED, PINK, YELLOW, GREEN, BLUE
}
}

```

Скріншоти програми:



The screenshot shows an IDE interface with a dark theme. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'TERMINAL' (which is active and underlined), 'PORTS', and 'SQL CONSOLE'. The terminal window displays the following output:

```

Humidity: -45.9729446244417
Temperature: 72.15215808762619
Factory Name: #1 ClimateControlDevice
Temperature (Device 2): 46.878568169406044
Humidity (Device 2): -7.3985648322339586
Factory Name (Device 2): Conditioner 3005

```

Рис. 1. Результат роботи програми.

Контрольні питання:

1. Синтаксис реалізації спадкування.

Відповідь:

```
class Підклас extends Суперклас
{
    Додаткові поля і методи
}
```

2. Що таке суперклас та підклас?

Відповідь: Суперклас – батьківський клас. Підклас – дочірній.

3. Як звернутися до членів суперкласу з підкласу?

Відповідь: `super.назваМетоду([параметри]);` `super.назваПоля;`

4. Коли використовується статичне зв'язування при виклику методу?

Відповідь: метод є приватним, статичним, фінальним або конструктором. Механізм статичного зв'язування передбачає визначення методу, який необхідно викликати, на етапі компіляції.

5. Як відбувається динамічне зв'язування при виклику методу?

Відповідь: метод, що необхідно викликати, визначається по фактичному типу неявного параметру.

6. Що таке абстрактний клас та як його реалізувати?

Відповідь: Це клас який оголошений з ключовим словом `abstract`. Об'єкт такого класу не може бути створеним, може вміщати абстрактні методи.

7. Для чого використовується ключове слово `instanceof`?

Відповідь: Для встановлення чи є певний клас спадкоємцем другого.

8. Як перевірити чи клас є підкласом іншого класу?

Відповідь: використати ключове слово `instanceof`.

9. Що таке інтерфейс?

Відповідь: Інтерфейси вказують що повинен робити клас не вказуючи як саме він це повинен робити. Інтерфейси покликані компенсувати відсутність

множинного спадкування у мові Java та гарантують визначення у класах оголошених у собі прототипів методів.

10. Як оголосити та застосувати інтерфейс?

Відповідь: [public] interface НазваІнтерфейсу

{

Прототипи методів та оголошення констант інтерфейсу

}

Застосувати можна імплементуючи його, або створюючи посилання на дочірній об'єкт класу.

Висновок:

Виконавши лабораторну роботу, я ознайомився з спадкуванням та інтерфейсами у мові Java.