

Design Patterns



HAMZA ATIF

FA20-BSE-018

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

No, overloading the main method is not possible in Java. The main method has a specific signature, and it serves as the entry point for a Java program. It must be declared as public static void main(String[] args).

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

A constructor in Java does not return a value explicitly. It is responsible for initializing and constructing an object of a class. You can identify a constructor from its declaration by looking for the following characteristics:

It has the same name as the class.

It does not have a return type (not even void).

It typically includes initialization code for the object.

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

In a standard Java program, you must have a main method to serve as the entry point of the program. You can create multiple classes with their own main methods, but each class can have only one main method.

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

The this keyword in Java can be used in the following six ways:

To differentiate between instance variables and method parameters.

To call another constructor from the same class using this().

To call another method within the same class.

To return the current instance from a method (chaining).

To pass the current instance as an argument to another method.

To refer to the current class's instance variables or methods in a constructor.

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

Java does not support multiple inheritance for classes. It uses a single inheritance model, meaning a class can inherit from only one superclass. This is done to avoid the "diamond problem," where ambiguity can arise when a subclass inherits from two or more superclasses that have the same method or field names.

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

Use composition when the child object is a fundamental part of the parent object and has a strong lifecycle dependency. Use aggregation when the child object is loosely coupled with the parent and can exist independently.

**How to override a static method?**

You cannot override static methods in Java. Static methods belong to the class itself and are not associated with instances of the class. Subclasses can hide static methods by declaring methods with the same signature, but this is not considered method overriding.

**Give any real-world example of using the covariant return type?**

An example of covariant return types is in method overriding, where a subclass can override a method from the superclass with a more specific return type. For instance, in Java 5 and later, the Object.clone() method returns an object of the specific class type rather than an Object.

**Discuss different usages of the Java super keyword?**

The super keyword is used to:

Call a superclass's constructor.

Access a superclass's method or variable when it's overridden in the subclass.

Invoke a superclass's constructor with arguments.

Access a superclass's constructor or method from a subclass.

**What is an instance initializer block, and why do we use it?**

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 for initializing instance variables or performing complex initialization logic that cannot be achieved in a constructor.

**What are the different usages of final variables?**

Final variables in Java can be used for:

Creating constants.

Ensuring immutability in objects.

Enforcing initialization before use in constructors.

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

A marker or tagged interface is an interface in Java that doesn't declare any methods. Its purpose is to mark classes that implement it, indicating that they possess certain characteristics or should be treated in a special way by the code.

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

Runtime polymorphism, also known as dynamic method dispatch, is a feature in Java where the method to be executed is determined at runtime based on the actual object type rather than the reference type. It allows for method overriding and enables more flexible and extensible code.

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

Static binding is determined at compile-time, where the method to be invoked is based on the reference type. Dynamic binding, on the other hand, is determined at runtime, where the method to be invoked is based on the actual object type.

**How is down-casting possible in Java?**

Down-casting in Java can be done by explicitly casting a reference of a superclass type to a reference of a subclass type. It is allowed, but it can result in a ClassCastException at runtime if the object being referred to is not an instance of the subclass.

**What is the purpose of a private constructor?**

A private constructor is used to prevent the instantiation of a class from outside the class itself. It is often employed in utility classes where you want to group related methods together but don't want instances of the class to be created.

**What is object cloning?**

Object cloning in Java refers to the process of creating a new object that is a copy of an existing object. You can clone an object either by implementing the Cloneable interface and using the clone method or by creating a new object and manually copying the values of the fields.

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

A shallow copy duplicates the object itself but not the objects referenced by it. A deep copy duplicates both the object and all objects referenced by it.

Shallow Copy Example:

java

class Person {

String name;

Address address;

public Person(String name, Address address) {

this.name = name;

this.address = address;

}

public Object clone() throws CloneNotSupportedException {

return super.clone();

}

}

class Address {

String street;

public Address(String street) {

this.street = street;

}

}

public class Main {

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

Address addr = new Address("123 Main St");

Person person1 = new Person("Alice", addr);

Person person2 = (Person) person1.clone();

// Both person1 and person2 share the same Address object

System.out.println(person1.address == person2.address); // true

}

}

Deep Copy Example (using serialization):

java



import java.io.\*;

class Person implements Serializable {

String name;

Address address;

public Person(String name, Address address) {

this.name = name;

this.address = address;

}

public Person deepCopy() throws IOException, ClassNotFoundException {

ByteArrayOutputStream bos = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(bos);

out.writeObject(this);

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());

ObjectInputStream in = new ObjectInputStream(bis);

return (Person) in.readObject();

}

}

class Address implements Serializable {

String street;

public Address(String street) {

this.street = street;

}

}

public class Main {

public static void main(String[] args) throws IOException, ClassNotFoundException {

Address addr = new Address("123 Main St");

Person person1 = new Person("Alice", addr);

Person person2 = person1.deepCopy();

// person1 and person2 have separate Address objects

System.out.println(person1.address == person2.address); // false

}

}

**In your class context, find the implementation of OOPs concepts.**

Implementing OOP concepts in a class context typically involves using class, object, inheritance, polymorphism, encapsulation, and abstraction. Here's a brief example:

java

// Class definition (Abstraction)

class Animal {

private String name; // Encapsulation

public Animal(String name) {

this.name = name;

}

public void eat() {

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

}

public void makeSound() {

System.out.println(name + " makes a sound.");

}

}

// Inheritance (Polymorphism)

class Dog extends Animal {

public Dog(String name) {

super(name);

}

@Override

public void makeSound() {

System.out.println(getName() + " barks.");

}

public void fetch() {

System.out.println(getName() + " fetches a ball.");

}

}

class Cat extends Animal {

public Cat(String name) {

super(name);

}

@Override

public void makeSound() {

System.out.println(getName() + " meows.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog("Buddy");

Cat cat = new Cat("Whiskers");

dog.eat();

dog.makeSound();

dog.fetch();

cat.eat();

cat.makeSound();

}

}

In this example:

We define a class hierarchy (inheritance) with Animal, Dog, and Cat classes.

We use encapsulation by making the name field private and providing getter methods (getName) to access it.

We demonstrate polymorphism by overriding the makeSound method in the Dog and Cat classes.

We create objects (instances) of the classes and call their methods.

We achieve abstraction by creating an abstract concept of an Animal class that can be extended by specific animal types like Dog and Cat.

This example illustrates the fundamental OOP concepts of Java in a class context.