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 KI-304 Мох М. П.

**Прийняв:**

Олексів М. В.

Львів – 2023

**Мета роботи:** 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 №2, для реалізації предметної області заданої варіантом. Суперклас, що реалізований у лабораторній роботі №2, 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

**Силка на GitHub:** https://github.com/MarianMoh/LabsMohKI-304

# Варіант № 12:

# Базовий клас – Водойма

# Похідний клас – Море

**Вихідний код програми:**

Файл Lab3, що демонструє роботу програми:

package KI304.Moh.Lab3;  
  
*/\*\*  
 \* The {@code Lab3} class serves as a demonstration of the functionality of the {@code Sea} class.  
 \* It creates a {@code Sea} object, performs various operations on it, and displays the results.  
 \*  
 \* @author Marian Moh  
 \* @version 1.0  
 \* @since 2023-11-25  
 \*/*public class Lab3 {  
  
 */\*\*  
 \* The main method serves as the entry point of the program.  
 \* It creates a {@code Sea} object, performs various operations on it, and displays the results.  
 \*  
 \* @param args The command line arguments (not used in this example).  
 \*/* public static void main(String[] args) {  
 // Create object sea  
 Sea sea = new Sea("Black Sea ", 1000000, 15, 2500);

// Add season  
 sea.addSeason();  
 // Next Season and show info  
 sea.nextSeason();  
  
 // Shark fishing, that reduces the number of sharks  
 sea.fish();  
 // Shark fishing is not allowed, because too few sharks  
 sea.fish();  
 // Feed our sharks and after this shark fishing is not allowed  
 sea.feed();  
  
 // Show the sea's fields  
 sea.showInfo();  
  
 // Dry up our sea  
 sea.dry();  
 }  
}

# Файл інтерфейсу Fishing:

package KI304.Moh.Lab3;  
  
*/\*\*  
 \* The {@code Fishing} interface represents the actions related to fishing.  
 \* Implementing classes should provide concrete implementations for the fishing and feeding methods,  
 \* and a method to check whether fishing is allowed.  
 \*/*public interface Fishing {  
  
 */\*\*  
 \* Performs the action of fishing.  
 \* Implementations should define the specific behavior of fishing.  
 \*/* void fish();  
  
 */\*\*  
 \* Performs the action of feeding.  
 \* Implementations should define the specific behavior of feeding.  
 \*/* void feed();  
  
 */\*\*  
 \* Checks whether fishing is currently allowed.  
 \*/* void isFishingAllowed();  
}

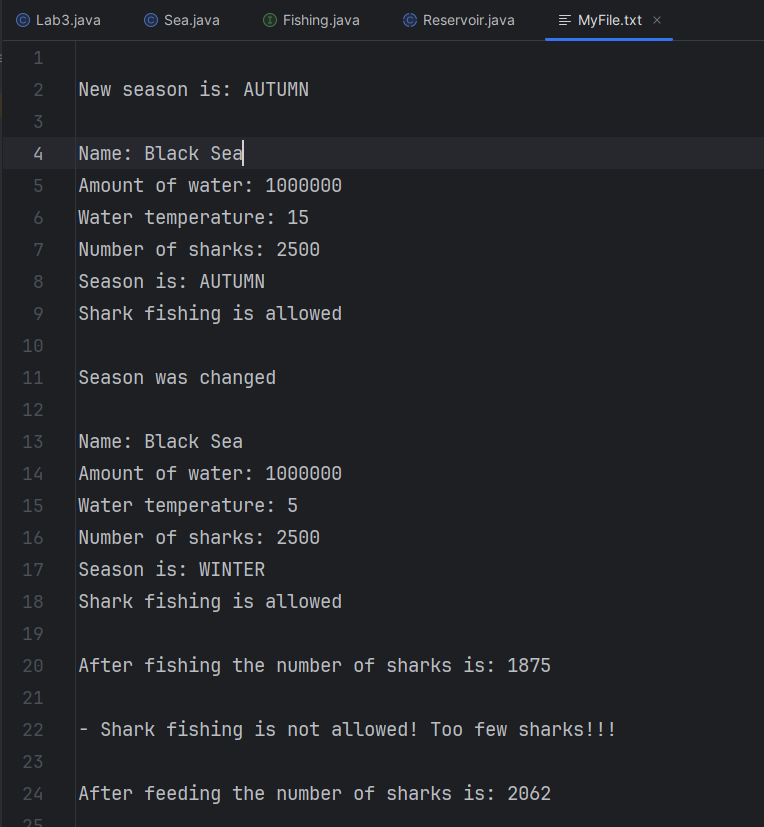
# Файл батьківського абстрактного класу Reservoir:

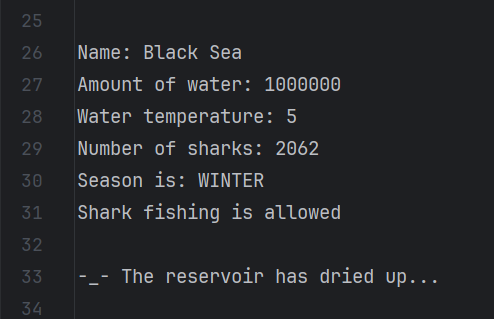
package KI304.Moh.Lab3;;  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.Scanner;  
  
*/\*\*  
 \* The {@code Reservoir} class represents a reservoir with various properties  
 \* such as name, amount of water, water temperature, and current season.  
 \* It provides methods to perform operations on the reservoir.  
 \*  
 \* @author Marian Moh  
 \* @version 1.0  
 \* @since 2023-11-25  
 \*/*public abstract class Reservoir {  
  
 */\*\* The name of the reservoir. \*/* private String name;  
  
 */\*\* The current season of the reservoir. \*/* private Season season;  
  
 */\*\* The amount of water in the reservoir. \*/* private int amountOfWater;  
  
 */\*\* The temperature of the water in the reservoir. \*/* private int waterTemperature;  
  
 */\*\* Flag indicating whether the water in the reservoir is frozen. \*/* private boolean waterIsFrozen;  
  
 */\*\* The FileWriter used to log information to a file. \*/* private FileWriter fileWriter;  
  
 */\*\* The constant file name for logging. \*/* private static final String *FILE\_NAME* = "MyFile.txt";  
  
 */\*\*  
 \* Default constructor for the {@code Reservoir} class.  
 \* Initializes the reservoir and creates a log file.  
 \*/* public Reservoir() {  
 createFile();  
 season = Season.*UNKNOWN*;  
 waterIsFrozen = waterTemperature < 0;  
 }  
  
 */\*\*  
 \* Parameterized constructor for the {@code Reservoir} class.  
 \* Initializes the reservoir with specified properties and creates a log file.  
 \*  
 \* @param name The name of the reservoir.  
 \* @param amountOfWater The amount of water in the reservoir.  
 \* @param waterTemperature The temperature of the water in the reservoir.  
 \*/* public Reservoir(String name, int amountOfWater, int waterTemperature) {  
 this();  
 this.name = name;  
 this.amountOfWater = amountOfWater;  
 this.waterTemperature = waterTemperature;  
 }  
  
 */\*\*  
 \* Gets the name of the reservoir.  
 \*  
 \* @return The name of the reservoir.  
 \*/* public String getName() {  
 return name;  
 }  
  
 */\*\*  
 \* Gets the current season of the reservoir.  
 \*  
 \* @return The current season of the reservoir.  
 \*/* public Season getSeason() {  
 return season;  
 }  
  
 */\*\*  
 \* Sets the season of the reservoir.  
 \*  
 \* @param season The new season to set.  
 \*/* public void setSeason(Season season) {  
 this.season = season;  
 }  
  
 */\*\*  
 \* Gets the amount of water in the reservoir.  
 \*  
 \* @return The amount of water in the reservoir.  
 \*/* public int getAmountOfWater() {  
 return amountOfWater;  
 }  
  
 */\*\*  
 \* Sets the amount of water in the reservoir.  
 \*  
 \* @param amountOfWater The new amount of water to set.  
 \*/* public void setAmountOfWater(int amountOfWater) {  
 this.amountOfWater = amountOfWater;  
 }  
  
 */\*\*  
 \* Gets the temperature of the water in the reservoir.  
 \*  
 \* @return The temperature of the water in the reservoir.  
 \*/* public int getWaterTemperature() {  
 return waterTemperature;  
 }  
  
 */\*\*  
 \* Sets the temperature of the water in the reservoir.  
 \*  
 \* @param waterTemperature The new temperature to set.  
 \*/* public void setWaterTemperature(int waterTemperature) {  
 this.waterTemperature = waterTemperature;  
 }  
  
 */\*\*  
 \* Creates a log file for the reservoir.  
 \* If the file does not exist, it will be created.  
 \*/* private void createFile() {  
 File file = new File(*FILE\_NAME*);  
 try {  
 if (!file.exists() && file.createNewFile()) {  
 System.*out*.println("File created: " + *FILE\_NAME* + "\n");  
 }  
 fileWriter = new FileWriter(file);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Logs a message to the file associated with the reservoir.  
 \*  
 \* @param message The message to be logged.  
 \*/* public void log(String message) {  
 try {  
 fileWriter.write(message + "\n");  
 fileWriter.flush();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Closes the file associated with the reservoir.  
 \* If the file is not null, it will be closed.  
 \*/* public void closeFile() {  
 try {  
 if (fileWriter != null) {  
 fileWriter.close();  
 System.*out*.println("\nFile closed.");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Displays information about the reservoir  
 \*/* public abstract void showInfo();  
  
 */\*\*  
 \* Renames the reservoir with a new name, if the provided name is valid.  
 \*  
 \* @param newName The new name for the reservoir.  
 \*/* public void rename(String newName) {  
 if (newName.matches("^[a-zA-Z ]{3,}$")) {  
 name = newName;  
 log("\nNew name: " + newName);  
 } else {  
 log("\n- Error! New name \"" + newName + "\" is incorrect :(");  
 }  
 }  
  
 */\*\*  
 \* Drains the reservoir, setting its properties to default values,  
 \* logs a message, closes the log file, and exits the program.  
 \*/* public void dry() {  
 name = "Unknown";  
 amountOfWater = 0;  
 waterTemperature = 0;  
  
 log("\n-\_- The reservoir has dried up...");  
 closeFile();  
 System.*exit*(0);  
 }  
  
 */\*\*  
 \* Adds a new season to the reservoir, if the current season is unknown.  
 \* Prompts the user to enter a season and logs the new season.  
 \*/* public void addSeason() {  
 if(seasonIsUnknown()) {  
 Scanner scanner = new Scanner(System.*in*);  
 String strOfSeason = null;  
  
 while (true) {  
 try {  
 System.*out*.print("Enter a season: ");  
 strOfSeason = scanner.nextLine().toUpperCase();  
 season = Season.*valueOf*(strOfSeason);  
  
 log("\nNew season is: " + season);  
 break;  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Invalid season. Please enter a valid season.\n");  
 log("\nError! Invalid season: " + strOfSeason);  
 }  
 }  
 showInfo();  
 } else {  
 log("\n- Error! You already have the season: " + season);  
 }  
 }  
  
 */\*\*  
 \* Deletes the current season of the reservoir and logs the action.  
 \*/* public void deleteSeason() {  
 season = Season.*UNKNOWN*;  
 log("\nSeason was deleted");  
 }  
  
 */\*\*  
 \* Changes the current season of the reservoir based on the current season.  
 \* Logs the change and provides information about the new season.  
 \*/* public void nextSeason() {  
 if(!seasonIsUnknown()) {  
 log("\nSeason was changed");  
 switch (season) {  
 case *WINTER* -> springIsComing();  
 case *SPRING* -> summerIsComing();  
 case *SUMMER* -> autumnIsComing();  
 case *AUTUMN* -> winterIsComing();  
 }  
 } else {  
 log("\n- Error! You don't have a season to change it");  
 }  
 }  
  
 */\*\*  
 \* Checks if the current season of the reservoir is unknown.  
 \*  
 \* @return {@code true} if the season is unknown, {@code false} otherwise.  
 \*/* public boolean seasonIsUnknown() {  
 return season.equals(Season.*UNKNOWN*);  
 }  
  
 */\*\*  
 \* Sets the properties of the reservoir for the winter.  
 \*/* public abstract void winterIsComing();  
  
 */\*\*  
 \* Sets the properties of the reservoir for the spring.  
 \*/* public abstract void springIsComing();  
  
 */\*\*  
 \* Sets the properties of the reservoir for the summer.  
 \*/* public abstract void summerIsComing();  
  
 */\*\*  
 \* Sets the properties of the reservoir for the autumn.  
 \*/* public abstract void autumnIsComing();  
}

# Файл дочірнього класу Sea:

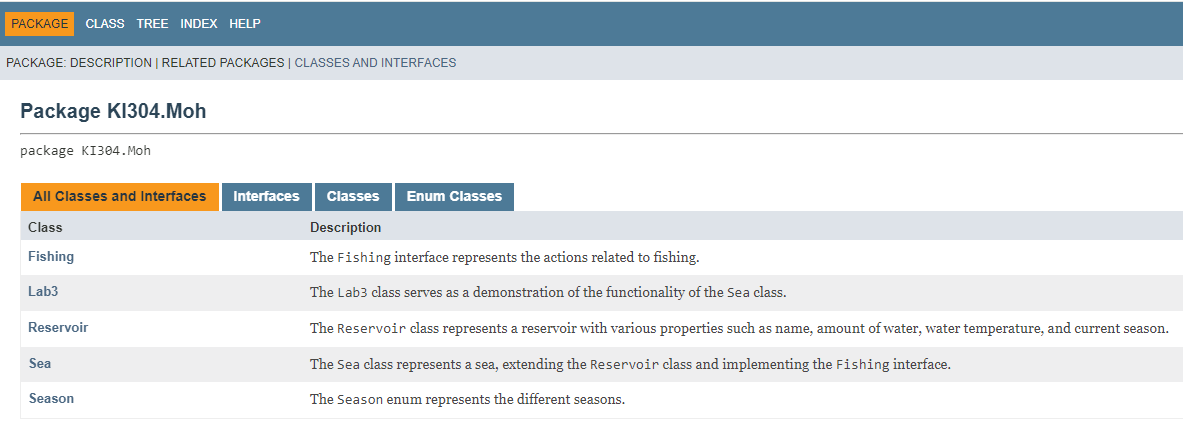
package KI304.Moh.Lab3;;  
  
*/\*\*  
 \* The {@code Sea} class represents a sea, extending the {@code Reservoir} class  
 \* and implementing the {@code Fishing} interface. It includes additional properties  
 \* specific to a sea, such as the number of sharks and fishing allowances.  
 \*  
 \* @author Marian Moh  
 \* @version 1.0  
 \* @since 2023-11-25  
 \*/*public class Sea extends Reservoir implements Fishing {  
  
 */\*\* The number of sharks in the sea. \*/* public int numberOfSharks;  
  
 */\*\* Flag indicating whether shark fishing is allowed in the sea. \*/* public boolean sharkFishingIsAllowed;  
  
 */\*\*  
 \* Default constructor for the {@code Sea} class.  
 \*/* public Sea() {  
 }  
  
 */\*\*  
 \* Parameterized constructor for the {@code Sea} class.  
 \* Initializes the sea with specified properties and checks fishing allowances.  
 \*  
 \* @param name The name of the sea.  
 \* @param amountOfWater The amount of water in the sea.  
 \* @param waterTemperature The temperature of the water in the sea.  
 \* @param numberOfSharks The initial number of sharks in the sea.  
 \*/* public Sea(String name, int amountOfWater, int waterTemperature, int numberOfSharks) {  
 super(name, amountOfWater, waterTemperature);  
 this.numberOfSharks = numberOfSharks;  
 isFishingAllowed();  
 }  
  
 */\*\*  
 \* Displays information about the sea, including its name, amount of water,  
 \* water temperature, number of sharks, current season, and fishing allowances.  
 \*/* @Override  
 public void showInfo() {  
 log("\nName: " + super.getName());  
 log("Amount of water: " + super.getAmountOfWater());  
 log("Water temperature: " + super.getWaterTemperature());  
 log("Number of sharks: " + numberOfSharks);  
  
 if(!seasonIsUnknown()) {  
 log("Season is: " + super.getSeason());  
 }  
  
 if(!sharkFishingIsAllowed) {  
 log("- Shark fishing is not allowed!!!");  
 } else {  
 log("Shark fishing is allowed");  
 }  
 }  
  
 */\*\*  
 \* Sets the properties of the sea for the winter.  
 \* Logs the change and displays information about the sea.  
 \*/* @Override  
 public void winterIsComing() {  
 setSeason(Season.*WINTER*);  
 setWaterTemperature(5);  
 showInfo();  
 }  
  
 */\*\*  
 \* Sets the properties of the sea for the spring.  
 \* Logs the change, adjusts the amount of water, and displays information about the sea.  
 \*/* @Override  
 public void springIsComing() {  
 setSeason(Season.*SPRING*);  
 setWaterTemperature(14);  
 setAmountOfWater((int) (getAmountOfWater() \* 1.2));  
 showInfo();  
 }  
  
 */\*\*  
 \* Sets the properties of the sea for the summer.  
 \* Logs the change, adjusts the amount of water, and displays information about the sea.  
 \* If the amount of water is less than 500000, the sea dries up.  
 \*/* @Override  
 public void summerIsComing() {  
 setSeason(Season.*SUMMER*);  
 setWaterTemperature(19);  
 setAmountOfWater((int) (getAmountOfWater() \* 0.8));  
 if(getAmountOfWater() < 500000) {  
 dry();  
 }  
 showInfo();  
 }  
  
 */\*\*  
 \* Sets the properties of the sea for the autumn.  
 \* Logs the change and displays information about the sea.  
 \*/* @Override  
 public void autumnIsComing() {  
 setSeason(Season.*AUTUMN*);  
 setWaterTemperature(16);  
 showInfo();  
 }  
  
 */\*\*  
 \* Performs the action of fishing in the sea.  
 \* If shark fishing is allowed, reduces the number of sharks,  
 \* checks if further fishing is allowed, and logs the result.  
 \*/* @Override  
 public void fish() {  
 if(sharkFishingIsAllowed) {  
 numberOfSharks = (int) (numberOfSharks \* 0.75);  
 isFishingAllowed();  
 log("\nAfter fishing the number of sharks is: " + numberOfSharks);  
 } else {  
 log("\n- Shark fishing is not allowed! Too few sharks!!!");  
 }  
 }  
  
 */\*\*  
 \* Performs the action of feeding in the sea.  
 \* Increases the number of sharks, checks if fishing is allowed,  
 \* and logs the result.  
 \*/* @Override  
 public void feed() {  
 numberOfSharks = (int) (numberOfSharks \* 1.1);  
 isFishingAllowed();  
 log("\nAfter feeding the number of sharks is: " + numberOfSharks);  
 }  
  
 */\*\*  
 \* Checks whether fishing is currently allowed in the sea.  
 \* Updates the fishing allowance flag based on the number of sharks.  
 \*/* @Override  
 public void isFishingAllowed() {  
 sharkFishingIsAllowed = numberOfSharks > 2000;  
 }  
}

**Результат виконання програми:**





**Фрагмент згенерованої документації:**



**Відповіді на контрольні запитання:**

1. Синтаксис реалізації спадкування.
   * class МійКлас implements Інтерфейс {

// тіло класу

}

1. 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.

1. Як звернутися до членів суперкласу з підкласу?
   * super.назваМетоду([параметри]); // виклик методу суперкласу super.назваПоля;

// звернення до поля суперкласу

1. 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.
2. Як відбувається динамічне зв’язування при виклику методу?
   * вибір методу для виклику відбувається під час виконання програми на основі фактичного типу об’єкта.
3. 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();

}

1. 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

}

1. 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.
2. 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.
3. 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.