Міністерство освіти і науки України
Національний університет «Львівська політехніка»

Кафедра ЕОМ

Звіт

до лабораторної роботи № 6

З дисципліни: «Кросплатформенні засоби програмування»

На тему: **«ПАРАМЕТРИЗОВАНЕ ПРОГРАМУВАННЯ»**

Варіант 16

Виконав: ст.гр. КІ-301
Онисько М.М.
Прийняв: доц.
Майдан М.В.

Львів 2023

Мета: оволодіти навиками параметризованого програмування мовою Java.

Завдання:
1. Створити параметризований клас, що реалізує предметну область задану варіантом. Клас має містити мінімум 4 методи опрацювання даних включаючи розміщення та виймання елементів. Парні варіанти реалізують пошук мінімального елементу, непарні – максимального. Написати на мові Java та налагодити програму-драйвер для розробленого класу, яка мстить мінімум 2 різні класи екземпляри яких розмішуються у екземплярі розробленого класу-контейнеру. Програма має розміщуватися в пакеті Група.Прізвище.Lab6 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.

Варіант:
16. Земельна ділянка

Виконання роботи:

Код програми:

Lab6Onysko.java

```java
package KI301.Onysko.Lab6;
import java.io.*;
import java.util.Scanner;

/**
 * @author Mykola Onysko KI-301
 * @version 1.0
*/

public class Lab6Onysko {
	public static void main(String[] args) {
		// Creating an instance of LandPlot for House objects
		LandPlot<House> houseLandPlot = new LandPlot<>();

		// Adding houses to the land plot
		houseLandPlot.addElement(new House("Modern House", 250, 500000));
		houseLandPlot.addElement(new House("Victorian Mansion", 1200, 1500000));
		houseLandPlot.addElement(new House("Cottage", 100, 200000));

		// Finding and printing the house with the minimum value
		House minHouse = houseLandPlot.findMin();
		System.out.println("House with the minimum value:");
		minHouse.print();

		// Printing all houses in descending order of values
		houseLandPlot.printAllDescendingByValue();

		// Creating an instance of LandPlot for Farm objects
		LandPlot<Farm> farmLandPlot = new LandPlot<>();
		farmLandPlot.addElement(new Farm("Wheat Farm", 200, 300000));
		farmLandPlot.addElement(new Farm("Dairy Farm", 150, 250000));
```

```
        farmLandPlot.addElement(new Farm("Vineyard", 80, 400000));

        // Finding and printing the farm with the minimum value
        Farm minFarm = farmLandPlot.findMin();
        System.out.println("Farm with the minimum value:");
        minFarm.print();

        // Finding and printing the farm with the minimum value
        farmLandPlot.printAllDescendingByValue();

        // Finding and printing the farm with the minimum value
        farmLandPlot.searchByDescription("Wheat Farm");

        // Finding and printing the farm with the minimum value
        farmLandPlot.searchByDescription("XXX");
    }
}
```

LandPlot.java

```
package KI301.Onysko.Lab6;
import java.util.ArrayList;
import java.util.Collections;
import java.io.*;

/**
 * * Class <code>LandPlot</code>
 * @author Mykola Onysko KI-301
 * @version 1.0
*/
class LandPlot<T extends Data> {
      private ArrayList<T> items;
    /**
     * Constructor to initialize the land plot with an empty ArrayList.
     */
    public LandPlot() {
        items = new ArrayList<>();
    }
    /**
     * Adds a data element to the land plot.
     *
     * @param element The data element to add to the land plot.
     */
    public void addElement(T element) {
        items.add(element);
        System.out.println("Element added: ");
        element.print();
    }
    /**
     * Removes a data element from the land plot at the specified index.
     *
     * @param index The index of the data element to remove.
     */
    public void removeElement(int index) {
        if (index >= 0 && index < items.size()) {
            items.remove(index);
            System.out.println("Element removed at index " + index);
        } else {
            System.out.println("Invalid index");
        }
    }
    /**
     * Finds and returns the data element with the minimum value in the land plot.
     *
```

```
     * @return The data element with the minimum value, or null if the land plot is
empty.
     */
    public T findMin() {
        if (!items.isEmpty()) {
            T min = items.get(0);
            for (int i = 1; i < items.size(); i++) {
                if (items.get(i).compareTo(min) < 0)
                    min = items.get(i);
            }
            return min;
        }
        return null;
    }
    /**
     * Prints all items in the land plot in descending order of their values.
     */
    public void printAllDescendingByValue() {
        Collections.sort(items, Collections.reverseOrder());
        System.out.println("Contents of the land plot in descending order of
value:");
        for (T element : items) {
            element.print();
        }
    }

    /**
     * Searches for data elements with a specific description in the land plot.
     *
     * @param searchDescription The description to search for.
     */
    public void searchByDescription(String searchDescription) {
      boolean found = false;
        System.out.println("Items with description '" + searchDescription + "':");
        for (T element : items) {
            if (element instanceof House) {
                House house = (House) element;
                if (house.getDescription().equalsIgnoreCase(searchDescription)) {
                    element.print();
                    found = true;
                }
            } else if (element instanceof Farm) {
                Farm farm = (Farm) element;
                if (farm.getDescription().equalsIgnoreCase(searchDescription)) {
                    element.print();
                    found = true;
                }
            }
        }
        if (!found) {
            System.out.println("No items found with description '" +
searchDescription + "'.");
        }
    }

}
/**
 * The `House` class represents a type of data object that represents a house.
 */
class House implements Data {
    private String description;
    private int size;
    private int value;
    /**
```

```java
     * Constructor to initialize a house with the given description, size, and value.
     *
     * @param description The description of the house.
     * @param size        The size of the house.
     * @param value       The value of the house.
     */
    public House(String description, int size, int value) {
        this.description = description;
        this.size = size;
        this.value = value;
    }
    /**
     * Gets the size of the house.
     *
     * @return The size of the house.
     */
    public int getSize() {
        return size;
    }
    /**
     * Sets the size of the house.
     *
     * @param size The new size of the house.
     */
    public void setSize(int size) {
        this.size = size;
    }
    /**
     * Gets the value of the house.
     *
     * @return The value of the house.
     */
    public int getValue() {
        return value;
    }
    /**
     * Gets the description of the house.
     *
     * @return The description of the house.
     */
    public String getDescription() {
        return description;
    }
    /**
     * Compares the house to another data object based on their values.
     *
     * @param p The data object to compare to.
     * @return A negative integer, zero, or a positive integer if this object is less
than,
     * equal to, or greater than the specified object.
     */
    public int compareTo(Data p) {
        Integer s = value;
        return s.compareTo(p.getValue());
    }

    /**
     * Prints information about the house.
     */
    public void print() {
        System.out.println("House description: " + description + ", Size: " + size +
", House value: " + value);
    }
}
```

```java
/**
 * The `Farm` class represents a type of data object that represents a farm.
 */
class Farm implements Data {
    private String description;
    private int acreage;
    private int value;
    /**
     * Constructor to initialize a farm with the given description, acreage, and
value.
     *
     * @param description The description of the farm.
     * @param acreage     The acreage of the farm.
     * @param value       The value of the farm.
     */
    public Farm(String description, int acreage, int value) {
        this.description = description;
        this.acreage = acreage;
        this.value = value;
    }
    /**
     * Gets the acreage of the farm.
     *
     * @return The acreage of the farm.
     */
    public int getAcreage() {
        return acreage;
    }
    /**
     * Sets the acreage of the farm.
     *
     * @param acreage The new acreage of the farm.
     */
    public void setAcreage(int acreage) {
        this.acreage = acreage;
    }
    /**
     * Gets the value of the farm.
     *
     * @return The value of the farm.
     */
    public int getValue() {
        return value;
    }
    /**
     * Gets the description of the farm.
     *
     * @return The description of the farm.
     */
    public String getDescription() {
        return description;
    }
    /**
     * Compares the farm to another data object based on their values.
     *
     * @param p The data object to compare to.
     * @return A negative integer, zero, or a positive integer if this object is less
than,
     * equal to, or greater than the specified object.
     */
    public int compareTo(Data p) {
        Integer s = value;
        return s.compareTo(p.getValue());
    }
```

```java
    /**
     * Prints information about the farm.
     */
    public void print() {
        System.out.println("Farm description: " + description + ", Acreage: " +
acreage + ", Farm value: " + value);
    }
}
/**
 * The `Data` interface represents a common interface for objects that have a
 * value and can be compared based on their values.
 */
interface Data extends Comparable<Data> {
    /**
     * Gets the value of the data object.
     *
     * @return The value of the data object.
     */
    int getValue();
    /**
     * Prints information about the data object.
     */
    void print();
}
```

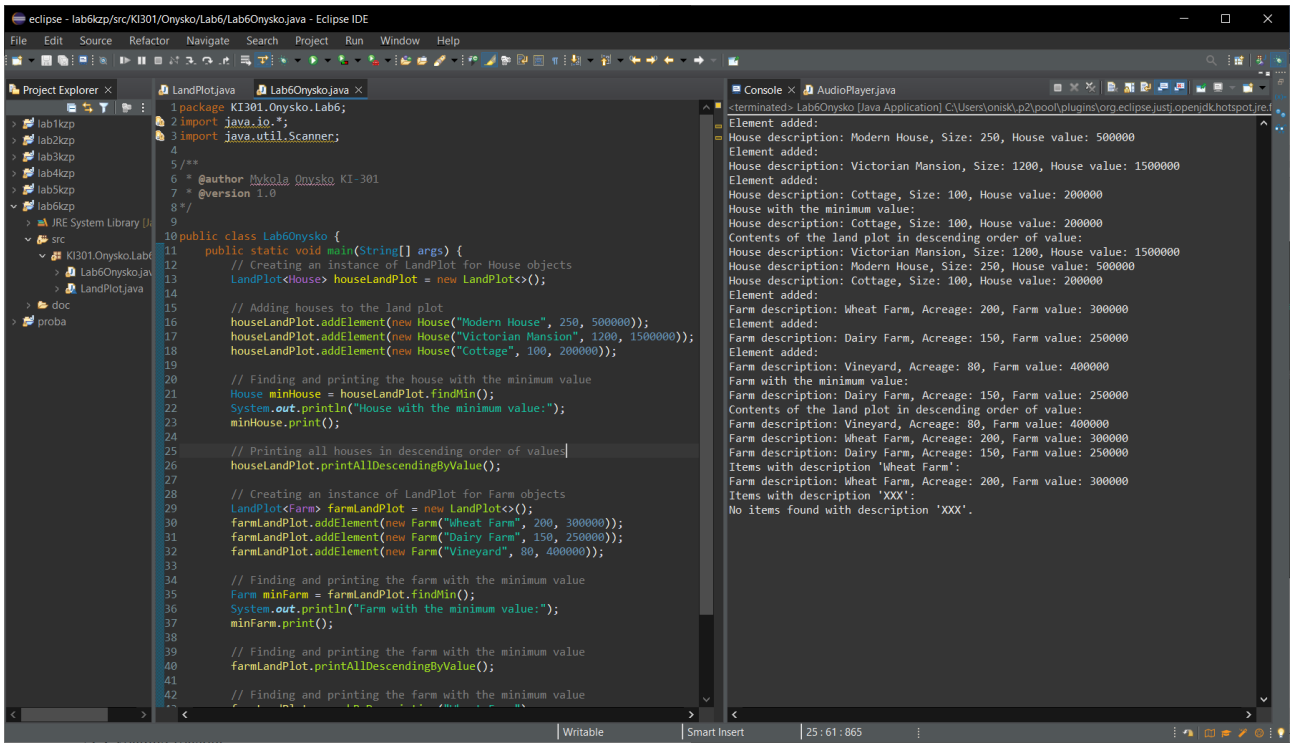

Рис.1 Код та вивід програми

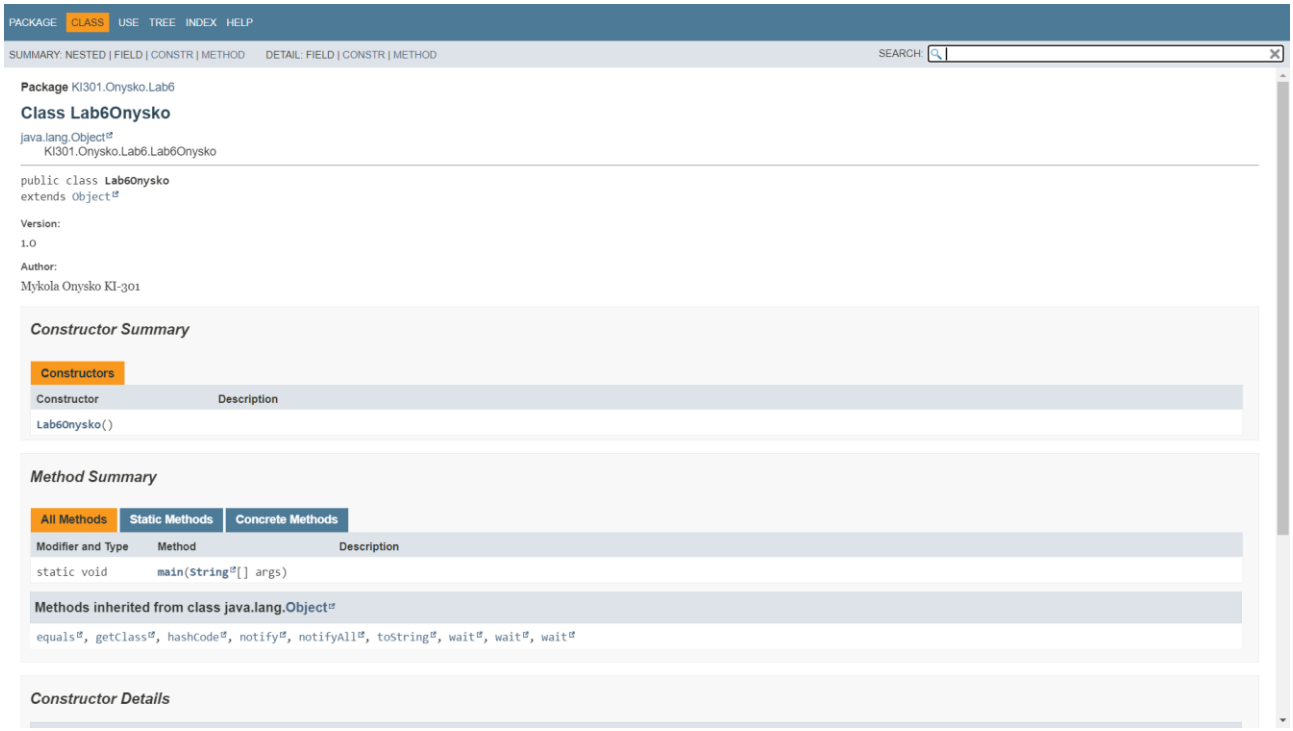

Рис.2 Документація Lab6Onysko

Висновок:

В ході виконання лабораторної роботи я оволодів навичками параметризованого програмування мовою Java та створення автоматично згенерованої документації, яка включає повну інформацію про всі класи, методи та їхні параметри, що полегшує розуміння та використання розроблених компонентів.

Контрольні питання:

1. Дайте визначення терміну «параметризоване програмування».

Параметризоване програмування — це підхід до розробки програмного коду, при якому класи або методи можуть бути написані без прив'язки до конкретного типу даних. Замість цього вони використовують параметри, які визначають типи даних, з якими вони будуть взаємодіяти.

2. Розкрийте синтаксис визначення простого параметризованого класу.

```
public class MyGenericClass<T> {
    // код класу тут
}
```

3. Розкрийте синтаксис створення об’єкту параметризованого класу.

```
MyGenericClass<Integer> myObject = new MyGenericClass<>();
```

4. Розкрийте синтаксис визначення параметризованого методу.

```
public <T> void myGenericMethod(T parameter) {
    // код методу тут
}
```

5. Розкрийте синтаксис виклику параметризованого методу.

```
myGenericMethod("Hello, World!");
```

6. Яку роль відіграє встановлення обмежень для змінних типів?

Встановлення обмежень для змінних типів (generics constraints) дозволяє обмежувати типи даних, які можна використовувати в параметризованих класах чи методах. Це сприяє збільшенню безпеки типів та уникненню помилок.

7. Як встановити обмеження для змінних типів?

Обмеження встановлюються за допомогою ключового слова extends для класів або інтерфейсів. Наприклад:

```
public class MyGenericClass<T extends SomeClass> {
    // код класу тут
}
```

8. Розкрийте правила спадкування параметризованих типів.

Параметризовані типи не спадковують один від одного незалежно від параметрів типу. Наприклад, MyClass<Integer> та MyClass<String> не будуть мати спільного базового параметризованого класу.

9. Яке призначення підстановочних типів?

Підстановочні типи (wildcards) дозволяють зробити код більш гнучким щодо типів даних. Два основних підстановочних типи: ? extends T (вказує, що тип може бути T або його підкласом) та ? super T (вказує, що тип може бути T або його суперкласом).

10. Застосування підстановочних типів.

- Забезпечення гнучкості в коді при роботі з параметризованими типами.
- Зручне використання у методах, де потрібно взаємодіяти з колекціями різних типів.
- Забезпечення безпеки типів у випадках, де конкретний тип не є важливим.