**Name:**

Haleema Binte Jamil

**Registration Number:**

FA20-BSE-005

**Department:**

Computer Science

**Bachelor’s In:**

Software Engineering

**Semester:**

7h

**Section:**

A

**Subject:**

Design Patterns

**Topic:**

Lab 1

**Submitted To:**

Sir Mukhtiar Zamin

**Submission Date:**

07/09/2023

**Lab 1**

<https://github.com/haleema-jamil/DesignPatterns.git>

* Abstraction:

package abstraction;

abstract class Shape {

String name;

public Shape(String name) {

this.name = name;

}

abstract double area();

void displayInfo() {

System.out.println("Name: " + name);

}

}

class Circle extends Shape {

double radius;

public Circle(String name, double radius) {

super(name);

this.radius = radius;

}

@Override

double area() {

return Math.PI \* radius \* radius;

}

}

class Rectangle extends Shape {

double length;

double width;

public Rectangle(String name, double length, double width) {

super(name);

this.length = length;

this.width = width;

}

@Override

double area() {

return length \* width;

}

}

public class Abstraction {

public static void main(String[] args) {

Circle circle = new Circle("Circle", 5.0);

Rectangle rectangle = new Rectangle("Rectangle", 4.0, 6.0);

circle.displayInfo();

System.out.println("Area: " + circle.area());

rectangle.displayInfo();

System.out.println("Area: " + rectangle.area());

}

}

* Encapsulation:

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age >= 0) {

this.age = age;

}

}

public void displayInfo() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

public static void main(String[] args) {

Person person = new Person("Binte Jamil", 21);

System.out.println("Original Name: " + person.getName());

person.setName("Haleema");

System.out.println("Updated Name: " + person.getName());

System.out.println("Original Age: " + person.getAge());

person.setAge(-5);

System.out.println("Updated Age: " + person.getAge());

person.displayInfo();

}

}

* Inheritance:

class Animal {

String name;

public Animal(String name) {

this.name = name;

}

public void eat() {

System.out.println(name + " is eating.");

}

public void sleep() {

System.out.println(name + " is sleeping.");

}

}

class Dog extends Animal {

public Dog(String name) {

super(name);

}

public void bark() {

System.out.println(name + " is barking.");

}

}

public class Inheritance {

public static void main(String[] args) {

Dog dog = new Dog("Buddy");

dog.eat();

dog.sleep();

dog.bark();

}

}

* Polymorphism:

class Animal {

public void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Cat meows");

}

}

public class Polymorphism {

public static void main(String[] args) {

Animal animal = new Animal();

Dog dog = new Dog();

Cat cat = new Cat();

Animal[] animals = { animal, dog, cat };

for (Animal a : animals) {

a.makeSound();

}

}

}

* Association:

class University {

private String name;

public University(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

class Student {

private String name;

private University university;

public Student(String name, University university) {

this.name = name;

this.university = university;

}

public String getName() {

return name;

}

public University getUniversity() {

return university;

}

}

public class Association {

public static void main(String[] args) {

University university = new University("COMSATS Abbottabad");

Student student1 = new Student("Haleema", university);

Student student2 = new Student("Marwa", university);

System.out.println(student1.getName() + " is a student at " + student1.getUniversity().getName());

System.out.println(student2.getName() + " is a student at " + student2.getUniversity().getName());

}

}

* Aggregation:

import java.util.ArrayList;

import java.util.List;

class Department {

private String name;

public Department(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

class Employee {

private String name;

private Department department;

public Employee(String name, Department department) {

this.name = name;

this.department = department;

}

public String getName() {

return name;

}

public Department getDepartment() {

return department;

}

}

public class Aggregation {

public static void main(String[] args) {

Department HR = new Department("Human Resources");

Department IT = new Department("Information Technology");

Employee employee1 = new Employee("Haleema", HR);

Employee employee2 = new Employee("Marwa", IT);

Employee employee3 = new Employee("Tooba", HR);

List<Employee> HREmployees = new ArrayList<>();

HREmployees.add(employee1);

HREmployees.add(employee3);

System.out.println(employee1.getName() + " works in " + employee1.getDepartment().getName());

System.out.println(employee2.getName() + " works in " + employee2.getDepartment().getName());

System.out.println(employee3.getName() + " works in " + employee3.getDepartment().getName());

System.out.println("Employees in HR department:");

for (Employee employee : HREmployees) {

System.out.println("- " + employee.getName());

}

}

}

* Composition:

class Engine {

public void start() {

System.out.println("Engine started");

}

}

class Car {

private String brand;

private Engine engine;

public Car(String brand) {

this.brand = brand;

this.engine = new Engine();

}

public void startCar() {

System.out.println(brand + " car is starting...");

engine.start();

}

}

public class Composition {

public static void main(String[] args) {

Car myCar = new Car("Toyota");

myCar.startCar();

}

}

**Q: Overloading of main method is possible in java or not?**

**A:** Yes, overloading of the main method is possible in Java.

* When we overload a method, we define multiple methods in the same class with the same name but with different parameters.
* In the case of the main method, Java requires the method signature to be `public static void main(String[] args)` for the standard entry point of a Java application.
* However, you can define overloaded `main` methods with different parameter lists, and they will be treated as regular methods rather than the entry point of the program.

Here's an example demonstrating overloading of the `main` method in Java:

public class MainMethodOverloading {

public static void main(int x) {

System.out.println("This is the int main method with parameter: " + x);

}

public static void main(String[] args) {

System.out.println("This is the standard main method with args.");

for (String arg : args) {

System.out.println("Argument: " + arg);

}

}

}

In this example, we have overloaded the `main` method with different parameter type `int`. This overloaded `main` method can be called like any other methods within the class.

**Q: What is returned by the constructor and how can you identify it from its declaration?**

**A:** In Java, constructors are special methods used to initialize objects when they are created.

* Constructors do not specify a return type, not even Void.
* You can identify a constructor from its declaration by the following characteristics:
  + Constructors have the same name as the class in which they are declared.
  + Constructors do not specify a return type.
  + Constructors are called when you create a new instance of an object using the `new` keyword.
  + Constructors can have parameters, allowing you to initialize the object with specific values. Parameterized constructors are used to pass arguments when creating an object.

**Q: Can we create a program without main method? How many main methods are allowed in java program?**

**A:** No, a Java program must have a `public static void main(String[] args)` method to serve as the entry point of the program. The Java Virtual Machine (JVM) looks for this specific method to start the program execution.

**Q: What are the six ways to use “this” keyword?**

**A:** The "this" keyword in Java can be used in six different ways:

1. To differentiate between instance variables and method parameters: `this.variableName`
2. To call another constructor of the same class: `this()`
3. To pass itself as an argument to another method or constructor: `methodName(this)` or `new AnotherClass(this)`
4. To return the current instance from a method: `return this`
5. To refer to the current class's instance methods: `this.methodName()`
6. To invoke the constructor of the parent class in case of inheritance: `super()`

**Q: Prove that multiple inheritance is not supported in java.**

**A:** Java does not support multiple inheritance through classes. You can achieve multiple inheritance in Java through interfaces. This is because multiple inheritance with classes can lead to the "diamond problem" and other complexities. Java's design choice was to avoid these issues by allowing multiple inheritance through interfaces.

**Q: When to use aggregation and not composition and vice versa?**

**A:**

* Use composition when one class is a part of another class, and the part cannot exist independently.
* Use aggregation when one class is associated with another class, but the associated class can exist independently.

**Q: How to override the static method?**

**A:** Static methods cannot be overridden in Java because they are associated with the class rather than instances of the class. However, you can have a static method in a subclass with the same name as a static method in the superclass. This is called method hiding.

**Q: Is overloading of main method possible?**

**A:** Yes, overloading of the main method is possible in Java.

* When we overload a method, we define multiple methods in the same class with the same name but with different parameters.
* In the case of the main method, Java requires the method signature to be `public static void main(String[] args)` for the standard entry point of a Java application.
* However, you can define overloaded `main` methods with different parameter lists, and they will be treated as regular methods rather than the entry point of the program.

**Q: Give any real world example of using the covariant return type.**

**A:** Covariant return types are used when a subclass overrides a method from the superclass and returns a subtype of the type returned by the superclass method.

* For example, in Java 5 and later versions, the `java.util.ArrayList` class overrides the `clone` method and returns `Object`. In Java 5 and later, it returns `ArrayList<E>`, which is more specific.

**Q: Discuss different usages of Java “super” keyword.**

**A:** The "super" keyword in Java is used for various purposes:

* To call the constructor of the parent class: `super()`
* To call the parent class's method: `super.methodName()`
* To access the parent class's instance variables: `super.variableName`

**Q: What is instance initializer block and why we use it?**

**A:** An instance initializer block is a block of code within a class that is executed when an instance of the class is created. It is used to initialize instance variables or perform some other operations during object creation. It is especially useful when multiple constructors exist in a class, as it centralizes common initialization code.

**Q: What are different usages of “final” variable?**

**A:** A "final" variable in Java can be used for the following purposes:

* To create constants.
* To prevent the value of a variable from being changed after initialization.
* To indicate that a variable should not be overridden in a subclass (when used with methods).
* To ensure thread safety in multi-threaded applications when used with shared variables.

**Q: What is a marker or tagged interface?**

**A:** A marker or tagged interface is an interface in Java that does not declare any methods. It is used to mark or tag classes that implement the interface, indicating that they possess certain characteristics or should be treated in a special way by the system or frameworks.

**Q: What is runtime polymorphism or dynamic method dispatch?**

**A:** Runtime polymorphism, also known as dynamic method dispatch, is a mechanism in Java where the appropriate method to invoke is determined at runtime based on the actual object type rather than the reference type. It allows you to override methods in subclasses and have the correct method called based on the object's type.

**Q: What is the difference between static and dynamic binding?**

**A:** Static binding is resolved at compile time and is based on the reference type, whereas dynamic binding is resolved at runtime and is based on the actual object type.

**Q: How down-casting is possible in java?**

**A:** Down-casting is possible in Java when you have a reference to a superclass object, and you want to cast it to a subclass reference type. This can be done explicitly using casting, but it may result in a `ClassCastException` if the object is not actually an instance of the subclass.

**Q: What is the purpose of private constructor?**

**A:** A private constructor is used to prevent the instantiation of a class from outside the class itself. It is often used in utility classes where all methods are static, and you don't want to create instances of the class.

**Q: What is object cloning?**

**A:** Object cloning in Java is the process of creating a copy of an existing object. It can be achieved by implementing the Cloneable interface and overriding the clone method. Cloning can be shallow or deep, depending on whether it copies only the references to objects (shallow) or the objects themselves (deep).

**Q: Differentiate shallow copy from deep copy and implement each with an example.**

**A:** Shallow copy copies references to objects, so changes in the copied objects are reflected in the original objects. Deep copy creates entirely new objects, so changes in the copied objects do not affect the original objects. Implementing both types of copies requires overriding the clone method and creating a custom copy logic.

**Q: In your class context, find the implementation of OOPs Concept.**

**A:** <https://raygun.com/blog/oop-concepts-java/>