**COMSATS University Islamabad**

**Abbottabad campus**

**Lab-1 Part-2**

**Subject: Design Pattern**

**Class: BSE-7B**

**Name: Irfan Khan**

**Registration No: FA20-BSE-070**

**Submitted to: Mukhtiar Zamin**

**Link to GitHub Course repository:**

<https://github.com/Irfankhan761/Design-Pattern-Lab-Work>

**Link to GitHub Lab1 Part2 repository:**

https://github.com/Irfankhan761/Design-Pattern-Lab-Work/tree/main/LAB1%20Part2

**Assignment Title: Lab1 part2 Exploring Java Advance Concepts**

**Find answer and implementation to the following questions:**

1. **Overloading of the main method is possible or not?**

Overloading of the main method is possible in Java. However, JVM only calls the original main method, it will never call our overloaded main method. This means that if we have an overloaded main method, it will not be executed automatically when we run the program. To execute an overloaded main method, we must call it from the original main method.

**Implementation:**

public class Q1 {

// 1. Overloading of the main method is possible or not?

public static void main(String[] args) {

System.out.println("This is the original main method.");

// Call the overloaded main method.

main(10, 20);

}

public static void main(int a, int b) {

System.out.println("This is the overloaded main method.");

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

The overloaded main method was only executed because we called it from the original main method.

It is important to note that overloading the main method is not a common practice. It is generally considered to be unnecessary and confusing. However, there are some rare cases where it may be useful to overload the main method.

For example, you might want to overload the main method to provide different entry points for your program, depending on the command-line arguments.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Can we create a program without the main method? How many main methods are allowed in Java Programs?**

It is not possible to create a Java program without the main method. The main method is the entry point for all Java programs. It is the method that is called when the program is run.

We can have only one main method in a Java program. If we try to have multiple main methods, the compiler will generate an error.

There is a way to work around this limitation by using static blocks. Static blocks are executed before the main method is called. However, this is not a good practice, and it is not recommended.

**Here is an example of a Java program with a main method:**

public class Main {

public static void main(String[] args) {

System.out.println("Hello, world!");

}

}

**Here is an example of a Java program without a main method:**

public class Main {

// No main method

}

**If we try to compile this program, the compiler will generate the following error:**

error: Main class Main does not have a main method.

This is because the compiler requires all Java programs to have a main method.

In conclusion, it is not possible to create a Java program without the main method. We can have only one main method in a Java program.

1. **What are the six ways to use this keyword?**

This keyword in Java can be used in six ways:

1. **Accessing Instance Variables:** Using this to set and get the value of the value instance variable.
2. **Invoking Constructors (Constructor Chaining):** Calling one constructor from another constructor using this.
3. **Returning the Current Instance:** Modifying the object and returning the current instance for method chaining.
4. **Passing "this" as an Argument:** Passing the object itself as an argument to a method.
5. **Returning "this" from a Method:** Returning the current instance from a method.
6. **Referring to Inner Class Instances:** Accessing the outer class's instance variable from an inner class using MyClass.this.

**Implementation:**

class MyClass {

private int value;

// 1. Accessing Instance Variables

public void setValue(int value) {

this.value = value;

}

public int getValue() {

return this.value;

}

// 2. Invoking Constructors (Constructor Chaining)

public MyClass() {

this(0); // Calling another constructor with an argument

}

public MyClass(int value) {

this.value = value;

}

// 3. Returning the Current Instance

public MyClass modifyValue(int newValue) {

this.value = newValue;

return this;

}

// 4. Passing "this" as an Argument

public void printObjectInfo() {

System.out.println("Object Info: " + this);

}

// 5. Returning "this" from a Method

public MyClass getObject() {

return this;

}

// 6. Referring to Inner Class Instances

public class InnerClass {

public void printOuterValue() {

System.out.println("Outer Value from Inner Class: " + MyClass.this.value);

}

}

}

public class Q3 {

public static void main(String[] args) {

// Creating an instance of MyClass

MyClass obj1 = new MyClass();

// 1. Accessing Instance Variables

obj1.setValue(42);

System.out.println("Value: " + obj1.getValue());

// 2. Invoking Constructors (Constructor Chaining)

MyClass obj2 = new MyClass(10);

System.out.println("Value: " + obj2.getValue());

// 3. Returning the Current Instance

obj1.modifyValue(100).modifyValue(200);

System.out.println("Modified Value: " + obj1.getValue());

// 4. Passing "this" as an Argument

obj1.printObjectInfo();

// 5. Returning "this" from a Method

MyClass obj3 = obj1.getObject();

System.out.println("Are obj1 and obj3 the same object? " + (obj1 == obj3));

// 6. Referring to Inner Class Instances

MyClass.InnerClass inner = obj1.new InnerClass();

inner.printOuterValue();

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **Prove that multiple inheritance is not supported in Java?**

Java does not support multiple inheritance for classes. Multiple inheritance is a feature that allows a class to inherit properties and behavior from more than one parent class, and it can lead to the "diamond problem," which causes ambiguity in method and variable resolution. To avoid these issues, Java uses a mechanism called interface-based multiple inheritance.

**Example that illustrates why Java doesn't support multiple inheritance for classes:**

class Parent1 {

void showMessage() {

System.out.println("Message from Parent1");

}

}

class Parent2 {

void showMessage() {

System.out.println("Message from Parent2");

}

}

// This will result in a compilation error because Java does not allow

// a class to inherit from multiple classes.

class Child extends Parent1, Parent2 {

// Child class inherits showMessage() method from both Parent1 and Parent2

// causing an ambiguity.

}

public class Main {

public static void main(String[] args) {

Child child = new Child();

child.showMessage(); // This would be ambiguous if it were allowed.

}

}

**In this example:**

we have two parent classes, Parent1 and Parent2, both of which have a showMessage method. We then attempt to create a Child class that inherits from both Parent1 and Parent2. This will result in a compilation error because Java does not support multiple inheritance for classes.

**Multiple Inheritance through interface:**

To achieve multiple inheritance of behavior in Java, you can use interfaces. Interfaces allow a class to implement multiple interfaces, which effectively provides a form of multiple inheritance for behavior without the ambiguity associated with inheriting multiple classes.

**Implementation:**

interface Parent1 {

void showMessage1();

}

interface Parent2 {

void showMessage2();

}

class Child implements Parent1, Parent2 {

public void showMessage1() {

System.out.println("Message from Parent1");

}

public void showMessage2() {

System.out.println("Message from Parent2");

}

}

public class Main {

public static void main(String[] args) {

Child child = new Child();

child.showMessage1();

child.showMessage2();

}

}

In this example, the Child class implements both Parent1 and Parent2 interfaces, allowing it to inherit behavior from both interfaces without the ambiguity problem associated with multiple inheritance of classes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **How to override the static method?**

Static methods cannot be overridden in Java. Overriding is a feature of object-oriented programming that allows a subclass to provide its own implementation of a method that is already defined in its parent class. This is achieved through dynamic binding, which means that the method that is called is determined at runtime based on the type of the object that the method is called on.

Static methods, on the other hand, are bound at compile time, meaning that the method that is called is determined based on the type of the class that the method is called on. This is because static methods do not belong to any particular object, but rather to the class itself.

As a result of this difference in binding, it is not possible to override static methods in Java. If you try to declare a static method in a subclass with the same signature as a static method in the parent class, the compiler will generate an error.

Here is an example of a Java program that tries to override a static method:

**Implementation:**

class A {

public static void show() {

System.out.println("Hello from Parent");

}

}

class B extends A {

public static void show() {

System.out.println("Hello from Child");

}

}

public class Q5 {

public static void main(String[] args) {

A obA =new A();

obA.show();

B obB = new B();

obB.show();

}

}

* Static methods are bound at compile time, while method overriding relies on dynamic binding.
* This difference in binding means that it is not possible to override static methods in Java.
* If you try to declare a static method in a subclass with the same signature as a static method in the parent class, the compiler will generate an error.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*